LY33-6010-1

Program Product

DOS
PL/I Optimizing Compiler
Program Logic

Program Number 5736-PL1
(This product is also distributed as
part of composite package 5736-PL3)

Feature Number 8050

IBM

Order No. LY33-6010-1, Page Revised by TNL LN33-6175, October 1976

Second Edition (June, 1972)

This is a major revision of, and obsoletes, LY33-6010-0. Changes or additions to the text and figures are indicated by a vertical line to the left of the change.

This edition applies to Version 1, Release 5, Modification 0 of the DOS PL/I Optimizing Compiler, Program Product 5736-PL1, and to any subsequent version, release, and modification.

Information in this publication is subject to significant change. Any such changes will be published in new editions or technical newsletters. Before using the publication, consult the latest IBM System/370 Bibliography, GC20-0001, and the technical newsletters that amend the bibliography, to learn which edition and technical newsletters are applicable and current.

Requests for copies of IBM publications should be made to the IBM branch office that serves you.

Forms for readers' comments are provided at the back of the publication. If the forms have been removed, comments may be addressed to IBM Corporation, P.O. Box 50020, Programming Publishing, San Jose, California 95150. All comments and suggestions become the property of IBM.

Preface

The internal logic of the DOS PL/I Optimizing Compiler is described in this manual. It is written for use by people involved in program maintenance or in modification of the program design. The manual consists of seven sections, organized as follows:

Section 1: Introduction
Section 2: Method of Operation
Section 3: Program Organization
Section 4: Directory
Section 5: Data Area Layouts
Section 6: Diagnostic Aids
Section 7: Appendixes

This organization is intended to enable ease of access when the manual is used either for initial education purposes or for reference purposes.

For readers who are not familiar with the compiler, descriptive text and illustrations are contained in sections 1 and 2, and in the first part of section 3. Section 1 contains brief descriptions of the purpose and capabilities of the compiler, and of its relationship to the Disk Operating System. The first part of section 2 contains a general description of compiler operation, and descriptions of the housekeeping features that are common to all aspects of compiler operation. The main part of section 2 consists of descriptions of the functions and methods of operation of component sections of the compiler. These descriptions contain references to the figures in section 5 that illustrate the formats of various data areas. The overall physical organization of the compiler is described in the first part of section 3.

When the manual is used for reference purposes, such as diagnosis of possible errors in compiler operation, initial access via the directory in section 4 is recommended. If the compilation of a particular statement is to be examined, the first list in the directory indicates the phases of the compiler that process particular PL/I language features. If the execution of a particular compiler phase is being examined, the second directory list indicates the processing functions performed by each individual phase. The lists in section 3, which show the functions of the main sections of code within each phase, can be used to identify the approximate position of a section of code within a phase listing. Each phase list in section 3 is accompanied by a flowchart for the phase. Section 6 contains details of the various diagnostic aids in the compiler, and describes how they can be made available when detailed examination of compiler operation is required.

The attention of all readers is drawn to the two fold-out figures in appendixes C and D. The first shows the sequences of phase loading that are used in various circumstances. The second shows the main data areas that may be accessed during the execution of any phase. Special reference information, such as the functions of macros used in the compiler, and the causes of compiler error messages, is contained in other appendixes.

PREREQUISITE PUBLICATIONS

To enable effective use to be made of this manual, an understanding of the contents of the following publications is required:

DOS PL/I Optimizing Compiler:
Language Reference Manual
Order No. SG33-0005

IBM System/360 DOS/TOS
Assembler Language
Order No. GC24-3414

ASSOCIATED PUBLICATIONS

The following publications are associated with the program product described in this manual, and should be consulted as required:

DOS PL/I Optimizing Compiler:
Messages
Order No. SC33-0021

DOS PL/I Optimizing Compiler:
Installation
Order No. SC33-0020

DOS PL/I Optimizing Compiler:
Execution Logic
Order No. SC33-0019

DOS PL/I Optimizing Compiler:
Programmer's Guide
Order No. SC33-0008

DOS PL/I Optimizing Compiler:
General Information
Order No. GC33-0004

DOS PL/I Resident Library:
Program Logic
Order No. LY33-6011

DOS PL/I Transient Library:
Program Logic
Order No. LY33-6012

AVAILABILITY OF PUBLICATIONS

The availability of a publication is indicated by its use key, the first letter in the order number. The use keys and their indications are:

G - General: available to all users of IBM systems, products, and services without charge, in quantities to meet their normal requirements; can also be purchased by anyone through IBM branch offices.

S - Sell: can be purchased by anyone through IBM branch offices.

L - Licensed material, property of IBM: available only to licensees of the related program products under the terms of the license agreement.

Contents

Figures

Charts

Section 1: Introduction

The PL/I Optimizing Compiler is a multi-phase, multi-pass compiler which operates as a processing program under System/360, System/370 Disk Operating System. It analyzes and processes source programs written in the PL/I language as described in the publication, DOS PL/I Optimizing Compiler: Language Reference Manual. The compiler is written in System/360 Assembler Language. Extensive use is made of specially designed macro instructions.

PURPOSE OF THE COMPILER

The purpose of each execution of the compiler is the translation of a PL/I external procedure into a series of machine instructions which form a relocatable object module. One or more object modules produced by the compiler can be link-edited to form an executable program phase, and a punched object deck can optionally be produced. A number of other facilities, such as printed listings of source programs and object modules, can be provided in response to programmer-specified compiler options. If errors are detected in a source program, appropriate diagnostic messages are generated and can be printed. In order that compilation can be continued, erroneous statements may be ignored, or suitable corrections may be attempted within the compiler.

An important feature of this compiler is that it performs code optimization, i.e., the compiler recognizes situations where it can generate machine code that can be executed more efficiently than code produced by direct translation of the PL/I source program. Two main forms of optimization are performed. One is the optimization that is performed when particular language features or situations are recognized by various sections of the compiler during any compilation. This form of optimization is referred to as local optimization, and is not optional. The other main form of optimization is referred to as global optimization, and is optional. It is performed by a particular section of the compiler, which is only executed in response to programmer specification of the OPTIMIZE compiler option. This option enables the programmer to specify that the program is to be modified in such a way that less time is required for execution of the object program. This optimization may also have a secondary effect of reducing the amount of storage required for the object module.

The code generated by the compiler also has good performance characteristics with virtual storage systems. The modularity of PL/I and the way code and data are separated into different CSECTs by the compiler, assist in minimizing the impact of a PL/I program on the paging of a virtual storage system.

The machine instructions in the object module do not always reflect all the operations indicated by the PL/I source program. Certain types of statement are translated into instructions that call standard subroutines held in a library. These subroutines perform the required operation, or may in turn call other library subroutines. The library calls are resolved during link-editing. Descriptions in this manual assume that the compiler is used in conjunction with two IBM program products: the DOS PL/I Resident Library (Program Number 5736-LM4) and the DOS PL/I Transient Library (Program Number 5736-LM5).

THE COMPILER AND THE DISK OPERATING SYSTEM

The compiler acts as a processing program under the control of the Disk Operating System. It requires a partition of at least 44K bytes, and can only be used in the batched-job processing mode. The compiler can be used in the background partition and, if the multiprogramming option of the operating system is used, it can be link-edited into private core-image libraries for compile-link-go operation in the F1 and/or F2 foreground partitions. Further requirements and implementation-defined features are listed in the Programmer's Guide for this compiler.

The name of the compiler program is PLIOPT. The compiler consists of a number of phases. Each phase can be referred to by one of two symbolic names. In the relocatable library, each phase is referred to by a name beginning with the characters IEL0, e.g., IEL0AE, IEL0GI, IEL0KX, etc. In the core-image library, each phase is referred to by a name beginning with characters PLIO, e.g., PLIOAE, PLIOGI, PLIOKX, etc. Except where it is necessary to refer specifically to one of these names, phases are referred to in this manual by the last two characters of their symbolic names, e.g., Phase AE, Phase GI, Phase KX, etc. An exception to this convention is the resident control phase. In the relocatable library, this phase has the name IEL0AA, but in the core-image library the phase has the same name as the compiler program, PLIOPT. To avoid confusion, the resident control phase is referred to as Phase AA.

The first compiler phase to be loaded is the compiler control phase, Phase AA, which remains in main storage throughout the compilation process. Among its many functions, this phase provides an interface between the compiler and the Disk Operating System. It communicates with the DOS control programs for loading of other compiler phases, and input/output operations between main storage and the data sets used for spill purposes by the compiler.

Apart from Phase AA, each phase of the compiler is loaded in turn into an area of main storage allocated as a phase area. Because the execution of certain compiler phases is optional, depending upon the compiler options specified or upon the contents of the PL/I source program, some of the compiler processing phases may not be loaded for every compilation. When a compiler phase has completed its processing, it uses a special macro instruction, XPST, to identify the next phase to be loaded. Phase AA passes the name of the specified phase as an argument to the DOS control program when requesting the loading of another phase.

On completion of compilation, the compiler optionally writes the compiled object module onto the SYSLNK data set ready for link-editing. Unless further compilations have been specified (as part of a batched-job) by use of a *PROCESS control statement, control is then returned to the DOS control program.

COMPILER INPUT AND OUTPUT

A number of data sets are accessed or created by the compiler. The input/output devices used for these data sets are shown in figure 1.1.

Input data read from SYSIPT consists of one or more PL/I external procedures. The batched-compilation facility of the compiler enables more than one external procedure to be compiled in a single job step. Each external procedure may be a complete program or part of a program. An external procedure can contain %INCLUDE statements that specify source-statement modules held in a private source-statement library. In such cases, the specified source module is read from the device assigned to SYSSLB, and incorporated in the source data.

Throughout this manual the term PL/I source program is used to refer in general to external data passed to the compiler for processing. The

term text is used to refer to the main stream of internal data, consisting of the internal representation of statements and other items of information, originally corresponding to the PL/I source program and progressively transformed by phases of the compiler into the format required at output. During compilation, data is extracted or derived from the text and collected in various tables, lists, etc. The term dictionary is used to refer to a particular collection of data, used extensively during compilation.

Data Set	Function	Device Type	Device Symbolic Name	When Required
File name = IJSYSIN DTF name = XINPUT	Input	DASD Magnetic tape Card reader	SYSIPT	Always
Source Statement Library	Input	DASD	SYSSLB	When preprocessor %INCLUDE is used
File name = IJSYSLS DTF name = XPRINT	Listings	DASD Magnetic tape Printer	SYSLST	Always
File name = IJSYS01 DTF name = XSPILL1	Data spill	DASD	SYS001	Always
File name = IJSYS02 DTF name = XSPILL2	Data spill	DASD	SYS002	Always
File name = IJSYSLN DTF name = XLOADF	Output to linkage editor	DASD Magnetic tape	SYSLNK	When linkage editing follows compilation in the same job
File name = IJSYSPH DTF name = XPUNCH	Output to linkage editor (card deck)	DASD Magnetic tape Card punch	SYSPCH	When linkage editing takes place in a subsequent job
Core-image Library	Compiler residence	DASD	SYSRES	Always

Figure 1.1. Data sets and input/output devices used by the compiler

Data sets on the devices assigned to SYS001 and SYS002 are used to hold internal data (text, dictionary, etc.,) that is not currently being accessed or processed, and which cannot be retained in main storage. These data sets are known as the spill data sets. Input/output operations between main storage and the spill data sets take place throughout most compilations.

Output from the compiler consists of one or more object modules that are suitable for link-editing and inclusion in an executable program phase. Each object module consists of an external symbol dictionary (ESD), a relocation dictionary (RLD), and a series of machine instructions in the form of TXT records. A description of an object module is given in the publication DOS: System Control and System Service Programs.

Output from the compiler is in the form of fixed-length 322-byte records, which can be transmitted to a data set on the device assigned to SYSLNK. Output can also be transmitted to a data set on the device assigned to SYSPCH, in which case the output is in the form of 80-byte unblocked records.

During compilation, a listing can be optionally generated and transmitted to a data set on the device assigned to SYSLST, and can be printed. The following list shows the information that can be included in the listing, and the options required.

Listing	Options Required
Options for the compilation	OPTIONS
Preprocessor input	INSOURCE
Source program	SOURCE
Statement nesting level	NEST
Attribute table	ATTRIBUTES
Cross-reference table	XREF
Aggregate-length table	AGGREGATE
External symbol dictionary	ESD
Items in static storage	MAP
Object module	LIST
Storage requirements	STORAGE
Statement offsets	OFFSET
Diagnostic messages for severe errors, errors, warnings and information conditions	FLAG(S), FLAG(E), FLAG(W), FLAG(I)

Source statements from SYSSLB are read into buffers in the main storage used by the preprocessor phase. Input/output operations between main storage and the spill file are made directly. All other input/output is read into, or written from, buffer areas in main storage which are allocated as follows:

Buffer Name	Size	Function
XIFBF1	80 bytes	Input from SYSIPT
XIFBF2	80 bytes	
XPRBF1	121 bytes	Output to SYSLST
XPRBF2	121 bytes	
XPFBF1	81 bytes	Output to SYSPCH
XPFBF2	81 bytes	
XOFBF1	322 bytes	Output to SYSLNK

GENERAL ORGANIZATION OF THE COMPILER

The compiler consists of 53 physical phases, which are stored in the core-image library on the system residence volume. Each phase can be individually loaded and executed.

When the compiler is invoked, the resident control phase (Phase AA) is loaded and control is passed to it. This phase remains in main storage throughout execution of the compiler. It contains routines which can be entered at any time during compilation to provide services for other phases and to communicate with the DOS control program where necessary, e.g., for loading of phases, or for input/output operations. This phase also contains a control section, with the symbolic name XCOMM, that defines an area of storage used for communication between phases.

Located within this communication area are six of the seven input-output buffers previously mentioned. (The XOFBF1 buffer for output to SYSLNK is allocated in the working storage of Phase SI, the Object-Module-Assembly phase.)

All other phases are loaded individually in sequence into an area of main storage known as the phase area. Only one phase in addition to the control phase can be in main storage at any time. (An exception to this is the requirement for Phase AI to be resident in main storage throughout compilation if a compiler dump is required.) On completion of execution, each phase uses a macro statement to inform the control phase of the name of the next phase to be loaded.

Almost immediately after receiving control, Phase AA has the initialization phase, Phase AE, loaded and passes control to it. Phase AE performs most of the once-only housekeeping required to prepare the compiler operating environment, if necessary making adjustments in accordance with compiler options specifications. In addition to processing compiler options it initializes various fields in the communications area, opens the data sets used by the compiler, and allocates an area of main storage for the storage of data such as text and dictionary tables. An important feature of compiler operation is that this data can be spilled onto auxiliary storage and read back into main storage as required. To facilitate this data handling, the data is organized in records known as pages. Each page can contain 1080,1680, or 3480 bytes of processable data, depending on the size of the main storage area available for the storage of data. If auxiliary storage is on either a 3330 or 3340 direct access storage device, and the SIZE option is large enough, a page size of 4040 bytes is used. The storage area is known as the page area; its size and the size of the pages to be used are calculated by Phase AE after the size of the compiler partition and value of the SIZE option (if specified) have been determined. Space for a minimum of eight pages is required in the page area but, if more space is available, more page spaces and/or the larger page size can be used, thus reducing the time required for compilation by reducing the number of input/output operations required for the spilling and reading of pages. Space within the page area is also allocated for a directory, used to facilitate searches for dictionary entries, and for a resident-page table, used in some page-handling operations.

When Phase AE has completed its functions and returned control to Phase AA, the operating environment is ready for the execution of the processing phases. The general organization of the compiler, the flow of control, and the flow of data is shown in figure 1.2, and the organization of storage is shown in figure 1.3.

Processing phases of the compiler are loaded one at a time into the phase area, and executed. Although the phases are loaded and executed sequentially, the functions performed by certain phases may not be required in some compilations. In such cases the relevant phases are not loaded. For example, unless the relevant compiler options are specified, the preprocessing phases and the global optimization phases are not loaded. Similarly, phases processing specific language features, e.g., stream oriented input/output statements, are not loaded if those features are not present in the source program. The sequence of phase loading is shown in appendix C, and the conditions affecting the sequence of phase loading are shown in figure 3.2.

In general, none of the processing phases is executed more than once in each compilation. Exceptions to this are the phases that produce diagnostic information. In order to satisfy certain compiler options, the phase that edits and prints diagnostic messages, Phase UA, may be loaded and executed twice during a compilation. The phase that provides printed dumps of the contents of data areas used by the compiler, Phase AI, may be executed a number of times to satisfy compiler options. This phase, if required by compiler options, must be resident in main storage throughout compilation, and requires approximately 16K bytes of additional storage.

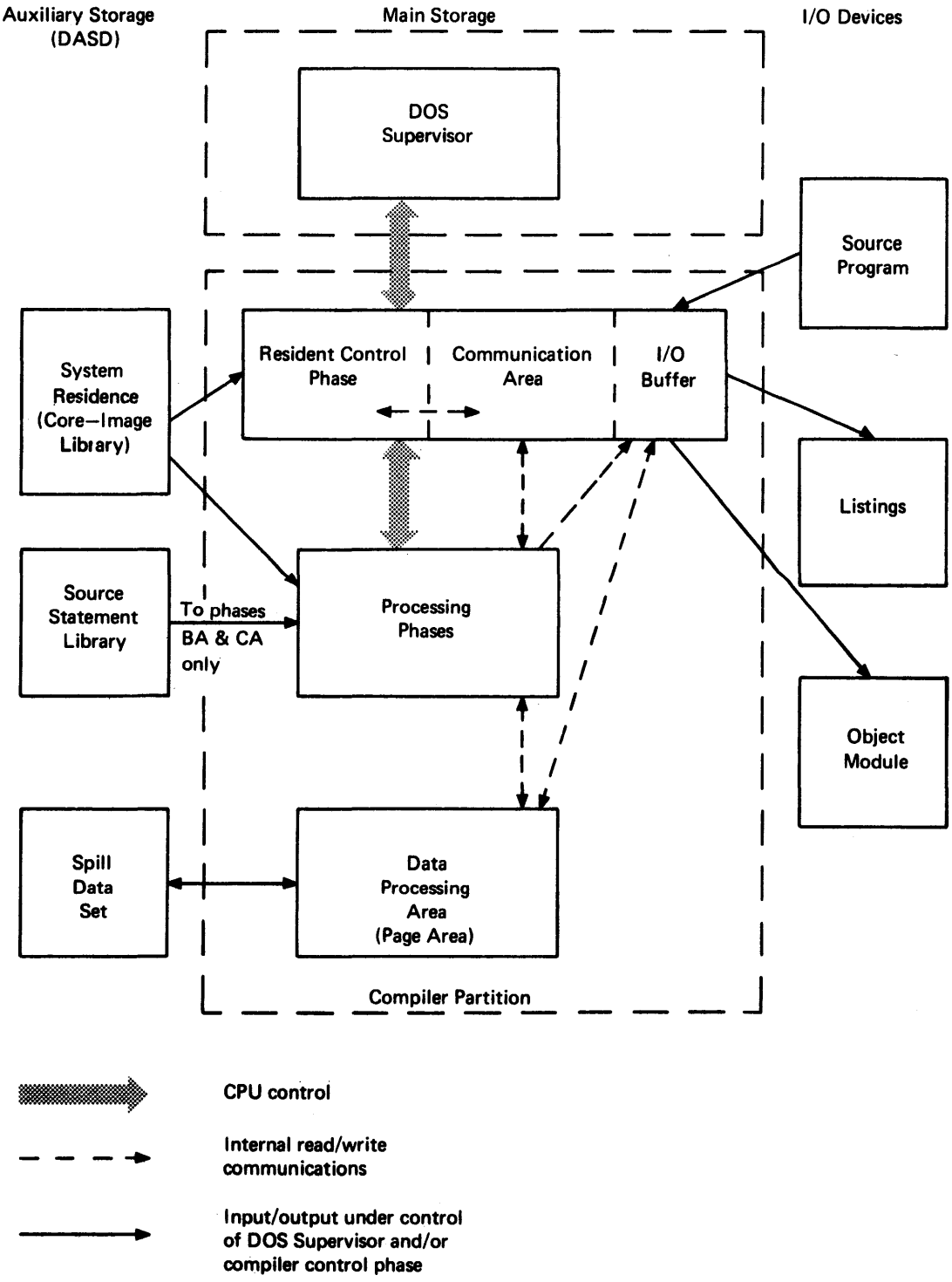


Figure 1.2. General organization of the compiler, showing control and data flow

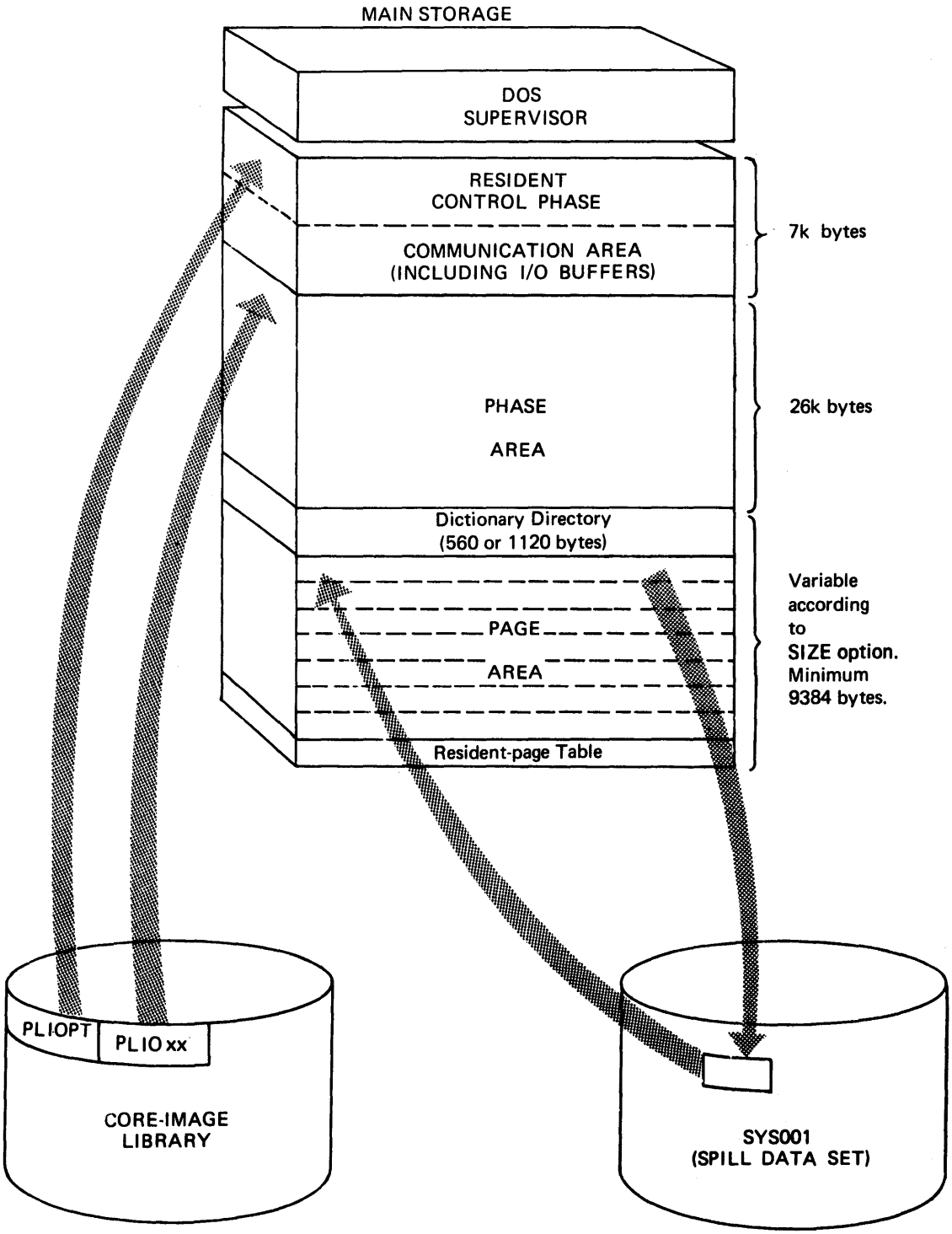


Figure 1.3. General organization of storage

Character Code Dependence

The following table identifies those areas that would require attention if it were intended to modify the compiler to accept its input and print its output in a different character set and/or language. (By language is meant, for example, German or French, not programming language.) The table does not attempt to detail how a particular area or item should be modified for a language/character set change; only the type of dependency that exists is identified. Those compiler phases that are totally independent of any change are listed at the foot of the table.

Phase	Area of dependency
AA	Character string constants for: 1. Error message text for one message. 2. Some phase names.
AE	Character string constants for: 1. File names. 2. Phase names. 3. Heading for SYSLST. 4. Option keywords. 5. Error message text for initialization errors.
AI	Translate table (internal to external code). Hex. to external character translation. Character string constants for, for example: 1. Messages. 2. Phase names. Test for numeric constants. Code for translation from hex. to external character. Comma, full stop, and blank values in trace table routine. Assumption that hex values of characters increase through the alphabet.
BA	
CA	
CE	Message list in XMTAB macro. Keyword list in XMCDE macro. Translate table ZTRAN1 (internal code to EBCDIC).
EA	Module EA1 Headings on source listing. Sequence numbers on input records (EBCDIC). Module EA2 Sequence numbers on input records (EBCDIC). SYSIPT or SYSLST assumed and output when particular ON conditions encountered. Module EA3 External to internal code translate tables (EBCDIC). Keyword tables. Carriage control characters on input records (EBCDIC).
EC	External to internal code translate tables (EBCDIC). Keyword tables. Carriage control characters on input records (EBCDIC).
EE	Module EE2 Keyword tables. In-line tests for some special ENVIRONMENT option parameters which are implementation-defined and could therefore be affected by a change of language, for example, TP(M) and TP(R).

	Module EE3 Explicit text inserts for message IEL0334I concerning missing file options on I/O statements such as READ, WRITE, etc.
	Keyword tables. SYSIPT or SYSLST assumed and output for certain I/O statements.
GA	BIF text tables.
GI	BIF text tables. Check for SYSLST. Table of such items as PLISRTA/B/C, PLIDUMP, etc.
GE	
GM	BIF text tables. Machine representation of external characters.
ID	Machine representation of external characters.
IK	Translate table ZTRAN1 (internal code to EBCDIC).
KL	Messages containing character string SYS. MEDIUM option with SYSIPT, SYSLST, and SYSPCH.
PA	Machine representation of external characters.
SM	
UA	Message list in XMTAB macro. Keyword list in XMCDE macro. Translate table ZTRAN1 (internal code to EBCDIC).

The following phases are independent of any language/character set change:

AS	EI	IA	IE	II	KT
IM	KA	IQ	KE	KI	OI
KM	KQ	KV	OA	OE	PC
OM	KK	OC	OX	KX	SA
PE	PI	QI	QA	QE	
SQ	SD	SC	SK	SI	

INTRODUCTION

The PL/I Optimizing Compiler transforms a PL/I external procedure into a relocatable object module, suitable for link editing and subsequent execution. The process of transformation is known as compilation.

This section contains descriptions of the methods used by the compiler to perform the compilation process. The major operations involved in compilation are shown in relation to the sections of the compiler that perform the required operations. Then follow descriptions of features of compiler operation that are common to all phases. The remainder of the section contains descriptions of the functions and operation of each phase of the compiler.

Note: While referring to descriptions in this section, readers may find it useful to refer to the flowcharts in section 3, and to the fold-out figure "Creation and usage of data areas" in appendix D. Detailed descriptions and illustrations of the format or contents of the main data areas referred to in the descriptions are contained in section 5.

The compilation process performed by this compiler consists of a number of major operations, which are performed in sequence by the execution of some or all of the 52 phases that make up the compiler. In relation to these major operations, the phases can be collected logically into ten groups which, for descriptive purposes, are referred to as stages. Within each stage, most of the phases perform related functions which together comprise one of the major operations in the compilation process. In some cases, a phase is included in a particular stage for implementation purposes, i.e., its function is best performed at that stage of compilation but is not logically related to the major operation performed by other phases in the stage. The relationship between compiler stages and the major operations is shown in figure 2.1. The main functions of each stage, and the phases which are included in each stage, are listed in figure 2.2.

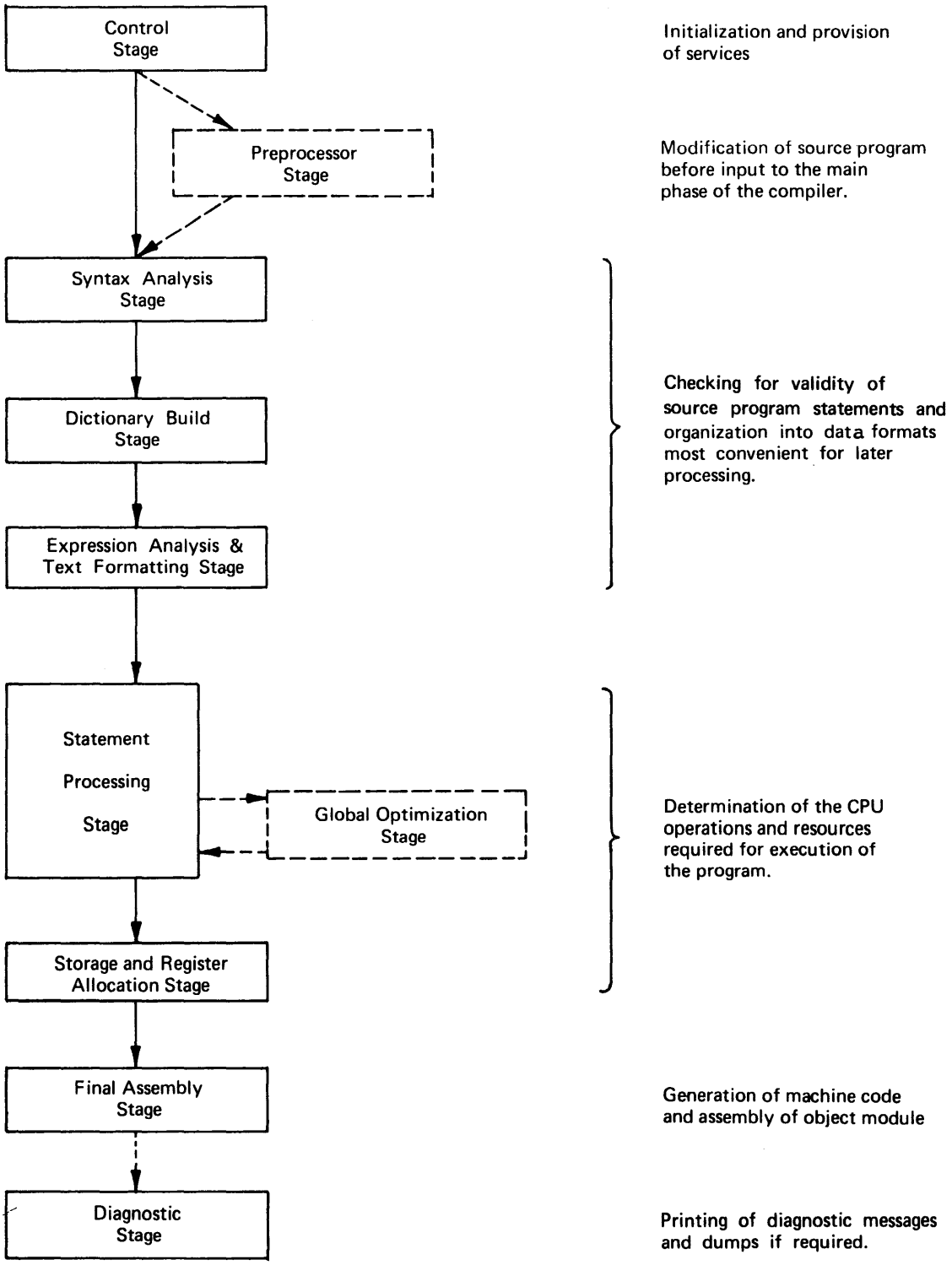


Figure 2.1. Relationship of compiler stages to major operations

Stage Name and Phases	Main Functions
Control Stage (Phases AA and AE)	Organizes storage and facilities required for compiler operation, including communication area and input/output buffers. Controls loading of other phases and provides services for them, e.g., satisfies requests for data pages.
Preprocessor Stage (Phases BA, CA and CE)	Optionally modifies source program by executing compile-time (%) statements and/or translating into processable code and character set.
Syntax Analysis Stage (Phases EA, EC, EE, and EI)	Eliminates comments and translates remaining text into compiler internal code. Identifies, classifies, and numbers statements, and checks syntax of statements and statement elements.
Dictionary Build Stage (Phases GA, GI, GE and GM)	Extracts information about each identifier and collects this information in dictionary tables for ease of reference.
Expression Analysis and Text Formatting Stage (Phases IA, ID, IE, and II)	Analyzes expressions and data aggregates, expanding them where necessary. Translates statements into a parenthesis-free format, consisting of fixed-length text tables.
Statement Processing Phase (Phases IK, KA, IM, IQ, KE, KI, KK, KL, KM, KQ, KT, KV, OC, OX, AND KX) (This stage is split into two parts if global optimization is executed.)	Interprets and analyzes the logical operations indicated by statements and statement elements, and generates modified or additional text tables to indicate the machine code required to perform those operations.
Global Optimization Stage (Phases OA, OE, OI, and OM)	Optionally analyzes the text stream and modifies its content and/or sequence to indicate code that is optimized to satisfy requirements specified in compiler options.
Storage and Register Allocation Stage (Phases PC, PA, PE, PI, QI, QA, and QE)	Modifies text and makes dictionary entries to indicate mapping and addressing requirements of different storage classes and data types. Allocates registers for use at execution time.
Final Assembly Stage (Phases SA, SQ, SD, SC, SK, SI, and SM)	Generates machine instructions indicated by text tables and according to addressing and register allocations. Assembles the object module and prints object listings if required.
Diagnostic Stage (Phases UA and AI)	Edits and prints diagnostic and compiler error messages. Prints dumps of compiler data areas if required.

Figure 2.2. Phases and functions of compiler stages

SPECIAL MACRO INSTRUCTIONS AND BOOKS

The design and construction of the compiler is based on the extensive use of specially designed macro instructions and books. Frequent references to such items are made in the published listings and in the descriptions in this manual.

Note: The term book is used to refer to an invariant sequence of code and/or data definitions that can be introduced into the assembly of a compiler module by use of a COPY statement.

At the time a compiler phase is assembled, an invocation of a macro instruction or book results in the generation of a predefined sequence of System/360 assembler language instructions. In cases where the function specified by a macro instruction requires a large number of assembler instructions, a call to a uniquely-labeled subroutine is generated rather than an extensive sequence of inline code. This feature is more fully described in section 3, "Program Organization."

All macro instructions designed especially for use in this compiler have a name beginning with the letter X; all routines and subroutines invoked by a macro instruction have a name beginning with the letter X, and similar to the name of the invoking macro instruction; all books have a name with the initial letter X, Y, or Z.

Approximately 160 different macros and books are used in the compiler to perform a wide variety of functions. A complete list of them, together with a brief description of their functions, is contained in appendix A. The published listings show the assembler code generated by each macro invocation.

REGISTER NAMING CONVENTION

Within the compiler code, all explicit references to general registers are made by use of symbolic names. The naming convention, which is also used in descriptions throughout this manual, is shown below:

Register Number	Symbolic Name
0	R0 or RO
1	R1 or RI
2	R2
3	R3
4	R4
5	R5
6	R6
7	R7
8	R8
9	R9
10	RA
11	RB
12	RC
13	RD
14	RE
15	RF

DATA REPRESENTATION

Data derived from statements in a PL/I source program is repeatedly processed during the sequential execution of the compiler phases, so that it is progressively transformed into code required in an object

module. The formats used for the internal representation of this data vary according to the type of processing being performed, i.e., the data is collected in basic formats that are most suited to the processing performed during one or more compiler stages. The general characteristics of the basic data formats used in the compiler are described in the following paragraphs; detailed descriptions and illustrations are contained in section 5, "Data Area Layouts."

FORMAT OF INPUT

Input to the compiler consists of a series of PL/I statements and comments grouped into one or more procedures. Each external procedure is read into the compiler input buffers as a series of 80-byte unblocked records (card-image format).

The compiler processes source statements written in the PL/I 60-character set and coded in EBCDIC. Options are provided which enable the compiler to accept input written in the PL/I 48-character set and/or coded in BCD. Use of input in these forms necessitates specification of the CHARSET(48) and/or CHARSET(BCD) option, which causes one of the preprocessor phases to be loaded to translate the records into the PL/I 60-character set and/or EBCDIC.

Modification of the source program at compile-time can be enabled by the inclusion of compile-time statements (identified by a preceding % character). Inclusion of such statements necessitates specification of the MACRO option or INCLUDE option, which cause one of the preprocessor phases to be loaded. If the MACRO option and either the CHARSET(48) or CHARSET(BCD) options are specified, the same preprocessor phase (Phase CA) performs all preprocessing.

Output from one of the preprocessor phases, consisting of preprocessed statements and comments, is passed on text pages in 84-byte record format to the main compiler read-in routines (in Phase EA). If the original source program was in the 48-character set and/or BCD, records in the original character set and code are also passed, interleaved with the translated records, in order that the SOURCE option can be satisfied.

The flowpaths of input records are illustrated in figure 2.3.

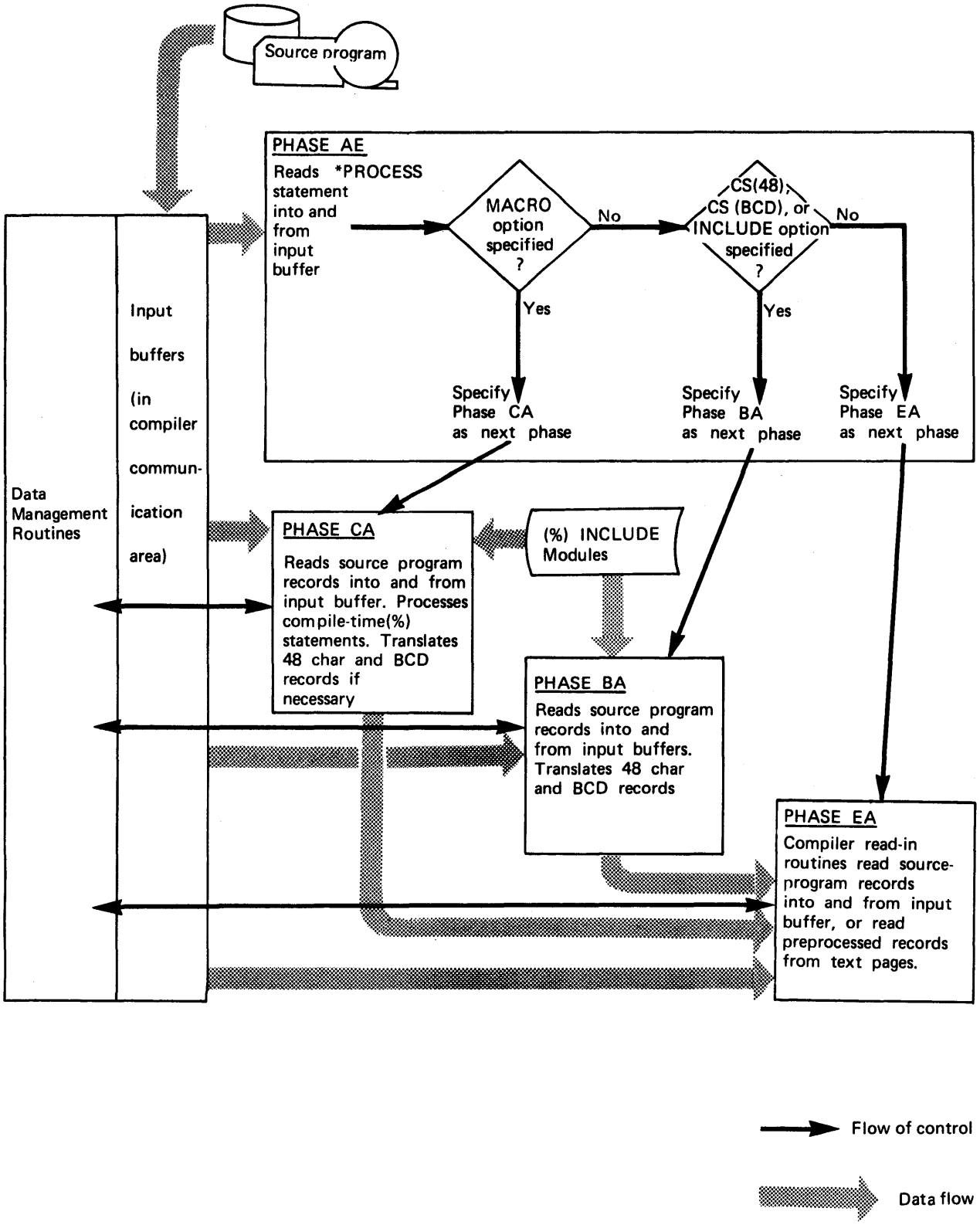


Figure 2.3. Flowpaths of input records

INTERNAL TEXT FORMATS

The main compiler read-in routines read the input records one at a time, either from the input buffers or from the pages output from the preprocessor stage, into an area of main storage acquired for initial processing. As records are read in, statements are identified and numbered, and comments and invalid characters are removed. Remaining characters are translated into an internal character code that is more convenient than EBCDIC for internal manipulation. This internal code is shown in figure 5.88. As translation of each source statement is performed, the input records containing the statement and comments in the original characters and code, together with other information such as statement number, etc., are copied to the print buffers if a source listing is required.

The internal representation of statements, initially corresponding to the source program, is referred to as text. The format of text is changed during compilation but at any time it can consist of a mixture of operators, operands, and program control elements. Operands can be constants, variables, files, etc., or compiler-generated items. Text is initially copied onto one or more pages to form a stream of data referred to as the main text stream. Additional streams of text extracted from the main text stream are temporarily created at various times during compilation, and are referred to as secondary text streams. Some secondary text streams are given names for ease of identification.

In the syntax analysis stage, statements are sorted so that all statements in a block appear before the first statement in a block at the next level of nesting. Within each block, statements are retained in source-program order, and within each statement, statement elements are also retained in source order. The text is organized in a sequential stream. This text format is referred to as Type-1 text. Although the internal representation of some statement elements is subsequently changed, and compiler-generated items are inserted in various places, the text retains the basic characteristics of Type-1 text format from its generation in the syntax analysis stage until it is processed by Phase II in the text formatting stage.

Routines in Phase II translate the text from a stream of statements into a series of fixed-length tables, each 32 bytes long. Each text table contains fields for an operator and three operands. All parentheses are removed, and data elements are arranged in appropriate fields in the text tables. Text in this format is referred to as Type-2 text. In addition to the operator and operand fields, Type-2 text tables contain fields that can be used for other purposes. For example, chain fields are used so that text tables can be inserted into, or deleted from, the logical sequence of text without requiring complete reorganization of its physical sequence.

The main text stream remains in Type-2 text format from Phase II until the text tables are replaced with machine code by the code-generation phases in the final assembly stage. Because more than one phase is involved in the generation of machine code, and because each of them only processes certain types of text tables, a time exists when the text stream contains machine code that has replaced text tables, and text tables that have yet to be replaced. In addition, special markers are inserted in the text to indicate processing required by later phases. This mixture of text formats, which exists from the start of code generation until the end of compilation, is referred to as extended code.

Where secondary text streams exist, the format of their contents do not always correspond to the format used in the main text stream at that time.

THE DICTIONARY

The attributes of an identifier directly affect the type of processing required when any reference to it is found in the text during compilation. It is therefore necessary for all relevant information to be available whenever an identifier is referred to. Instead of inserting extensive and numerous descriptions in the text streams, complete descriptions of all identifiers are collected in tables that can be accessed as required, and only the most frequently used information is inserted at references to identifiers in the text. The tables of descriptions are collectively referred to as the dictionary.

The dictionary is divided into four main sections: the names dictionary, variables dictionary, general dictionary, and storage dictionary. Collection of information for the dictionary starts in the syntax analysis stage, where certain types of statement are collected in a secondary text stream for ease of reference. The main parts of the names, variables, and general dictionaries are built in the dictionary build stage, from explicit, contextual, and implicit declarations. Additional entries are made in these dictionary sections throughout compilation. The storage dictionary is built during the storage allocation stage.

The names dictionary is used to hold the names of all the variable identifiers and some of the constants that appear in the text. Each entry (except entries for built-in function names) contains pointers to associated entries in other dictionary sections.

The variables dictionary is used to hold lists of the attributes of each variable in the program.

As its name implies, the general dictionary is used to hold a wide variety of information, such as:

- Details of the block structure of the program.
- The format and dimensions of data aggregates.
- Descriptions of constant values too great to be conveniently held in text.
- Standard default attributes to be applied to implicit declarations.
- Collections of information required for optimization.
- Descriptions of control blocks, etc., to be generated in the object module.

The storage dictionary is used to hold information about the amount and location of storage required for every identifier in the object module. It is built when most of the information about the object code to be generated has been determined, and can be considered as an extension of the variables dictionary.

When the main structure of the dictionary sections has been built, each reference to an identifier in the text is replaced by a brief description of its most important attributes and a reference to the dictionary entry containing the most complete description of it. To enable frequent access to dictionary entries without repeated use of lengthy addresses in the text, a directory is built and used to resolve dictionary references. This directory, which can be 560 or 1120 bytes long, is resident in the page area of the compiler partition throughout compilation.

PAGE-HANDLING SCHEME

The following paragraphs describe the system used in the compiler for internal data management, which is referred to as the page-handling scheme. This scheme is used by all phases of the compiler to acquire storage for newly created data, to access existing data, and to pass data to following phases. The facilities provided by the scheme are used to handle text and dictionary data, (previously described under the headings "Internal Text Formats" and "The Dictionary"), general reference data such as tables and lists which are required to be passed to other phases, and for the temporary storage and manipulation of data that is used by one phase only and which is discarded at the end of processing by that phase. Descriptions of the routines that implement and supervise the scheme are contained in the phase descriptions later in this section, in particular in the descriptions of Phases AA and AE.

The Page Area

An area of the compiler partition is allocated for the storage of data, such as the text, dictionary, etc. According to the relationship between the size of the partition and the content of the source program, it may not be possible for all data to be held in main storage throughout compilation. In such cases, data that is not currently being processed or accessed can be held in secondary storage. Space on direct-access storage devices assigned to SYS001 and SYS002 is allocated for a work data set in which this data can be stored. The transfer of data from main storage to secondary storage is known as spilling. The data sets are known as the spill data sets, and their DTF names are XSPILL1 and XSPILL2.

For ease of data handling, the area of main storage allocated for data storage is divided into eight or more equal divisions. The size of each record in the spill data set is related to the size of one division. Each data record is referred to as a page, the main storage area allocated for data storage is referred to as the page area, and each division of this area is referred to as a page space.

Page Size

The largest possible amount of the partition available to the compiler is allocated as the page area. The routine that allocates storage for the page area, and calculates the page space, is in the initialization phase (Phase AE). Design of the compiler is based upon a minimum of eight page spaces, but more can be used if sufficient main storage is available. The minimum page space size is 1100 bytes, of which 20 bytes (including the page header, which is described later) are used to hold data-handling information, and 1080 bytes are available for processable data. If storage is available, the page-space size is increased to either 1700, 3500, or 4096 bytes, of which 1680, 3480, or 4040 bytes can be used for processable data. Note that 4040 bytes are only used if the spill data sets are on either a 3330 or 3340 direct access storage device, and if the SIZE option is large enough.

At the beginning of each page space 16 bytes are used for a page header, the format of which is shown in figure 5.2. The page header is followed by space for processable data, and at the end of the page space is a 4-byte field, containing a page delimiter (X'99').

When a page is spilled, the last seven bytes of the page header, the processable data, and the four bytes containing the page delimiter, are spilled, Thus, the size of a record on the spill data set does not exactly coincide with the size of a page space.

To ensure that the compiler works efficiently in virtual storage, a page header table is used. This table contains copies of page header information. The copies of the page headers in the table differ only from the page headers themselves in that the pointer field points to the page itself rather than to another page header. Thus when a search is made for a required page, the page header table can be searched. This prevents the possibility of transfer of a page from virtual to real storage simply to inspect the page header.

Relationship between Main Storage and the Spill Data Set

If there is not enough workspace in main storage, one of the pages in main storage is copied onto the spill data set, and the space freed is used for another page.

To save space on the spill data set, information on whether a page has been spilled and the position of the copy on the spill data set is kept in a table called the in-core page directory. A list is also kept of pages that have been discarded by the compiler but still have copies on the spill data set. This list is called the discard table. Figure 2.3.1 shows the relationship between the fields involved.

The in-core page directory holds the relative track address of those pages that have been copied onto the spill data set, and zero for those that have not. Information about each page is held at the offset equal to the page number. (Page numbers are incremented in threes to allow for this.) The in-core page directory contains spaces for approximately five times the number of pages that are held in the page area. This allows all the pages that are likely to be needed during compilation to be addressed without the directory overflowing.

The discard table contains a list of those pages that have been discarded, but that still have copies on the spill data set.

Use of the tables allows discarded and outdated pages on the spill file to be overwritten. When a page is to be written onto the spill data set, the address at which it is to be placed is found by following the sequence below:

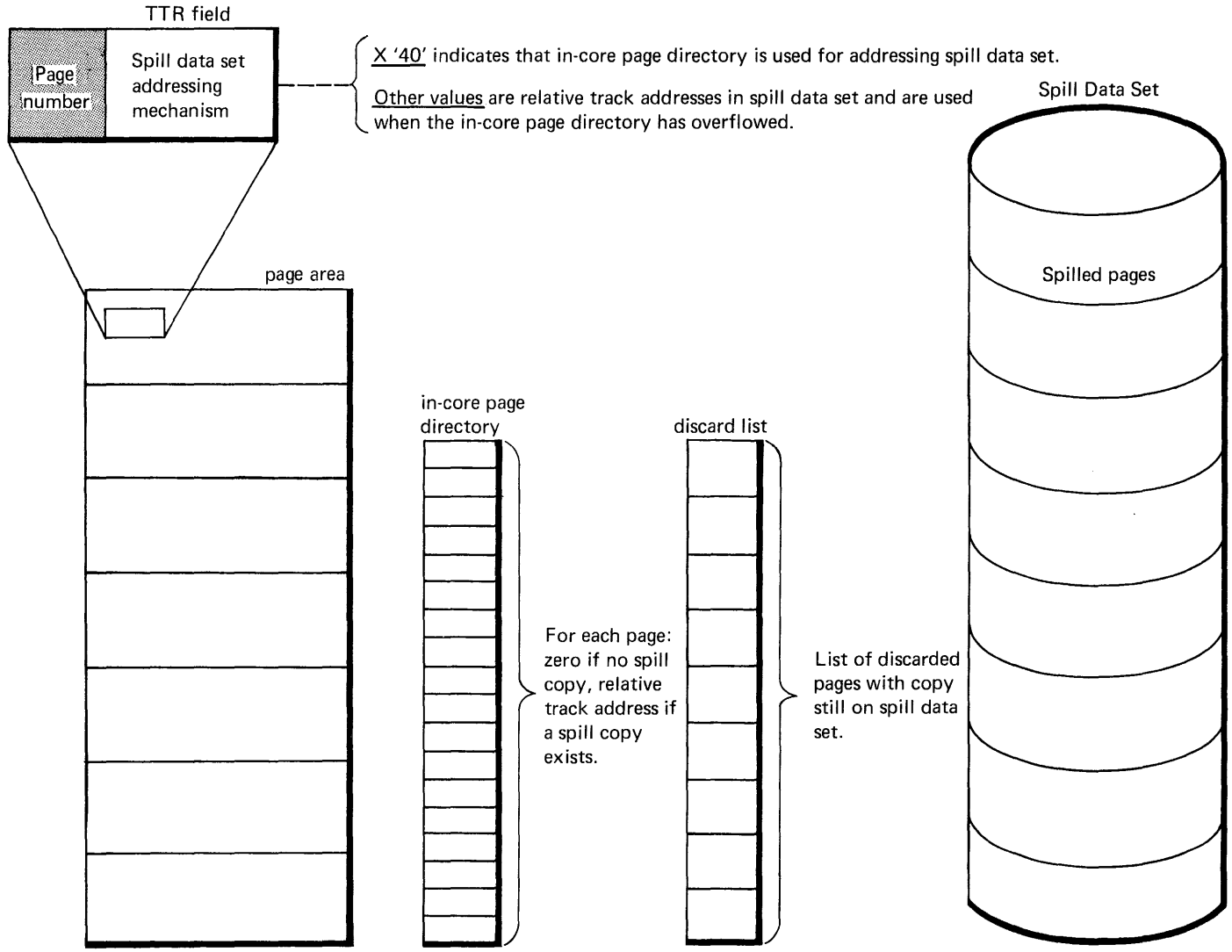
1. Looking in the discard list for a page that has been copied onto the spill data set and subsequently discarded. If one is found it is overwritten.

2. Looking for a read/write page that has a copy in main storage and a copy in the spill data set. In this situation, the copy on the spill data set will be outdated and can consequently be overwritten.

3. Extending the spill data set by writing a new record.

When the page has been copied onto the spill data set, its relative track address is entered in the in-core page directory. If this involves overwriting a page that has a copy in main storage (situation number 2 above), the in-core page directory entry for the page that is overwritten is set to zero.

At sizes of 80K and above, the in-core page directory holds about five times as many page spaces as there are pages in main storage. This means that five times the number of pages available can be handled by the method described above. If more pages are required than there are spaces in the in-core page directory, the relative track address of any additional pages is set in the TTR field in the page header. The TTR field contains the page number and either X'40' indicating that any spill page is to be addressed through the in-core directory, or, if the directory is full, the relative track address of any spill data set copy.



1. When a page is written on the spill data set, its track address is entered in the in-core page directory.
2. When a page is overwritten on the spill data set, its entry in the in-core page directory is set to zero. If it is a discarded page, it is removed from the discard table.
3. When more pages are used than there are spaces for in the in-core page directory, the relative track address into which such a page is to be spilled is placed in the TTR field.

Figure 2.3.1. The relationship between the page area and the spill data set

Page Status

A page in main storage is always given a specific status. The status, which can be one of six grades, indicates the relationship between the core copy and the spill copy, and the accessibility of the core page. The six status grades are:

Status	Indication
UNMOVABLE READ/WRITE	Core and spill copies are different; the core copy must not be spilled or overwritten.
UNMOVABLE READ-ONLY	Core and spill copies are the same; the core copy must not be spilled or overwritten.
SPILLABLE (MOVABLE READ/WRITE)	Core and spill copies are different; the core copy can be overwritten when the spill copy has been updated.
USABLE (MOVABLE READ-ONLY)	Core and spill copies are the same; the core copy can be overwritten immediately.
DISCARDED	Both core and spill copies are no longer required; both can be overwritten.
UNUSED	The page space has not yet been used; there is no associated spill copy.

Page Status Chains

To speed searches for required pages, all core pages of similar type and status are chained. There are six separate page chains, classified as follows:

TEXT UNMOVABLE
TEXT MOVABLE
DICTIONARY UNMOVABLE
DICTIONARY MOVABLE
UNUSED
DISCARDED

Note: For purposes of page handling, pages which contain general reference data that is not part of the dictionary are handled as text pages, even though the data they contain may not be part of a text stream.

Two-way chains are used so that, from any page in a chain, the preceding and following pages are immediately identifiable. The communications area contains a header field for each page chain. Each chain header field contains pointers to the first and last page in the chain. A page can be added to the beginning or end of a chain by referring to the chain header field. When a chain header is initialized (before there are any pages in the chain), the head and tail pointers both point to the head pointer. After initialization, all manipulation of page chains during compilation is performed by standard routines. These handle the general case, and the chains which contain only one or no pages.

BASIC PAGE-HANDLING OPERATIONS

Standard routines are used throughout the compiler for page-handling operations. The routine appropriate to the operation required is invoked by a macro instruction within the phase. Some macro routines call another macro routine in turn to perform the operation. For some page handling operations, in particular those operations that may involve input/output operations between main storage and the spill data set, the macro routines call routines in the resident control phase (Phase AA). The routines in Phase AA that are concerned with page handling operations are the page handling routine (AA4000), the spill supervisor routine (AA6000), and the phase loading routine (AA0300).

The three basic page handling operations that may be required by a processing phase are:

- Get a page space for a new page
- Get an existing page
- Change the status of a core page

The routines used to perform these operations vary according to whether an operation is connected with the handling of text (or general reference data) pages or dictionary pages.

Get a Space for a New Page: When a page space is required for the writing of a new page, the page chains are searched in the order: Discarded, Unused, Movable. The order in which Movable chains are searched is determined by a subroutine in the spill supervisor routine. In general, if the new page is to be a dictionary page, the text movable chain will be searched before the dictionary movable chain. The first suitable page found is known as the spill candidate. When a spill candidate is found, its status affects subsequent action as follows:

Status	Action
Discarded	The page space address and the TA are returned to the caller. The status is changed to Unmovable.
Unused	An identifying number (or core-TA), consisting of a two-byte number followed by X'40', is allocated, inserted in the page header, and returned to the caller together with the absolute address of the page. The status is changed to Unmovable.
Usable	A new TA is obtained from the data set and substituted for the existing TA. The page space address and the new TA are returned to the caller. The status is changed to Unmovable.
Spillable	The existing core page is written onto the data set. The action is then as for Usable.

Get an Existing Page: When an existing page is required, the Unmovable and Movable page chains of the appropriate type are searched in case the required page is already in main storage. If the page is found, its status is changed, the page is added to the appropriate chain, and the page address is returned to the caller.

If the page is not found in main storage, the page chains are searched for a spill candidate in the same order as for a new page search. The action taken when a spill candidate is found depends upon its status as follows:

Status	Action
Discarded	The TA of the discarded page is added to the discarded page table. (TAs in this table can be re-used for new pages.) The required page is then read in from the spill data set and overwrites the discarded page. The core-page address is returned to the caller.
Unused and Usable	The required page is read from the spill data set into the page space. The core-page address is returned to the caller.
Spillable	The spill candidate is written onto the spill data set at its existing track address. The required page is then read in from the spill data set to overwrite the new usable space. The core-page address is returned to the caller.

Change the Status of a Core Page: The status of an UNMOVABLE core page can be changed to MOVABLE or DISCARDED, and a MOVABLE core page can be changed to UNMOVABLE or DISCARDED.

A READ ONLY page can have its status changed to READ/WRITE. A READ/WRITE page cannot be changed to READ ONLY, as such a page may have been retained in main storage since its use by a previous phase, and the spill copy may not have been updated.

Selection of a Spill Candidate

The method used to select a spill candidate affects the number of input/output operations required during execution of a phase. For each phase there is an optimum method of selection, which is determined to a great extent by the use of dictionary pages.

A subroutine in the phase loading routine selects the optimum page handling routine for each particular phase before the phase is loaded. It can be called to change the selection method if processing requirements change during execution of a phase. Information about the selected method is passed to page handling routines via the communications area.

TEXT PAGE HANDLING

Requests for page-handling operations in connection with text or general-reference-data are made by use of macro statements which indicate the particular type of operation required, and contain information necessary to enable the operation to be performed. These statements generate inline macros which either perform the operation or call a macro subroutine. The subroutine may either perform the operation or call a control phase routine to perform those parts of the operation that are beyond its capabilities.

The majority of requests for basic page-handling operations are made by use of the XTXPG, XTXST, and XTXRF macros. The XTXRF macro calls the control phase routine AA4000 directly. The XTXST macro calls the XTXPGR macro subroutine, which performs the required operation. The XTXPG macro also calls the XTXPGR macro subroutine, but in this case XTXPGR calls the control phase routine AA4000 to perform some or all of the required operation. The relationship between these inline macro instructions, the macro subroutine, and the control phase routine is shown in figure 2.4. The functions of other macros involved in text handling are shown in appendix A.

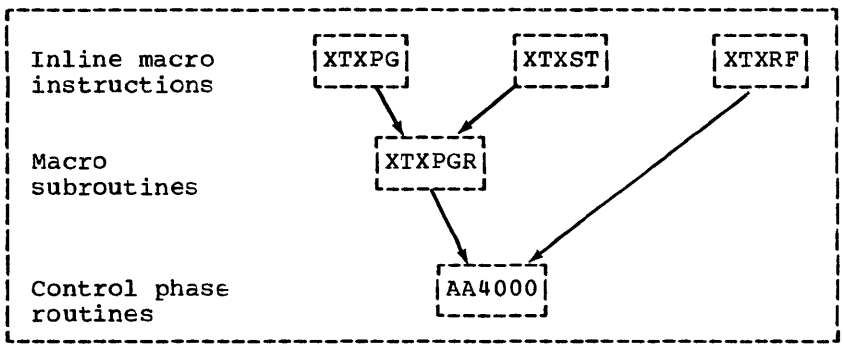


Figure 2.4. Routines and subroutines called in text page handling operations

The XTXPG Macro: Statements invoking the XTXPG macro can be used to specify three functions connected with page handling:

1. If a chain of text pages is being processed, XTXPG can be used to obtain the absolute address of the next sequential page in the chain, having the page read into main storage if necessary.

2. If a chain of text pages is being created, XTXPG can be used to get a new page and add it to the chain.

3. If an individual page is required for temporary storage of reference data, XTXPG can be used to get a new page without adding it to a page chain.

The particular function required is specified by using NEXT, ADD, or HEAD respectively as the first operand in the macro statement. Other operands can be used in the statement to specify:

- A revised status to be applied to the current page.

- A status to be applied to the next or new page.

- Registers or locations in which pointers to text references, or absolute addresses, in the current, next, or new pages can be found or are to be stored.

- Whether a read-ahead technique is to be used in any input/output operations connected with the request. If required, page input/output operations can be overlapped, and later checked for completion by use of the XCHECK macro. This technique enables time saving in cases where the need for a page can be predicted ahead of actual usage and where there is a page space available for the look-ahead page.

The XTXPG macro does not perform any of the functions, but sets bits in the XTXB0, XSSW and XTCLCD fields in XCOMM, and sometimes ensures that RB is pointing at a location in the current page, before calling the XTXPGR macro subroutine. Whenever it is called by XTXPG, XTXPGR calls the control phase routine AA4000 to perform some or all of the required functions.

The XTXST Macro: The XTXST macro is used to specify a change of status to be applied to a text page that is resident in main storage at the time the change is requested. The page for which the change is required can be identified in the invoking statement by the text reference, address of start of page, or the absolute address of any location in the page.

If a MOVABLE or UNMOVABLE page is to be changed from READ ONLY to READ/WRITE, the XTXST macro performs the function in line. For all other status changes, XTXST calls the XTXPGR macro subroutine, setting bits in the XTXB0 and XSSW fields in XCOMM to indicate the function

required. When called by XTXST, XTXPGR does not call any control phase routine.

The XTXPGR Subroutine: The XTXPGR subroutine is generated by the XROUT macro in response to an invocation of the XTXPG macro. XTXPGR receives control from either the XTXPG macro or the XTXST macro. These macros pass information about the required function in the XTXB0 field in XCOMM, and also in RB. XTXPG also passes information in the XCTLCD and XSSW fields in XCOMM, indicating functions required of the control phase routines in connection with next or new page requests.

Bit settings in XTXB0 give the following indications:

Bit No. 0123 4567	Function and Indication
	Information used to search for start of current page.
0--- ----	RB points at an absolute address in the current page.
1--- ----	RB points at a text reference in the current page.
	Function required of the XTXPGR subroutine.
--00 ----	Change status of current page only.
--01 ----	Get a new text page.
--10 ----	Get a new text page, and add it to the chain by inserting its TA in the OTXCN field of the current page.
--11 ----	Find the next page in the chain, as indicated in the OTXCN field of the current page.
	New Status to be applied to current page. (Characters in parentheses indicate value of operand in XTXPG or XTXST macro.)
-0-- 00--	UNMOVABLE READ/WRITE (UNRW)
-0-- 01--	UNMOVABLE (UNMV)
-0-- 10--	MOVABLE READ/WRITE (SPIL)
-0-- 11--	MOVABLE (SAVE)
-1-- 01--	DISCARDED (DISC)

Bit settings in XCTLCD give the following indications:

Bit No. 0123 4567	Function Required of Control Phase
---- ---0	Get next page in text page chain.
---- ---1	Get a new page.
---- --0-	Overlapped I/O not required.
---- --1-	Overlapped I/O required.
---- -0--	Make page READ/WRITE.
---- -1--	Make page READ ONLY.
---- 0---	Make page UNMOVABLE.
---- 1---	Make page MOVABLE.

Bit settings in XSSW give the following indications:

Bit No. 0123 4567	Function Required of Control Phase
0--- ----	Spill from oldest page.
1--- ----	Spill from newest page.
	If new page to be MOVABLE.
-0-- ----	Add to end of MOVABLE page chain.
-1-- ----	Test Bit 2.
--0- ----	Add to keep end of MOVABLE page chain.
--1- ----	Add to spill end of MOVABLE page chain.
	If old page to be MOVABLE.
---0 ----	Add to end of MOVABLE page chain.
---1 ----	Test Bit 4.
---- 0---	Add to keep end of MOVABLE page chain.
---- 1---	Add to spill end of MOVABLE page chain.
---- -1--	Page start address passed as parameter.

If RB has been set to point at a text reference or an absolute address, XTXPGR searches for the start of the containing page so that the page header can be accessed. The page must be in main storage. When the start of the page has been found, the page is removed from the status chain so that the page cannot be spilled during any input/output connected with the requested operation. Note that if bit 5 of the XSSW field is on, RB already points to the start of the page.

If the only function requested is a change in the status of a page, as when called by XTXST, this function is performed entirely by XTXPGR. The status field OSTAT in the page header is altered, and the page is added to the appropriate status chain by altering the OCNFD and OCNBK fields. If the page is to be made MOVABLE, XSSW bits 3 and 4 are examined to see whether the page is to be simply added to the most recent end of the MOVABLE chain or whether the required end has been specified. Control is then returned to the calling statement.

If the next page in a text-page chain is required, the TA of that page is found by examining the OTXCN field in the current page header. If the value of that field is zero, the current page is the last in the chain. This information is passed to the calling macro by setting RC to zero. If OTXCN contains a TA, RB is set to point at OTXCN and the control phase routine AA4000 is called. AA4000 finds the page, reads it into main storage if necessary, sets its status as required, and returns control with RC pointing at the start of the page. If a new page is to be added to a text page chain, AA4000 performs the functions indicated in XCTLCD and returns control to XTXPGR. Before returning control to the calling macro, XTXPGR inserts the TA of the new page in the OTXCN field of the current page header, and links the current page into the appropriate status chain.

If a new individual page is required, RB is not set. XTXPGR immediately calls AA4000 to acquire the page, and control is returned to the calling macro with RC pointing at the start of the new page.

The XTXRF Macro: The XTXRF macro is used to obtain the absolute address of an item identified by a text reference, i.e., the TA of a page and the offset of the item from the start of that page. XTXRF does not perform any of the function, but sets up information according to operands used in the invoking statement and then calls the control phase routine AA4000 directly. The text reference is passed in RB or in a defined area of storage. A register, or an area of storage, can be nominated for storing the absolute address of the item or the start of the page containing it. Other operands can be used to specify a status to be applied to the containing page, and whether overlapped input/output can be used in any spill operations. This information is passed to AA4000 in XCTLCD. AA4000 searches for the required page, reading it into main storage if necessary, applies the required status, and returns control with RC pointing at the start of the page.

DICTIONARY PAGE HANDLING

The basic difference between the handling of text data and the handling of dictionary data is the way in which the containing page is identified when a data item is referred to.

Each reference to an item in text is five bytes long, and consists of the TA of the containing page and the offset of the item from the start of the page. Thus the page is directly identified for page handling purposes.

Because of the large number of references to dictionary entries that are used, each reference to a dictionary entry is two bytes long, consisting of a unit number. Each section of the dictionary (names, variables, general, and storage) is built on a separate page or sequence of pages.

Within each section, the pages are numbered sequentially, starting at zero. (The section of the dictionary to which a page belongs, and the reference of its first entry, are shown in the ODCTP and ODCRF fields in the page header.) Entries within dictionary sections may be of fixed length or variable length, but within each section a fixed alignment length is used. Thus each dictionary section can be divided into units, and the start of each dictionary entry can be related to a unit. The number of units per page for each dictionary section varies according to the page size used in compilation. Dictionary references are two bytes long, and contain the unit number of the start of the required entry. To access the entry, the TA of the containing page, and the offset of the entry from the start of the page must be determined. To enable this, a directory is built at the start of the page area. The directory remains in main storage throughout compilation, and entries are made in it as new dictionary entries are created. Phase AE allocates 560 bytes of storage for the directory, or 1120 bytes if sufficient storage is available.

The directory is divided into four sections, each corresponding to a dictionary section. Each directory section contains a list of the track addresses of pages containing entries in the relevant dictionary section, the entries being made in section-page-number order. Thus the first three bytes of the directory section relating to the general dictionary contain the TA of general-dictionary-page number zero, the second three bytes contain the TA of general-dictionary-page number one, etc.

To identify the dictionary section to which a dictionary reference belongs, an identifying operand is used in the macro statement used to make a request for a dictionary page. The macro invoked by the statement sets up a corresponding value in the dictionary code byte XCODBT, in XCOMM. The values set in XCODBT are as follows:

XCODBT value	Indicated dictionary section
X'00'	General dictionary
X'08'	Names dictionary
X'10'	Variables dictionary
X'18'	Storage dictionary

XCODBT is used by the page-accessing routines and subroutines to index four 32-byte tables, XSQTBL, XMSKTB, XDRTBI, and XDICEN. (XDICEN is overlayed on XDRTBL.) Each of these tables, which are in XCOMM, consists of four 8-byte sections, (one section for each dictionary section). Each 8-byte section in the tables is organized as follows:

Table	Field name	Size in bytes	Field content
XSQTBL			
	XREF	2	Reference of next unused unit.
	XOFST	2	Offset within current page of next unused unit.
	XALGLN	1	Alignment length.
	XTA	3	Track address of current page.
XMSKTB			
	XOFMSK	4	Not used.
	XPGMSK	2	Not used.
	XELTH	2	Alignment length.
XDRTBL			
	XSHFT	2	Not used. (Number of dictionary units per page, when XDRTBL is overlayed with XDICEN.)
	XDROFS	2	The offset, from the start of the directory, of the start of a directory section corresponding to the relevant dictionary section.
	XSECSZ	2	The size of the section of the directory allocated to the relevant dictionary section.
	XCRDRF	2	The offset, from the start of a directory section, of the next free space.

Requests for dictionary page handling operations are made by use of macro statements, in which operands are used to specify the precise function required of the subroutines and the control phase routines that may be called. The functions of the various macros used for dictionary accessing are shown in appendix A. The macros most generally used and most directly involved with dictionary-page handling are XRFAB and XRFSEQ.

At compiler assembly time, any invocation of the XRFAB or XRFSEQ macros causes a few inline macro instructions to be generated. These instructions include a call to the XRFAB and XRFSEQ subroutines, which are respectively contained in the XRFABI and XRFSEQI macros. XRFABI and XRFSEQI are generated by the XROUT macro in response to the first inline invocation of the XRFAB and XRFSEQ macros respectively. The XRFAB and XRFSEQ subroutines may call control phase routines to perform part of a required function. The relationship between the inline macro instructions, subroutines, and control phase routines is shown in figure 2.5.

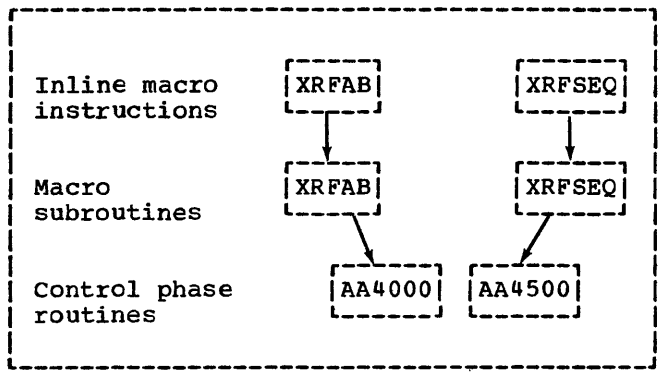


Figure 2.5. Routines and subroutines called in dictionary page handling operations

The XRFAB Subroutine: The XRFAB subroutine is called by the XRFAB macro to find the absolute address of a dictionary entry identified by a dictionary reference. If necessary it has the relevant page read into main storage.

The XRFAB subroutine uses the directory to determine the TA of the page that contains the referenced entry. If the reference is valid, it calls the control phase routine AA4000 to search for the page, passing any required information in the XCTLCD field of XCOMM. If necessary, AA4000 reads the page into main storage. The XRFAB routine then converts the unit number given in the reference into a byte-offset from the start address of the page, converts it into an absolute address, and returns it to the caller in RC or in some other specified register or location.

The XRFSEQ Subroutine: The XRFSEQ subroutine is called by the XRFSEQ macro, to find the absolute address of the next alignment unit available in a specified dictionary section for the creation of a new dictionary entry.

XRFSEQ examines the XOFST field in the relevant section of XSQTBL, to find the next available alignment unit in the current page identified in the XTA field. When the page and unit are identified, a check is made to see if there is sufficient space in the page to hold the new entry. All entries in the storage and variables dictionaries are of a known fixed length. For entries in the names and general dictionaries, the length of the entry is specified in the invoking statement. If the new entry can be created in the space available in the current page, routine AA4000 is called to find the address of that page, reading it into main storage if necessary, and XRESEQ applies the necessary offset to obtain the absolute address at which the new entry is to be made. If there is insufficient space available in the current page, routine AA4000 is called to get a new page. The directory, and the XDRTBL table in XCOMM, are updated before the address is returned to the caller.

Before the required address for a dictionary entry is returned to a caller, the XOFST and XREF fields of XSQTBL are updated in preparation for a subsequent request.

CONTROL STAGE

The control stage of the compiler consists of two phases, AA and AE. These phases perform functions that enable the compilation process to be carried out, but do not directly perform any processing of the text.

Phase AA is the resident control phase. It is the first compiler phase to be loaded and remains in main storage throughout compilation. It provides an interface between the compiler and the operating system, and performs various housekeeping operations required during the compilation process.

Phase AE is the initialization phase. It performs the once-only housekeeping operations required to prepare the operating environment for execution of the compiler processing phases. If the compiler is operating in batched-processing mode, this phase is loaded and executed before the processing of each external procedure.

Because some results of processing by Phase AE affect the way in which operation of the compiler is controlled, this phase is loaded and executed almost immediately after Phase AA has received control from the DOS supervisor program. Accordingly, for descriptive purposes, the operation of Phase AE is described here before the operation of phase AA is described.

INITIALIZATION PHASE (PHASE AE)

Phase AE performs the housekeeping functions required to prepare the operating environment for the processing of source modules. Some of these functions must be performed before the processing of each source module when the compiler is operating in batched-processing mode. Much of the information used by Phase AE is contained in Phase AA. The instructions that process this information are included in Phase AE to avoid the retention in main storage of instructions that are used once only. The functions of Phase AE include:

- Determination of the partition size available to the compiler.
- Initialization of the compiler communications area, and re-initialization of this area as required for batched compilations.
- Opening of the input, print, and spill data sets. The punch data set is opened if it is required by compiler options.
- Processing of compiler options.
- Calculation of the main storage area available for use as the page area, and calculation of the page size.
- Advising the operating system of the interrupt-handling routine to be used in the case of program check interrupts.
- Obtaining time and date for compiler headings.

PHASE INPUT

When Phase AA passes control to Phase AE, the communication area exists as a control section in which some of the fields are already initialized. The fields that are initialized include:

XACTL — the address of the start of Phase AA.

XBATCH — a flag byte set by Phase AA to indicate the type of processing required of Phase AE.

DTFs — for the input, print, punch and spill data sets.

When the input data set is open, Phase AE reads the *PROCESS statement (if present) at the front of the source program to determine the compiler options specified for the compilation.

The DOS supervisor communication region is accessed by Phase AE to ascertain the size of the partition available to the compiler.

PHASE OUTPUT

On completion of processing by Phase AE, all fields in the compiler communication area have initial values set as required for operation of the processing phases. All data sets required are open. Main storage is allocated only as shown in figure 1.3 and initialized.

PHASE OPERATION

Initialization of the Compiler Communication Area

When Phase AA passes control to Phase AE, Register 13 contains the address of the compiler communication area, XCOMM. Some of the fields in XCOMM have initial values set. Phase AE sets initial values in all other fields that are required at the start of the processing of a source program.

Routine AE0000 accesses XCOMM and examines a field, XBATCH. This field contains a flag byte set by Phase AA (or by the XREAD macro in the case of batched compilation) to indicate whether Phase AE is required to perform initialization functions after invocation of the compiler, or whether re-initialization is required before the compiler processes the second or a subsequent member of a batched compilation.

If XBATCH indicates that the initial member of a batch is to be compiled, Phase AE copies into XCOMM the addresses of control routines in Phase AA. It also issues a COMRG macro to access the DOS supervisor communication region, and copies information from there into XCOMM. The information copied includes the start and end addresses of the partition allocated for use by the compiler. These addresses are copied into the XACTL and XAEND fields.

Initialization of other fields is described in following paragraphs. Fields in XCOMM that do not have an initial value set by Phase AA or AE, and may be read by a processing phase before being set, have their initial value set to zero.

Opening and Initialization of Data Sets

The input, print, spill, and punch are opened by Phase AE. The DTFs for these data sets are in XCOMM when it is loaded, and are partly initialized. Completion of the initialization, required before a data set can be used, includes the insertion of the address of the LIOCS module into the DTF.

The spill data sets are required to be open for every compilation. If batched compilation is performed, the data set is opened and initialized for processing the first member of the batch, and remains open until all compilations in the batch are completed. The requirement for opening and initialization is indicated by the setting of XBATCH, which is tested by Phase AE at the start of each compilation. The LIOCS module for the spill data sets is in Phase AA. When the page size has been calculated, the blocksize field in the DTF is completed, an OPEN statement is issued to open the spill data sets, and the address of the LIOCS module is inserted in the DTF. If the spill data sets have been opened for a previous compilation, the track address of the first record on the data set is inserted in the XNWTA field in XCOMM, to indicate to the page-handling routine in Phase AA the TA on the first page to be used.

The LIOCS modules for all other data sets are generated within the phases that use the data sets. A common LIOCS module is used for the input, print, punch, and load data sets. This LIOCS module is generated by use of an XPRINTR macro instruction within the phase. A different entry point in the LIOCS module is used for each data set, and the address of the relevant entry point is inserted in the appropriate DTF by use of an XREADI, XPRINTI, XLOADI, or XPUNCHI macro instruction.

The input and print data sets are used by Phase AE, and are therefore opened and initialized by this phase for all compilations. Use of the

punch and load data sets depends upon the compiler options that are specified. Phase AE examines the options and opens the punch data set if it is required. The LIOCS module is generated, and the DTF is initialized, by Phases CA or SI. The load data set is opened by Phase SI if required. Phase SI contains the DTF and also the buffer area for that data set, and generates the LIOCS module as required.

Processing the Compiler Options

The standard default specifications for all compiler options are defined in the PLIOAS module, which is link-edited and stored in the core-image library at system-generation time. In addition to indicating the default options, PLIOAS also indicates those options that are deleted from usage and which cannot be altered by options specified in the *PROCESS statement. If any compiler option deleted at system installation time is required for a particular compilation, it can be enabled by the use of the CONTROL option in the *PROCESS statement. This option must be specified with a password that is defined at system installation time. Use of an incorrect password will cause compilation to terminate.

The default and delete indications are given in separate 16-byte fields within the PLIOAS module. The following table describes the relative positions of the default bits and delete bits, and gives the standard defaults with appropriate default bit settings. The same byte offset and mask apply to the default and delete tables for the same option.

Bit	Address Byte	Address Mask	Option	Standard Default	Default Bit Setting
1	+0	80	ATTRIBUTES	NOATTRIBUTES	0
2		40	AGGREGATES	NOAGGREGATES	0
3		20	DYNBUF	NODYNBUF	0
			CHARSET		
4		10	EBCDIC|BCD	EBCDIC	1
5		08	60|48	60	1
6		04	CATALOG*	-	0
7		02	LIST	NOLIST	0
8		01	COMPILE	NOCOMPILE(S)	0
9	+1	80	NC(W)		0
10		40	NC(E)		0
11		20	NC(S)		1
12		10	DECK	NODECK	0
13		08			
14		04	DUMP	NODUMP	0
15		02			
16		01	ESD	NOESD	0
			FLAG	FLAG(I)	
17	+2	80	(I)		1
18		40	(W)		0
19		20	(E)		0
20		10	(S)		0
21		08	LIMSCONV	NOLIMSCONV	0
22		04			
23		02			
24		01	INSOURCE	INSOURCE	1
25	+3	80	LINECOUNT*	LINECOUNT(55)	0
26		40			
27		20	MACRO	NOMACRO	0
28		10	MARGINI	NOMARGINI	0
29		08	MARGINS*	MARGINS(2,72)	0
30		04	MDECK	NOMDECK	0
31		02	NAME*	-	0
32		01	NEST	NONEST	0
33	+4	80			

34		40	MAP	NOMAP	0	
35		20				
36		10	OFFSET	NOOFFSET	0	
37		08	OPTIMIZE	NOOPTIMIZE	0	
38		04	TIME			
39		02	TIME			
40		01	TIME		0	
41	+5	80	OPTICNS	OPTIONS	1	
42		40	STORAGE	NOSTORAGE	0	
43		20				
44		10				
45		08				
46		04				
47		02				
48		01	SIZE*	SIZE(MAX)	0	
49	+6	80	SOURCE	SOURCE	1	
50		40				
51		20	GOSTMT	NOGOSTMT	0	
52		10				
53		08	SYNTAX	NOSYNTAX(S)	0	
54		04	NOSYN(W)		0	
55		02	NOSYN(E)		0	
56		01	NOSYN(S)		1	
57	+7	80				
58		40	FLOW	NOFLOW	0	
59		20	XREF	NOXREF	0	
60		10				
61		08				
62		04	LINK	NOLINK(S)	0	
63		02	NOLINK(W)		0	
64		01	NOLINK(E)		0	
65	+8	80	NOLINK(S)		1	
66		40				
67		20				
68		10				
69		08				
70		04				
71		02				
72		01				
73	+9	80				
74		40				
75		20				
76		10				
77		08				
78		04				
79		02				
80		01				
81	+10	80				
82		40				
83		20				
84		10				
85		08	COUNT	NOCOUNT	0	
86		04				
87		02				
88		01	INCLUDE	NOINCLUDE	0	
89						
90						
91			XREF(SHORT)		0	
92						
93						

94			TSTAMP		0
95					
96					

NOTE: For FLAG, COMPILE, and SYNTAX options, only the first of the corresponding DELETE bits is used to indicate that the option is non-deleteable.

* Applies to the delete table only.

Each time Phase AE is executed, the PROCOPS routine issues a LOAD macro instruction to have the PLIOAS module loaded into an area of the phase working storage. For each compiler option specified in PLIOAS, an appropriate bit is set in the XNSYGBT field in XCOMM. Similarly, for each compiler option that is disabled, a bit is set in the XNDELET field in XCOMM to indicate that specification of that option in the *PROCESS

statement is invalid and that a diagnostic statement should be generated. For those compiler options for which a specific value can be specified (e.g., the SIZE and MARGINS options), the values specified in PLIOAS are copied into appropriate fields in XCOMM.

If XBATCH indicates that the first source program is to be processed, an XREAD macro instruction is issued to read the first record from the input stream into an input buffer. A test is made to check that this is a *PROCESS statement. If XBATCH indicates that the second or a subsequent member of a batched compilation is to be processed, a *PROCESS statement will already be held in the input buffer; a *PROCESS statement is treated as the end of file marker by the XREAD macro routine in Phases BA, CA, or EA, and is retained in an input buffer while the previous batch member is being processed.

Three subroutines, BLKSKP, CBSKP, and PNCSCN, are called to handle internal delimiters, such as blanks, commas, brackets, quotes, and equals signs, when scanning the record for an option keyword. When an alphameric character string is detected, its length to the next delimiter is determined. The KEYSCN routine then searches the options keyword table at KEYTBS for keyword entries of the same length as the detected character string.

Within the KEYTBS tables, valid keywords are grouped according to their length. Each entry in the tables consists of:

1. A DC instruction defining an option keyword.

2. A TM instruction which is used to determine whether the option has been deleted from usage.

3. An instruction which sets a bit in a field in XCOMM to indicate that the option is specified or, if the option has a value list, passes control to the appropriate processing routine.

When a valid option keyword corresponding to the detected character string is found, the instructions at items 2 and 3 above are executed. If the character string detected is not a valid option keyword, or if there is an error in its option value list, the STRGAJ subroutine is called to generate an error message and locate the next keyword.

Records are read in and processed until a semicolon indicates the end of the *PROCESS statement. The next record is then read in and tested to see if it is a further *PROCESS statement. If it is, the options are processed on an additive basis. If an option is found that has been specified previously, the later specification is used. If the later specification is invalid, the default specification is used rather than reverting to the previous specification. When a record that is not part of a *PROCESS statement is found, it is assumed to be the first record of the source program, and is copied into the XREC1 field of XCOMM for use when the SOURCE option is implemented. On completion of processing by the PROCOPS routine, information about all option settings applicable to the compilation are available in XCOMM to any compiler phase. The enablement of specified options can be tested by use of the XCOPT or XTOPT macro, and individual fields can be accessed to test for specified values.

Calculation of Page Area

The routine AE5000 calculates and identifies the space available for pages. The first operation performed by this routine is a comparison of the partition size allocated for use by the compiler (XAEND minus XACTL) with the partition size specified in the SIZE option (stored in XSIZE). If the specified size is the greater, an error message is generated and the maximum partition size is allocated. If the specified size is less

than or equal to the maximum partition size, the size of the page area is calculated using XSIZE.

The amount of storage required for the resident control phase, the compiler communication area, and the phase area, (which is the total non-page area of the compiler partition), is stored as a constant value in ZOVHD is Phase AE. The size and address of the page area is found by subtracting ZOVHD from the compiler partition size.

Compiler design requires a minimum page area of 9544 bytes, i.e., space for eight pages of 1080 bytes each, plus space for headers and tables to look after the pages. If the page area is less than 9544 bytes, an error message is generated and compilation is terminated. To reduce spill file input/output operations, three different page sizes are used. These are: 1680 bytes if the page area is greater than 13600 bytes, 3480 bytes if the page area is greater than 28000 bytes, 4040 bytes if the page area is greater than 32800 bytes, and the spill data sets are on either a 3330 or 3340 direct access storage device.

The page size determined is stored in XPAGS in XCOMM and the number of page spaces is stored in XPNO.

When the page size is known, the record length for the spill data set is calculated. The record length is the usable page space (1080, 1680, 3480, or 4040 bytes) plus 11 bytes for some of the page header information and the page delimiter. The XSPILL DTF is completed and the spill data sets opened.

The first usable track address on the spill file data set is stored in XNWTA, and used by Phase AA when formatting the data set. The UNUSED page-status chain is set up, the page-header tables are initialized, and the dictionary tables XMSKTB, XSQTBL, and XDRTBL (in XCOMM) are initialized.

If XBATCH indicates that batched processing is in operation, all the information about the page area, page size, etc., will already be available and the spill file will be open. Phase AE checks the highest track address used in any of the previous compilations and stores it in XSAVTA. For the new compilation, no new page will have to be acquired until all the track addresses up to XSAVTA are used. XNWTA points at the next track address on the data set to be used when a new page is required. All pages in main storage are given the UNUSED status, and other page-status chains are set to null.

Identification of Interrupt-handling Routine

The interrupt handling routine is at AA0600 is Phase AA. To reduce the storage space required by the resident control phase, routine AE6000 identifies the interrupt handling routine and the dump save area, and passes their addresses as arguments when issuing a STXIT macro.

Compiler Headings

Routine AE0100 issues a GETIME macro to obtain the time of day from the computer timer feature. For each compilation, this time is printed out in the form HH.MM.SS at the head of any listing.

THE RESIDENT CONTROL PHASE (PHASE AA)

Phase AA consists of a number of service routines. These routines are used by the processing phases of the compiler to provide standard services, and to provide interfaces with the DOS supervisor program. The phase is loaded by the DOS supervisor, and remains in main storage until compilation is completed. Its functions include:

- Operations required at the start of each compilation.
- Control of the loading of all other compiler phases.
- Satisfaction of all requests from processing phases for new or existing data pages.
- Handling of program check interrupts.
- Operations required at the end of each compilation.

These functions are performed by the following routines:

AA0000	Compilation Start Routine
AA0300	Phase Loading Routine
AA4000	Page Handling Routine
AA6000	Spill Supervising Routine
AA0600	Program-interrupt Handling Routine
AA0500	Compilation End Routine

PHASE OPERATION

Note: Some of the routines described in the following paragraphs perform functions connected with page-handling operations. A general description of the page-handling scheme used by the compiler is given earlier in this section. The control phase routines involved in page handling" operations are shown in figure 2.6.

Compilation Start Routine (AA0000)

The DOS supervisor has Phase AA loaded from the core-image library and passes control to AA0000. A CSECT named XCOMM is loaded with Phase AA to provide an area of main storage that can be used for communication between compiler phases. Phase AA identifies XCOMM by setting Register 13 to point at it. The address of XCOMM is retained in Register 13 throughout compilation.

When XCOMM is loaded, some of its fields contain their requisite initial values. Phase AA stores its own start address in the field named XACTL. It also stores the time of the start of compilation in the XTIMU field. If the compiler is operating in batched processing mode, XTIMU is set for each source module.

Some of the fields in XCOMM require initialization only on invocation of the compiler. Other fields need to be re-initialized before compilation of each batch member. Phase AA sets the XBATCH field in XCOMM to zero to indicate that initialization is required for compilation of the first source module. (For batched compilation, XBATCH is subsequently set by the phase that reads the input records, Phase BA, CA, or EA.) Phase AE is then loaded and control passed to it.

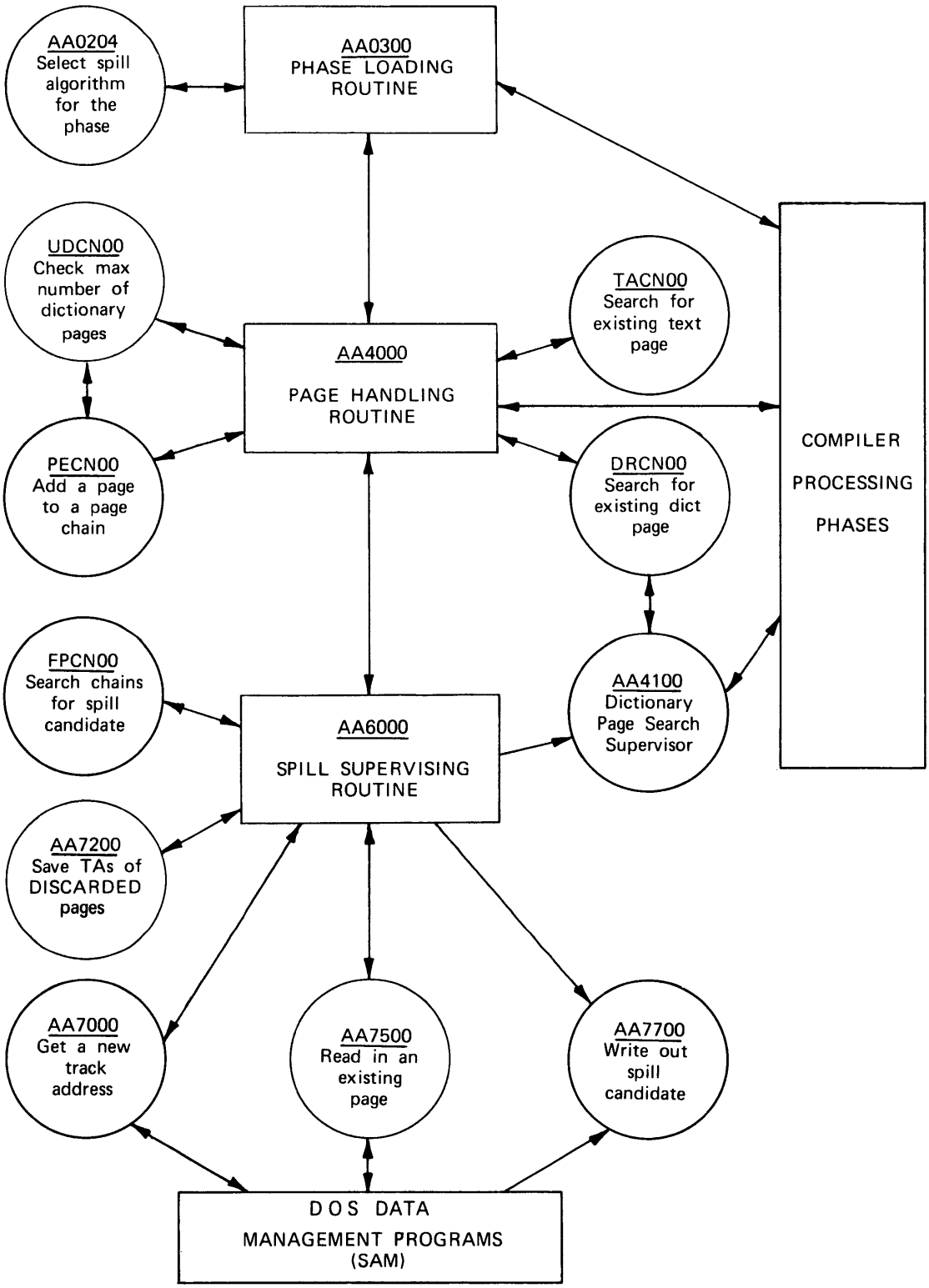


Figure 2.6. Control-phase routines and subroutines used in page-handling operations

Phase Loading Routine (AA0300)

The prime function of this routine is to issue instructions that cause a specified phase (or phase segment) to be loaded. The routine also performs some page-handling organization to ensure that data processed by a phase is made available in the most efficent manner.

When a phase has completed its processing, it uses an XPST macro instruction to call the phase loading routine, and to specify the next processing phase to be loaded. Before the specified phase is loaded, the phase list at XPHSL (in XCOMM) is examined to see if an interphase dump has been specified in the compiler options. If so, Phase AI is executed before any preparation for the next processing phase is performed.

In addition to specifying the name of the next phase, the XTEMP table generated by the XPST macro instruction also contains code bytes that indicate the paging requirements of the next phase. The code bytes are used to select the optimum page-spilling algorithm for the phase. The information in the code bytes is also used by the main routine, which has pages added to appropriate page chains and re-initializes fields in XCOMM that contain page-handling information. In order to perform these functions, the page handling routine (AA4000) and the spill supervising routine (AA6000) are called. When the required page reorganization is complete, the new phase is loaded.

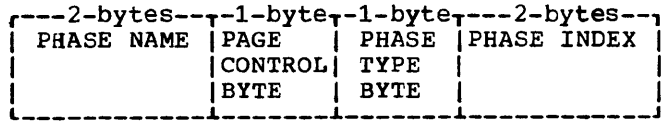
In addition to ensuring that data is available for any phase when it is loaded, this routine may also be called to perform similar functions during the execution of a phase, if the page-handling requirements alter because of a change in the nature of processing by that phase (e.g., if dictionary pages are required during part of its processing, but not required later). In such cases, the routine is called by use of an XFREE macro instruction, and control is returned to the phase when the page reorganization is complete.

Selection of Page Spilling Algorithm: For each processing phase there is an optimum method of selecting a spill candidate when a page space is required. The method of selection may be affected by:

- The scanning methods used -- sequential or non-sequential.
- The ratio of the number of page spaces to the number of pages in use.
- The type of processing performed, e.g., a phase that scans text only will require spill candidates to be selected on a different basis from a phase that accesses both text and dictionary.

Selection of the optimum page-spilling algorithm is performed in the phase-building routine, AA0300.

At the end of each processing phase, an XPST macro generates a 6-byte table, XTEMP, which contains the name and details of the next phase to be loaded. (The details for each phase are predetermined.) The format of the table is as follows:



Page Control byte:

Bit Number		Indication
0	not set	Spill text before dictionary.
	set	Spill dictionary before text. Bit 0 is used when looking for a spill candidate to replace with a text page.
1	not set	Phase simply adds to text stream.
	set	Phase creates new text stream. Bit 1 is used to determine relative sizes of text stream and page space.
2 - 4		Factor 1.
5 - 7		Factor 2.

Phase Type byte:

Bit Number		Indication
0	not set	Separate logical phase.
	set	Another branch in same overlay structure.
1 - 4		Spare.
5	not set	Dictionary not required.
	set	Dictionary required.
6 - 7		Number of dictionary pages to be maintained in unmovable dictionary chain.

The page-spill algorithm routine determines its algorithm from:

1. The control byte,

2. The size of the text stream, XTXC, and

3. The number of pages in main storage, XPNO.

The first bit in the control byte determines whether text-spill candidates are to be taken from the text (bit 0 = 0) or from the dictionary (bit 0 = 1); the former tends to increase the number of dictionary pages as opposed to text, the latter reduces it.

The second bit determines how the size of the text stream after this phase is to be determined. Either the phase creates a new text stream (bit 1 = 1) in which case the size of the stream is just the number of new text page requests (XTXC = XNWTP) or the phase just adds to the existing stream (bit 1 = 0) in which case the size of the text stream is incremented by the new text pages (XTXC = XTXC + XNWTP).

The last six bits contain two factors, F1 and F2, which enable the routine to calculate three different ranges for the ratio XTXC/XPNO. In general when this ratio is very large or very small, the spill candidate is taken as the oldest movable page, at intermediate values the newest page is chosen. However, the ranges are variable: some phases always spill the oldest (F1 = 0) whilst others always spill the newest (F1 = 7).

Having determined the algorithm, XSSW bit 0 is set accordingly: bit 0 = 1 if the oldest is to be spilled, but 0 = 0 if the newest is to be spilled.

The processing phase is then loaded and the page handling routines in Phase AA use the masks and switch as required during page handling operations.

Page-handling Routine (AA4000)

This routine is called to handle all requests for pages required during processing. The routine may be called by macro routines in the processing phases, or may be called by the phase loading routine. In dealing with page requests, this routine calls the spill supervising routine to control input/output operations required if a request is made for a page not in main storage, or for a new page.

The routine consists of a main routine that supervises the search for a requested page, and a number of subroutines that perform functions connected with the search. Some of these subroutines may be called directly from the processing phase routines.

The main routine is called to handle all requests for text and dictionary pages, except when an existing dictionary page is identified by a dictionary reference (in such cases the routine is entered at AA4100). If a request is made for an existing page, the routine decides which chains to search, and builds a list indicating the order in which the chains are to be searched. Control is then passed to either TACN00 or DRCN00 to search these chains. TACN00 searches for a text page that is identified by its track address. DRCN00 is used to search for a dictionary page that is identified by the dictionary reference of the first entry in the page. If the required page is found, the subroutine PECN00 is called, and, according to the setting of the first three bits in XSSW, may choose which end of the chain to add a movable page. The address of the page is returned to the processing phase.

For each phase there is a maximum number of dictionary pages that can be maintained with the unmovable status. The number is specified when the XPST macro statement is used to specify the phase. Before any page is added to the dictionary unmovable chain, the UDCN00 subroutine is called to ensure that the limit is not exceeded. If necessary, the oldest page, (i.e., the page at the start of the chain), is removed from the unmovable chain, and added to the movable chain by the PECN00 subroutine, which then adds the found page to the unmovable chain. Dictionary pages are always added to the newest end of the movable page chain.

Spill Supervising Routine (AA6000)

This routine is called to satisfy a request for a new page, or for an existing page that is not in main storage. The routine has no direct interface with the compiler phases. It can be called from a number of places in the page handling routine whenever input/output operations between main storage and the spill data set may be required. The routine contains a number of subroutines, some of which interface with the DOS BSAM data management programs by use of system macro instructions.

The subroutine AA7200 is used to maintain a table, in the communication area, of the track addresses of DISCARDED pages. Re-use of these track addresses reduces the rate of expansion of the spill data set. The list is updated each time a track address is re-used.

The subroutine AA7000 is called whenever a new page is requested. If there is a track address available in the list maintained by AA7200, that track address is allocated to the new page. If there is no discarded track address available, the routine writes a new formatting record after the last record on the spill data set. The track address of the new record is obtained, and becomes the name of the new page. The system macro instruction instructions used in this subroutine are WRITE, CHECK, and NOTE.

Subroutine AA7500 is called to read a page from the spill data set into a page space in main storage. Subroutine AA7700 is called to write a page from main storage to the spill data set. When a page is written, the target track address is the name of the page. When a page is read, the source track address is the name of the page requested. These subroutines check for completion of input/output operations if overlapped input/output is specified in the page request. The system macro instructions used in the subroutines are CHECK and POINTR.

When the main routine is called to satisfy a page request, the subroutine FPCN00 is called. This subroutine searches page chains in an order specified by AA4000. The first page found in this search is the spill candidate. The action then taken depends upon the nature of the page request, and the status of the spill candidate.

If the request is for a new page, the action taken is as follows. If the status of the spill candidate is UNUSED or DISCARDED, it can be used immediately for the new page. If the spill candidate is USABLE, a new track address is required, and subroutine AA7000 is called to provide the new track address. If the spill candidate is SPILLABLE, subroutine AA7700 is called to write the spill candidate to the spill data set, and AA7000 is then called to provide a track address for the new page.

If the routine receives a request to read an existing page into main storage, the action is as follows. If the status of the spill candidate is DISCARDED, subroutine AA7200 saves its track address for future use,and subroutine AA7500 reads the requested page into the available page space. If the spill candidate is SPILLABLE, subroutine AA7700 is called to write it to the spill data set. Subroutine AA7500 is then called to read the requested page into the available page space.

Interrupt-handling Routine (AA0600)

If a program check interrupt occurs during compilation, the STXIT macro issued by Phase AE at initialization time causes the operating system to pass control to routine AA0600.

This routine contains instructions which cause a branch-and-link to the address held in the XDMADR field of XCOMM. If an "abort" dump has been specified, the address in XDMADR is the entry point of the dump phase (Phase AI). Phase AI is executed to print the required dump, and control then returns to the interrupt-handling routine. If the DUMP option has not been specified, Phase AI will not have been loaded, and XDMADR contains an address which causes control to be returned immediately to the interrupt-handling routine. An XDIAG macro instruction is then executed to test which of the message-editing phases is to be loaded. Phase CE prints the compiler-error message and any diagnostic messages generated by Phase CA; Phase UA prints the compiler-error message generated by any other phase, plus any diagnostic messages generated prior to the interrupt. When control returns to the interrupt-handling routine, it passes control to the compilation end routine.

Compilation End Routine (AA0500)

This routine is entered on completion or termination of the compilation of a source module.

The linkage to the system interrupt handling routine is cancelled. If the XBATCH switch indicates that there are more source modules to compile, control is passed to the start routine. If all compilations are complete, all data sets are closed and control is returned to the operating system.

THE PREPROCESSOR STAGE

The preprocessor stage consists of three phases: Phases BA, CA, and CE. The functions of Phase BA and CA are to modify the source program so that it is passed to the syntax analysis stage of the compiler in a format acceptable as compiler input. When necessary, the content of the PL/I source program is modified by translation into 60-character set EBCDIC format and by execution of any compile-time statements (identified by a preceding % character) that it contains. Phase CE edits and prints any diagnostic or compiler-error messages generated during preprocessing. The flowpaths of input records are shown in figure 2.3.

Phases in the preprocessor stage are loaded and executed only if the relevant compiler options are specified. These options are:

MACRO	Source program may include compile-time statements.
CHARSET(48)	Source program written in 48-character set and coded in EBCDIC.
CHARSET(48,BCD)	Source program written in 48-character set and coded in BCD.
CHARSET(60,BCD)	Source program written in 60-character set and coded in BCD.
INCLUDE	Source program may contain %INCLUDE statements.

If the MACRO option is specified, Phase CA is loaded and executed. Phase CA performs all preprocessing of the source program, including translation into 60-character set EBCDIC if CHARSET(48), CHARSET(48,BCD), or CHARSET(60,BCD) is also specified.

If the NOMACRO option is specified (by default) and one of the options CHARSET(48), CHARSET(48,BCD), CHARSET(60,BCD) and INCLUDE is specified, Phase BA is loaded and executed. This phase translates the source program into 60-character set EBCDIC and performs any %INCLUDE statements in the program.

If any diagnostic or compiler-error messages are generated during the execution of Phase BA or CA, Phase CE is called to edit and print them.

If the options NOMACRO (by default), NOINCLUDE and CHARSET(60) are specified, no preprocessing is required, and the phases of the preprocessor stage are not loaded.

The diagnostic messages produced by the diagnostic message editing phase, Phase CE, are graded in order of severity. Implementation of the FLAG option permits specification of messages of a chosen minimum level of severity to be listed. The choice is indicated by means of the FLAG option value list as follows:

FLAG(I)	All diagnostic messages listed.
FLAG(W)	All diagnostic messages except 'informatory' messages listed.
FLAG(E)	All diagnostic messages except 'warning' and 'informatory' messages listed.
FLAG(S)	Only 'severe' errors and 'unrecoverable' errors listed.

Note: The specification FLAG is equivalent to FLAG(I).

Full details of the diagnostic message severity levels and of all preprocessor options are given in the Programmer's Guide for this compiler.

48-CHARACTER/BCD/INCLUDE PREPROCESSOR (PHASE BA)

The function of this phase is to convert source statements written in the PL/I 48-character set to the PL/I 60-character set, and to process %INCLUDE statements if the INCLUDE compiler option applies. In addition, if specification of the CHARSET(BCD) option indicates that the source program is represented by BCD, the program will be translated to the EBCDIC representation. This phase is loaded only if the NOMACRO option applies.

PHASE INPUT

The input to the phase consists of records, with a maximum length of 80-bytes, in card-image form. The records can contain PL/I statements (and comments) written in either the PL/I 48-character set or the PL/I 60-character set and coded in BCD or EBCDIC.

PHASE OUTPUT

The output of the phase is to Phase EA in the syntax analysis stage, and consists of one or more text pages containing pairs of records, one or two 84-byte records being written for each record (the extra 4 bytes contain the record length). One record of the pair is used for the source listing on specification of the SOURCE option, and the other is the preprocessed record, used for compilation.

Conversion and/or translation are carried out by the phase, depending on the format of the input records, providing the following output record formats:

Input record	Output records	
	SOURCE listing	Compilation
48-char. EBCDIC	48-char. EBCDIC	60-char. EBCDIC
48-char. BCD	48-char. EBCDIC	60-char. EBCDIC
60-char. BCD	60-char. EBCDIC	60-char. EBCDIC

If the INCLUDE and CS(EBCDIC,60) options apply, then except during processing of %INCLUDE statements, only one record is put out for each input record. This applies to both SYSIPT input and included text.

PHASE OPERATION

The 80-byte source records in card-image form are read into a buffer by the XREAD MACRO. All necessary processing is carried out whilst a record is in this buffer, and the record is output from the buffer direct to the output text stream.

Immediately after a record has been read into the buffer, any BCD-to-EBCDIC conversion required is carried out by means of the translate table TREECDIC. A copy of the record is then output to the current text page, to be used for the source listing on specification of the SOURCE option.

Each byte of active source (that is, within the source margins) in the buffer is scanned by a translate-and-test instruction for one of the characters which are valid as the first character of a 48-character symbol. If one is found it is translated to a value between 1 and 11.

A TRT on the next byte for a character which is valid as the second character of a 48-character symbol gives a value between 1 and 12. These two half-bytes are concatenated into one byte which is translated and tested for a valid combination of characters forming a 48-character symbol.

There are five different types of 48-character symbols to be replaced by 60-character symbols. The separate routines required for replacing the different types are selected by the translate-and-test instruction.

1. 3-character symbols which must be preceded by a character which is not one of A-Z, 0-9, $, _, or &. For example, 'AND'.
   Replacement routines: BA2250, BA2260.

2. 2-character symbols, with restrictions as for the above case. For example, 'GE'.
   Replacement routine: BA2240.

3. 26-character symbols which must be followed by a character which is not one of 0-9. The only example is ',.'.
   Replacement routine: BA2230.

4. 2-character symbols which must be followed by a character which is not an asterisk. The only example is the combination of two slashes (//).
   Replacement routine: BA2235.

5. 2-character symbols without any such restrictions. For example, '..'.
   Replacement routine: BA2220.

When a 48-character symbol has been found, it is replaced by the 60-character equivalent from the table REP0. If the 48-character symbol spans two records the replace is in two parts; the second part of the symbol is replaced in the XREAD buffer, the first part is replaced in the output text stream since the contents of the buffer have already been output. (As the last eight pages remian in main store, the required text page will still be available even if it is no longer the current output page.)

Comments within the records are found by treating /* as a 48-character symbol then skipping by TRT to */.

Quotes are found by including the quote sign (') in the translate tables and skipping by TRT to the next quote sign that is not immediately followed by a second one.

If the INCLUDE compiler option is specified, and the MACRO compiler option does not override it, Phase BA performs %INCLUDE processing.

If a % character is detected outside a comment or quoted string, and then the keyword INCLUDE (optionally preceded by a comment) is found, the statement is interpreted immediately. The DOS macros for searching and reading the source statement library (private and/or system) are used to incorporate the specified books into the text passed to Phase EA. Possible nesting of %INCLUDE statements is catered for by the use of NOTE and POINT logic.

Phase BA creates twc output records for each input source record if either of the options CHARSET(48) or CHARSET(BCD) apply.

In the case where only the INCLUDE option applies, one output record is created for each input source reccrd except those input records which contain a %INCLUDE statement. In that case, two records are generated, the first to be printed by Phase EA, and the second to be processed. This applies equally to included records which contain %INCLUDE statements.

Included books appear as new records; where one record specifies the inclusion of several books, only that part of the including record relating to a particular book will be passed to Phase EA at that point in the text stream; remaining parts of that record will be passed immediately before remaining books are read in. Asterisks will show where text has been erased. As far as the unseen 60-character-set record is concerned, all %INCLUDE specifications and asterisks are blanked out.

For example:

Including record:

```
A=B;   %INCLUDE X, Y;   J=Z;
```

Passed to Phase EA:

CHAR48 :

```
A=B;   %INCLUDE X,**********************
                                .
                                .
                    included text from X
                                .
*******************Y;   *************** *
                                .
                                .
                    included text from Y
                                .
                                .
*******************    J=Z;
```

CHAR60 : as for CHAR48 but asterisks and %INCLUDE specifications are blanked out.

COMPILE-TIME STATEMENT PREPROCESSOR (PHASE CA)

The main function of this phase is to read the source program into the compiler work area (text pages) and to execute any compile-time statements (identified by a preceding % character) so that the source program is modified as specified by the programmer and passed to the syntax analysis stage in a format that can be processed by phases in that stage. The phase is loaded and executed only if the MACRO compiler option is specified.

If necessary, this phase also carries out translation from BCD to EBCDIC representation and/or PL/I 48-character set to PL/I 60-character set.

In addition to providing input to later phases of the compiler, the phase can also generate listings and a punched-card deck if required by compiler options.

PHASE INPUT

Input to Phase CA is from source program records, cards, disk, or tape, using the compiler macro XREAD, or from a partitioned data set as a result of a %INCLUDE statement. Input may be in 48-character or 60-character set, BCD or EBCDIC.

PHASE OUTPUT

Four forms of output can be produced by the preprocessor phase:

1. The modified PL/I source program with all compile-time statements preprocessed, ready for processing by Phase EA, in the form of 84-byte records held in text pages. If the CHARSET(48) option is specified, each 84-byte record is preceded by an 84-byte record in 48-character form. These additional records enable Phase EA to satisfy the SOURCE option if specified.

2. If the INSOURCE option is specified, the phase input from SYSIPT, followed by books specified in %INCLUDE statements (in order of inclusion) are copied to SYSLST for printing.

3. Punched card deck on SYSPCH, being copies of the 84-byte records passed to Phase EA in 48-character set or 60-character set, and EBCDIC or BCD as specified for input. Card columns 73 through 80 contain 'MCDKnnnn', where nnnn is a serial number. This form of output is optional and is obtained by specification of the MDECK option.

4. If any diagnostic messages have been produced, Phase CE is called to print them. The preprocessor general and variables dictionaries are passed so that identifier names may also be printed by Phase CE. If the dictionary pages are not so used they are freed for further use.

Phase Structure

Phase CA is the only phase in the compiler to employ a form of overlay. The phase consists of a total of eight assembled modules (CA, CA1, CB, CB1, CB2, CC, CC1, and CC2) organized into an initialization subphase (CA1) and two partial-overlay subphases (CB and CC), the storage area for the subphases being assembled with the root module CA.

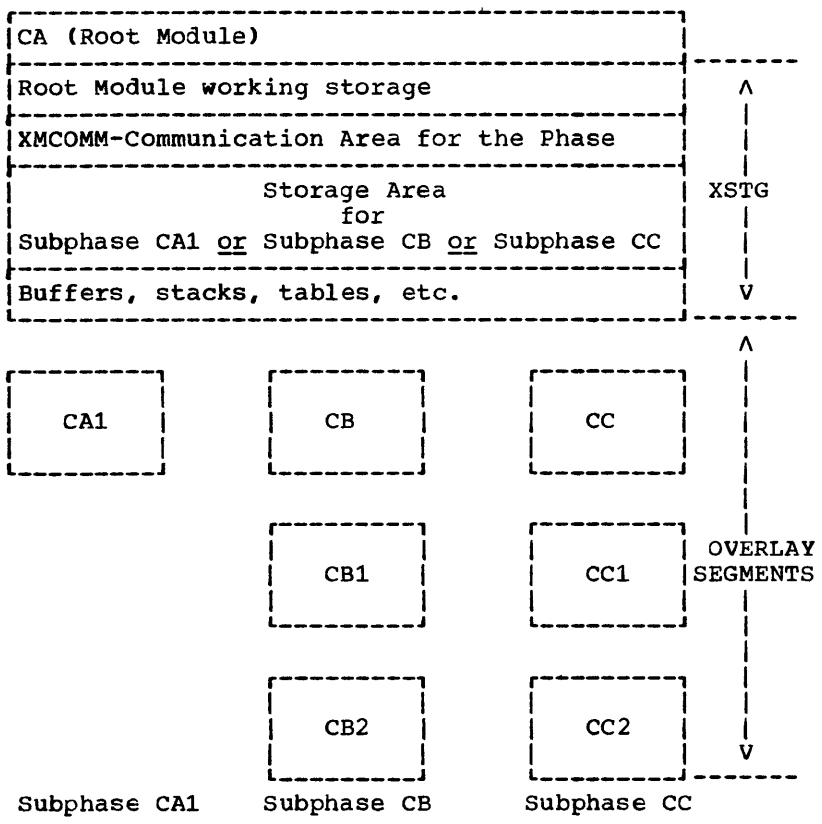


Figure 2.8. Structure of Phase CA

The structure of the phase is illustrated in more detail in section 3, "Program Organization."

Sequence of Processing

Module CA is the resident phase of the compile-time statement preprocessor, remaining in main storage during compile-time preprocessing and, via the control phase (Phase AA), loading the preprocessor subphases. Translation of input text to EBCDIC, if necessary, is also carried out by routine INPUT, contained in this module.

Phase CA contains the initialization module CA1, which initializes tables, pointers, and buffers for use by subphases CB and CC, the print file (if the INSOURCE option is specified), and the punch file (if MDECK is specified).

Subphase CB, which is overlayed on the initialization module CA1, reads and analyzes the input to the compiler. Source text that does not contain compile-time statements is moved directly into the output text stream. Consecutive blanks are replaced by a special marker, and line numbers are encoded and placed in the text. Compile-time statements are decoded and placed into the text pages. An entry is made in the preprocessor general dictionary for each compile-time variable, constant, procedure, label, or INCLUDE identifier.

Subphase CC is overlayed on subphase CB, and scans the text pages produced as a result of the operation of that subphase, executing compile-time statements and writing out PL/I source program text after effecting any necessary modifications. The output consists of text pages containing EBCDIC code, organized in 84-byte blocks ready for processing by the read-in routines of the syntax analysis stage. If the compiler option CHARSET(48) has been specified, the output consists of pairs of 84-byte blocks, the first in each pair being in the original 48-character set, for printing, the second in the 60-character set for processing. If the MDECK option is specified, the output is also passed to SYSPCH.

If a %INCLUDE statement is encountered by subphase CC, it searches the source statement library for the specified bookname and returns control to subphase CB for processing of statements within the book.

When the end of the original input source is reached, subphase CC passes control to Phase CE if there are any diagnostic messages to be printed; otherwise control is passed to Phase EA.

Input/Output Subroutines

All routines in Phase CA and subphases CB and CC that either scan input or write out characters use routines GNC and OUTPTC to read in or write out one character. The calling routine sets a flag as a means of indication of the form of the I/O. This may be:

1. From SYSIPT or SYSSLB (input only).
2. Text page (input and output).
3. Identifier value block (IVB) in the variables dictionary (input and output).
4. Output buffer (XOUTBUF) for insertion into text pages, which are passed to the read-in routines of the syntax analysis stage (output only).

Routines GNC and OUTPTC act on buffers, which they refill by testing the I/O flag calling the the appropriate routine.

Building and Usage of Preprocessor Dictionaries

The various routines of Phase CA and subphases CB and CC use the preprocessor general and preprocessor variables dictionaries to communicate information regarding the compile-time preprocessor variables. General dictionary entries relate to compile-time variables and constants, whilst literal values of character variables and constants are stored in the variables dictionary. Thus an entry in the general dictionary for a character-type identifier contains a pointer to an entry in the variables dictionary. Detailed descriptions of dictionary formats are given in section 5, "Data Area Layouts," figures 5.126 and 5.127.

During the first scan, compile-time statements are analysed and, if an identifier is detected, an entry is made in the general dictionary. If an entry already exists for a particular identifier, the existing entry is used to check for any incompatibility between the uses of that identifier. A hashing technique is employed to reduce the dictionary scan time required to check for and locate an existing identifier entry.

Each entry made in the general dictionary is eighteen bytes long, plus the name of the identifier if necessary.

The preprocessor uses the variables dictionary to hold character string values for variables and intermediate text during the replacement activity of the second scan. Entries, which may be chained together, are of forty bytes.

Reading and Analysis of Source Text

After initialization in module CA1, Phase CA calls subphase CB to scan the input using the routine PH1SCN. This routine employs a subroutine, FINDPC, which calls routine GNC to get characters from the input medium. GNC, in turn, uses a subroutine, INPUT, to read a record from SYSIPT or from a book specified in a %INCLUDE statement, to translate to EBCDIC if necessary and, if the INSOURCE option has been specified, to print each record on SYSLST.

On receiving a character from GNC, subroutine FINDPC examines it for end-of-file or the % character. If either is found, it is passed to PH1SCN. If they are not found, FINDPC transmits the character, by means of the routine OUTPTC, to text page. When successive blanks are found, however, these are replaced by a special marker which is then transmitted. At the start of each new line, FINDPC puts out a line marker followed by an updated line number.

If PH1SCN receives back an end-of-file indication, it returns control to Phase CA, which then calls subphase CC. If PH1SCN is returned the % character, it transmits a character (CPMA) to indicate compile-time text action and starts to scan the compile-time statement.

Processing of Compile-time Statements

The initial text scan examines and translates compile-time text into a postfix Polish format in preparation for the second scan, which handles all conversions and retains operands in a push-down stack.

Thus the statement D = (A + B) || C; for example, would be translated into:

```
PUSH A

PUSH B

ADD

PUSH C

CONCAT

ASSIGN D
```

The PUSH operator causes an operand to be added to its stack of 8-byte entries during the second scan. Operands are represented in the text by 2-byte dictionary references.

All instructions generated by the first scan begin with an operation code byte. Depending on the operation, this may be followed by zero cr by more bytes which form part of that operation.

Each operand will usually have either or both of the following characteristics:

STACK Operators with this characteristic take their operands from the push-down stack. After conversion to the required type (CHARACTER, BIT or FIXED), the result is usually added to the stack.

FIXED These operators are followed by a fixed number of bytes, usually a 2-byte dictionary operand reference.

Figure 2.9, "Code bytes used in ccmpile-time statements," provides a complete descriptive list of possible operators.

Mnemonic	Code	Type	Function	Remarks
ADD	0	STACK	A + B	Fixed result.
MA	1		Entry interpreter scan.	Always precedes text for compile-time statements.
SUB	2	STACK	A -B	Fixed result.
MUL	3	STACK	A * B	Fixed result.
UPDT	4	FIXED	Update line count.	Followed by 3-byte packed decimal line number.
DIV	5	STACK	A/B	Fixed result.
UNMIN	6	STACK	-A	Fixed result.
ASSIGN	7	STACK/FIXED	A = B (assignment)	Dictionary reference of A follows code. A is not stacked. B is unstacked.
NOT	8	STACK	¬A	Bit result.
AND	9	STACK	A & B	Bit result.
OR	10	STACK	A|B	Bit result.
CONCAT	11	STACK	A||B	Character result.
EQU	12	STACK	A = B (equality)	These produce a bit result of length 1 which is '1' if true and '0" if false.
GT	13	STACK	A>B	
LT	14	STACK	A<B	
INC	15	FIXED	Include A.	Dictionary reference of A follows.
ABORT	16		Terminate processing.	Follows from error detected in first scan.
TRA	17	STACK/FIXED	Branch if A is true.	Dictionary reference of branch location follows. A on stack.
TRAF	18	STACK/FIXED	Branch if A is false.	As for TRA.
INV	19	STACK/FIXED	Invoke procedure.	Arguments on stack. Dictionary reference of procedure (2 bytes), argument count (1 byte) and flag byte follow. The procedural text starts with a 2-byte procedure dictionary reference, and dictionary references of the parameters. These are scanned so that the arguments may be matched.
TRAI	20	FIXED	GOTO out of current INCLUDE.	Dictionary reference of branch location follows.

Figure 2.9. (Part 1 of 2). Code bytes used in compile-time statements

Mnemonic	Code	Type	Function	Remarks
PUSH	21	STACK/FIXED	Put A on stack.	Dictionary reference of A and flag byte follow.
PUSHI	22	STACK/FIXED	Put address of A on stack.	Dictionary reference of A and flag byte follow.
RTNS	24		Return to second scan.	Follows text for compile-time statements.
END	25	FIXED	Activate A with rescan.	Dictionary reference of A follows. Also produced by DECLARE statement.
DSB	26	FIXED	Deactivate A.	Dicticnary reference of A follows.
RTN	29	STACK/FIXED	Return from procedure.	Return value on stack. Dictionary reference of procedure follows.
CNVT	30	STACK/FIXED	Convert for RETURN statement.	Return value is converted to procedure's type.
UPDCR	31	FIXED	Update current DO count.	2-byte DO nest count follows.
TRAC	32	FIXED	GOTO	Dictionary reference of branch location follows.
UNPLS	33	STACK	+A	Fixed result.
ENBN	34	FIXED	Activate A with no rescan.	Dictionary reference of A follows.
PAGE	35		Output %PAGE	
SKIP	36	FIXED	Output %SKIP(n)	2-byte count in packed decimal.
CNTRL	37		Output %CONTROL:	Option list follows in text. This list is not syntax analyzed.
NOTE	38	STACK	%NOTE(A,B)	Produces an entry on the message page.
PRNT	39		Output %PRINT;	
NOPRNT	40		Output NOPRNT	

Figure 2.9. (Part 2 of 2). Code bytes used in compile-time statements

Scanning of Compile-time Text: After a % character has been located, control passes from PH1SCN to the STMNT routine, which examines the syntax of the compile-time statement.

After scanning for and checking labels, the type of the statement is determined and control passed to the appropriate routine to process that particular statement type (e.g., the DECLAR routine processes the %DECLARE statement).

When the statement has been processed, control returns to PH1SCN at the entry point DONE. A code to indicate the end of the compile-time statement action (OPRTNS) is transmitted, and scanning of the source text recommences.

Translation of Compile-time Statements: The form of output from the various statement processing routines is generally a series of 3-byte operator-identifier pairs, each consisting of a 1-byte operator and a 2-byte dictionary reference for the identifier.

As a routine locates an identifier within its input, it uses the routine IDSRCH to check whether an entry exists in the preprocessor general dictionary for that identifier. If an entry is found, the existing dictionary reference is used; otherwise, an entry is inserted in the dictionary by the routine ADICT. If the syntax rules for the statement permit expressions, the subroutine PARSE is used to translate the expression into a 3-byte format.

A %INCLUDE statement is placed in the output text stream in the same format as any other statement.

Compile-time procedures and the compile-time statements contained in them are placed in a second text stream, thus avoiding inter-leaving of PL/I source text and %PROCEDURE text.

Processing Intermediate Text: When the end of the input stream has been detected, subphase CB hands control to the root module which calls subphase CC to scan the output from subphase CB. Subphase CC is entered at routine PH2SCN.

PH2SCN scans the intermediate text, one token at a time, using the subroutine TOKSCN. (Tokens are logical units of text, and include identifiers, constants, operators, delimiters, etc.,)

If the token is a constant or an operator (e.g., plus, minus, semicolon, etc.) it is transmitted directly.

If the token is an identifier, routine SRHDIC is called to determine whether or not the named identifier is in the preprocessor general dictionary. If it is, the current value of the identifier replaces the token in the output stream, unless this current value itself contains replaceable compile-time variables. If there is no entry in the dictionary for the named identifier, the token (identifier name) is transmitted directly.

If the token returned by TOKSCN is the operator OPMA (indicating "enter compile-time text"), the routine INTPRT is entered in order that compile-time text statements may be executed.

When the end of the source text is encountered, control is passed to Phase CA so that phase operation can be terminated. If, however, the source text scanned was the subject of a %INCLUDE statement, scanning recommences immediately after the point in text at which the %INCLUDE statement was invoked.

The Output of Tokens: Each token to be written out is passed to the routine OUTPT. This routine employs a 71-byte buffer, into which it attempts to fit the current token, thus ensuring that the token does not span lines unnecessarily. When the buffer is full, routine CLSBUF is invoked to put out an 84-byte record to be printed by Phase EA. If the compiler option MDECK is specified, CLSBUF invokes the subroutine PUNCH to send the record to SYSPCH.

Rescanning of Tokens: Before routine PH2SCN passes an identifier token to routine OUTPUT, the identifier must be examined to determine whether it is a compile-time variable. This examination process is repeated, unless inhibited by the NORESCAN option, until all active compile-time variables have been replaced.

This rescanning is achieved by a process of stacking input pointers. Thus, when a compile-time variable is detected by PH2SCN, the input pointer is stacked and the input scanner is switched to examine the current value of the variable stored in the preprocessor variables

dictionary as IVB storage. This process is repeated if the IVB contains another replaceable token. When the IVB value has been scanned, a pointer is unstacked so that the next lower level IVB is scanned. When the original IVB has been completely scanned, token scanning continues normally.

Scanning of Compile-time Statement Text: When routine PH2SCN detects an OPMA character, the routine INTPRT is entered to scan and process compile-time text.

When a %INCLUDE operator is detected, the text reference of the input pointer is saved. Control passes to Phase CA, which calls PH1SCN to read from the book named in the %INCLUDE statement. The saved input-pointer reference is placed in the dictionary entry for the %INCLUDE statement.

PREPROCESSOR DIAGNOSTIC MESSAGE EDITOR (PHASE CE)

Diagnostic message entries are generated on the error pages by the preprocessor phase (Phase CA) by means of the XMESG macro. Phase CE receives these pages as input, and by reference to various tables, produces the final (sorted) listing of messages. These messages appear in the compiler listing immediately following the statements processed by the preprocessor, and are grouped according to their severity. The specification of the FLAG option (described below) indicates that messages of only selected severity levels must be listed.

The major functions of Phase CE are:

1. To sort the message entries in the message page stream into severity-code order, statement-number order within each severity code and, finally generation-sequence order within each statement number.

2. To insert, and where necessary to decode, the arguments supplied to the message.

3. To print out the message, with inserts, depending on the specification of the FLAG option.

PHASE INPUT

By means of the XMESG macro, a calling sequence is generated to a routine in XROUT. This routine uses a number of fields in the communication area (XCOMM) as follows:

XMPRF contains the page reference of the current message page. This is zero if no message has been created.

XMREM indicates the amount of space remaining in the current message page.

XSTAT is a slot containing the number of the statement currently being processed by a phase.

XMNUM contains an optional numeric argument for a message if one is specified as an argument to the XMESG macro.

XMDRF contains an optional dictionary reference which is to appear in the message.

XMDTP is a single byte used to contain a character indicating the type of dictionary to which XMDRF applies. Both the dictionary reference and the type code are specified as arguments to the XMESG macro.

In addition to the fields described above, registers RB and RC are also used if text is supplied as an argument to the message. Two forms of such text are permitted by the XMESG macro:

1. Implicit text pointers, where text is assumed to occupy the N bytes preceding the address in Register 1. (The value of N is a constant but it may be changed in the XMESG macro definition.)

2. Explicit text pointers, where a pointer to the start of the text and the length of the text are supplied explicitly to the macro. These values are loaded into registers RB and RC respectively by the calling sequence generated.

The XMESGR routine in XROUT generates a message entry (the format of which is described in the XMESGP DSECT) in a stream of pages chained from XMPRF in XCOMM. The routine calculates the size of the message entry to be made, and checks that the current message page contains sufficient space for the entry. If enough space is available, the page is brought into main storage, and the message entry made by moving into the page the contents of all the XCOMM fields described above, plus the message text if the length is not zero. It should be noted that, although these fields are not necessarily used in the message, no attempt is made to determine this in XMESGR since testing would be more time- and space-consuming than moving the fields in every case. Whether or not the fields are actually used is determined by the structure of the message.

If the current message page does not contain enough space to accommodate the latest message, it is not brought into main storage and a new page is requested. The old page reference is then placed in the new page, and the new page reference is placed in the XCOMM fields XMPRF. The message pages are thus chained in reverse order. The latest message is then entered on the new page in the manner described above.

PHASE OUTPUT

The output from Phase CE consists of diagnostic and error messages raised by error conditions occurring in Phase CA.

The preprocessor-error message has the message number IEL0001I and the format

```
$   PREPROCESSOR ERROR NUMBER n DURING PHASE pp.
```

This message is described in detail in appendix B to this publication.

The diagnostic messages output by Phase CE are identified by message numbers in the range IEL0061I through IEL0229I.

TABLES USED BY THE DIAGNOSTIC MESSAGE EDITOR PHASE

The tables used in the operation of Phase CE are all produced by the XMTAB macro and are as follows:

MCDE: The table is generated by the XMTAB macro, when XMTAB is called with MCDE as an argument, and consists of 1000 single-byte entries, one for each possible message. Each byte consists of a set of flags indicating the severity code and information concerning parameters to the message, and editing of it.

The MCDE table is used to set the severity code when building up the sort units. Its use avoids the necessity of specifying the severity code and the statement-number information, both in the message text and the macro call to XMESG, which creates the entry in the error message page stream.

KEYREF: The KEYREF table is produced by the XMTAB macro when called with the argument MESSAGE. XMTAB produces the table by a nested call to the XMCDE macro, with COUNT as a second argument.

KEYREF is used to obtain the appropriate keyword from the keyword table (KEYTAB) using the code obtained from the scan of the coded message string (refer to the sub-heading "Message Decoding" in the description of the phase operation given below).

The table consists of 16 pairs of 2-byte entries, one pair for each permissible length of words contained in messages. Thus the first entry is for 1-letter words (and special and parameter keywords), the second for 2-letter words, and so on. The first member of the pair is the number, in bytes, counting from the start of the keyword table (KEYTAB), of the last word of that number of letters. For example, if there are 12 single-letter words and 10 2-letter words, the first members of the first two entries in KEYREF are 12 and 22, respectively. The second member of each pair is the offset, in bytes, of the first word with the next highest number of letters from the start of KEYTAB. Thus, considering again the previous example, the second members of the first two entries in KEYREF would contain 12 and 32 respectively.

KEYTAB: The KEYTAB table is produced, with KEYREF, by a call to the XMTAB macro.

It consists of one DC instruction for each keyword which may appear in a message. The keywords are arranged in the table in order of length, shortest first, and for ease of updating they are arranged in alphabetic order within each length category. The order of the appearance of the keyword in the table determines the code that is assigned to the word in the coded form of messages. However, this fact is relatively unimportant since the keyword table and the message coding operation are both carried out by the XMCDE macro; the order in the table depends on the order of the arguments to the XKEY macro calls that are nested within XMCDE.

MESREF: This table is created by the XMTAB macro with an argument MESSAGE. It is used to access the coded message for a particular message number.

MESREF consists of 1000 halfword constants, one for each message. If message text for a particular message has not yet been supplied to the XMTAB macro, the relative halfword constant has a value of -1; otherwise the constant is the relative address of the corresponding coded message from the start of the message table (MESTAB).

MESTAB: The message table is produced by the XMTAB macro with MESSAGE as an argument. The individual messages in the table are coded by the XMCDE macro calls nested within the XMTAB macro.

MESTAB consists of strings of coded error messages with one or more code bytes for each word in the message. Each message is preceded by a byte which specifies the length of the message. The code strings appear in order of message number, purely for convenience of updating the macro that produces the table.

All the tables described above are interrelated, but they may be classified into two groups:

1. Glossary of keywords (KEYREF and KEYTAB)

2. Lists of messages (MCDE, MESREF, and MESTAB)

The interrelationships between the tables are generated automatically by the relevant macros. In order to add messages or translate them into other languages, two areas of the macros require change: the message list in the XMTAB macro and the keyword list in the XMCDE macro.

The Message List

The message list appears at the end of the XMTAB macro as a series of nested calls to the XMCDE macro in the form:

```
   XMCDE   N,S,W1,W2,W3,W4,....Wx
```

where

N is a decimal number which identifies the message.

S is the severity code of the message. This may be T,S,E,W, or I for 'unrecoverable', 'severe', 'error', 'warning', or 'informatory'. (Note that this is the only place where the severity code is defined.)

W1-Wx are the words of the message.

If a statement number is to be applied to the message, the character '$' is coded as an argument following the severity code field.

One of the following special control characters may also appear anywhere in the word list:

Z signifies that the preceding and following words must be concatenated.

Q signifies that the following word must be enclosed in quotation marks.

All the words in the word list must appear in the keyword list, either explicitly or without one of the endings ATION, ING, LY, ED, ES, E, S, IZE, IZED.

The Keyword List

The keyword list appears in the macro XMCDE in the form:

```
 .XLn  XKEY C,W1,W2,W3,W4,....Wx

       AIF (&XK7).X2

       XKEY C,Wx+1,Wx+2,....

       AIF (&XK8).XI
```

where

n is the number of letters in the keywords W1 to Wx, and $1 \le n \le 16$.

C is &C1 if $n \le 8$ and &C1&C2 if $n > 8$.

Note: The assembler has a limitation of 128 characters (for DOS) per macro call. In order to overcome this restriction, if the total number of letters in the argument list exceeds two records (one record with the macro call and one continuation record) a second (unlabeled) call to the macro is made, separated by the statement AIF (&XK7).X2. The end of all the keyword lists for a particular word length is terminated by the statement AIF (&XK8).X2.

PHASE OPERATION

The error-message editing phase operates in two stages, to sort and decode the message.

The Message Sort

Sorting is achieved by a sequential scan of the chain of message pages. As each message is encountered, a 12-byte 'sort unit' is created, consisting of the severity code, the statement number, the message number, the generation-sequence number, and the 5-byte text reference of the message. The information regarding the severity code, and whether or not a message contains a statement number, is obtained from the MCDE table that is generated as a by-product from the XMTAB macro.

As the sort units are generated, they are placed in a new page stream. When each page is full, or when the end of the input stream is reached, the page is sorted by means of the SCHELL sort routine before the next output page is obtained.

On its completion, the new output stream is rescanned and the sorted pages merged into new pages, the old pages being discarded as they are emptied.

At the end of the message sort process, two page streams exist: the original message stream and a stream of pages containing sort units in their correct sequence.

Message Decoding

When the message sort is complete, the sort-unit page stream is scanned sequentially. As each sort unit is encountered, the page reference of the corresponding message is obtained and converted to an address. Each message is maintained in the diagnostic message editor in the form of a string of code bytes generated by the XMTAB macro. The appropriate message string is obtained from the table by indexing the message reference table (MESREF) with the message number.

The message string is scanned and each byte or group of bytes is converted to a keyword code, which is used to reference the table of keywords (KEYTAB) by means of the keyword reference table (KEYREF). The keyword may be one of three types:

1. Ordinary keyword. In this case, the keyword is moved directly into the print buffer.

2. Special-purpose keyword. This type is used to indicate either that the preceding and following keywords are not to be separated by a blank, or that some multiple of 256 must be added to the following code to construct the keyword code.

3. Parameter keyword. If a keyword of this type is encountered, the appropriate parameter is obtained from the message entry, is decoded or translated as necessary, and then placed in the print buffer.

When the print buffer is full, or when the message is complete, the message is printed and the scan moves on to the next sort unit.

Message Editing Facilities

In addition to the facilities described above (i.e., statement number, dictionary entry, text, and numeric arguments), the following inserts can also be generated in a message:

1. Two text parameters

2. A single attribute specification

3. Two attribute specifications

4. One attribute specification and a text parameter

These inserts may be generated by their being passed as arguments to the XMESG macro. In all the cases above, a character must also be set as an argument in the severity-code field of the XMCDE macro. Thus, although the XMESG macro may generate a text string containing two text inserts for a message, the string will be decoded as one insert unless this character is specified in the XMCDE call for that message. The character used in this case is 'E', which must be concatenated with the severity code field. This sets a bit in the MCDE vector.

In order to pass these arguments, explicit pointers must be set up to indicate the start and length of a formatted text string. The string would start with a code byte indicating which of the above four cases is concerned. This would be followed by a single-byte attribute specification and/or the text to be included, preceded by its corresponding length specification, depending on which of the four combinations is being passed.

If an attribute specification is passed in this manner, the single byte is decoded by Phase CE and the corresponding attribute is moved into the print buffer.

A further editing facility accumulates several generations of the same message in a message page and then prints it only once (i.e., instead of the message appearing in the listing N times for N different statement numbers, it is printed only once, following a list of statement numbers to which that particular diagnostic applies). Once again, for this facility to be implemented at message-decoding time, the corresponding bit must be set in the MCDE vector. This is achieved by concatenating the character 'C' with the severity-code field argument to the XMCDE macro. This accumulating facility can be applied only to those messages generated by the XMESG macro and having the statement number as the only parameter.

When, by reference to the MCDE table, a message in the message page stream is found to be of the cumulative type, two sort units are created for it. The first is created in the normal way, with the statement number field as specified in the XMESG call.

The second, however, is created with the normal statement field set to zero, the actual statement number being saved elsewhere in the sort unit. An entry is then made in a push-down stack, which always contains the lowest statement for which a particular cumulative message is generated. In addition to the statement number, the entry also includes the message number and the page reference of the first 'zero-statement-number' sort unit for the message.

Each time a further generation of the same message is encountered in the page stream, another sort unit with a zero statement number is created and the push-down stack is then checked to see if the new statement number is lower than the previous entry for that message. If this is the case, the previous entry in the stack is replaced by the new number and another sort unit is created with the statement number in the normal statement number field.

When the whole of the message page stream has been scanned, therefore, for a cumulative message which has been generated for N statements there will exist:

- N sort units with zero-statement-number fields (the actual statement number being held in the sort units).

- An entry in the push-down stack which contains the lowest statement number for which the message was generated, and a reference to the start of the zero-statement-number sort units.

- A normal sort unit for the message, containing the lowest statement number.

It should be noted that, if the first generation of the message in the message page stream should happen not to be the lowest statement number for which the message is relevant, a redundant sort unit will be created. This would be ignored at message-decoding time.

During the message sort, all the zero-statement-number sort units are shifted to the beginning of the sort-unit stream, and the normal sort unit for that message is moved to the correct position for printing. When this normal sort unit is encountered at message-decoding and printing time, the stack entry is checked to see if the message has been printed already (i.e., if this is in fact one of the above-mentioned redundant sort units). If the message has not been printed, the zero-statement-number sort units are referenced and all printed out in the position where the lowest statement number would have been printed.

Implementation of Compiler Options

FLAG Option: Diagnostic messages produced during preprocessing are grouped in order of severity. The FLAG option specification indicates the minimum severity level for which diagnostic messages, if produced, will be listed. The severity level is specified in the value list of the FLAG option, as follows:

FLAG(I) All diagnostic messages listed.

FLAG(W) All diagnostic messages except 'informatory' messages listed.

FLAG(E) All diagnostic messages except 'warning' and 'informatory' messages listed.

FLAG(S) Only 'severe' errors and 'unrecoverable' errors listed.

Note that the specification of FLAG is equivalent to FLAG(I). The severity levels are discussed fully in the Programmer's Guide for this compiler.

If, due to the specification of the FLAG option, any levels of messages are to be suppressed, suppression takes place early in operation of the phase and no sort units are created for those levels of messages.

SYNTAX or NOSYNTAX Option: Phase CE also implements the option SYNTAX|NOSYNTAX(W|E|S).

SYNTAX specifies unconditional syntax check after preprocessing unless 'unrecoverable' errors are encountered.

NOSYNTAX specifies unconditional termination of the compilation after preprocessing, without any syntax check being carried out.

The specification of NOSYNTAX with a value list makes the syntax check conditional upon errors detected during compile-time preprocessing. The value list specifies suppression of the syntax check if a diagnostic message equal to or exceeding the severity indicated by the value is encountered during preprocessing. Thus the syntax check is suppressed as follows:

NOSYNTAX(W) When 'warning', 'error', 'severe', or 'unrecoverable' diagnostics have been issued.

NOSYNTAX(E) When 'error', 'severe', or 'unrecoverable' diagnostics have been issued.

NOSYNTAX(S) When 'severe' or 'unrecoverable' diagnostics have been issued.

If NOSYNTAX is specified, or the specified severity value is equalled or exceeded, Phase CE returns control to the control phase (Phase AA). If however, the option SOURCE has also been specified then the SYNTAX flag is set and Phase EA is loaded to print the required source listing. Phase EA will immediately pass control back to Phase AA after so printing.

SYNTAX TABLE BUILDER (PHASE EC)

Phase EC makes space for phase EA by copying Translate and Keyword tables onto a text page. If possible, a single page is used, otherwise two pages are used. The addresses of the tables (in 5-byte format) are stored in the XCOMM fields XRIOCH and XDOCH.

SYNTAX ANALYSIS STAGE

The main function of phases in the syntax analysis stage is to analyze all data in the source program, and to process it so that only statements that are valid within the PL/I language implemented by this compiler are passed for processing by later phases. When an erroneous statement is detected, a phase may attempt to correct the error by making an assumption about the intention of the statement, or may delete the statement from the text passed to other phases. Diagnostic messages are generated to indicate errors that are detected.

Because of the wide scope of the PL/I language, the function is performed by three phases, EA, EE, and EI. Each phase checks the syntax of a particular range of PL/I statements. Phases EA and EE are loaded and executed in every compilation; Phase EI is only loaded and executed if the source program contains stream oriented input/output statements.

During processing, the text is translated into an internal code and organized into a format that is convenient for processing by subsequent phases. Certain types of statement are collected in a separate stream of text for the convenience of phases in the dictionary build stage.

SYNTAX ANALYSIS - PASS 1 (PHASES EA AND EC)

Before syntax analysis commences, phase EC is loaded. Its function is to move Keyword and Translate tables into one or two text pages for use by syntax analysis.

Phase EA performs the following functions:

1. Reads the source program input records into a text-page work area.
2. Prints the source program listing if required.
3. Removes comments and excess blanks from the text.
4. Translates the input text into an internal format code.
5. Identifies each statement and allocates statement numbers.
6. Analyzes completely the syntax of a number of types of PL/I statements, and issues error messages if required.
7. Builds the compiler main text stream by copying all statements (including those not fully analyzed) onto text pages.
8. Inserts chain fields in the text stream to assist Phase EE in de-nesting the program block structure.

PHASE INPUT

Input to the phase consists of records containing the compiler source text, written in 60-character EBCDIC. These records may be read into the compiler input buffers from SYSIPT, or may be passed to Phase EA on text pages as output from a compiler preprocessing phase (Phase BA or Phase CA).

If the external format of text is not 60-character EBCDIC, the text passed to Phase EA from the preprocessing stage consists of record pairs, one record being in the original source form (used for the source listing), and the other record of a pair being the preprocessed record in 60-character EBCDIC form (used for compilation).

The input contains free format PL/I statements and comments within the limits of the MARGINS option.

PHASE OUTPUT

Output from the phase consists of the main text stream, plus a diagnostic message stream if the source program contained syntactical errors.

The output text consists of a stream of PL/I statements in source statement order, in which all statements are numbered, and statement elements are retained in source program order. All characters are in compiler internal code (see figure 5.88). A statement header is inserted in front of every statement. The following types of statement are checked for correct syntax, and appear in Type 1 text format (see figure 5.60):

PROCEDURE statements (options are not processed at this time)

ENTRY statements (options are not processed at this time)

BEGIN statements (options are not processed at this time)

IF - THEN - ELSE statements and clauses

%SKIP and %PAGE statements

%PRINT and %NOPRINT statements

SIGNAL and REVERT statements

FETCH and RELEASE statements

ASSIGN statements

FREE statements

ON statements

END statements

GOTO statements

WAIT statements

RETURN statements

NULL statements

LEAVE statements

SELECT statements

WHEN statements

OTHERWISE statements

Following the statement header for each PROCEDURE, BEGIN, and ON statement, a 17-byte chain field is inserted for use by Phase EE (see figure 5.61).

The page-handling routines in the control stage are called as required to acquire page spaces for the text stream output.

PHASE OPERATION

Source Text Read-in

Phase EA acquires a page space for use as a read-in work area. This page is treated as a scratch page and is discarded on completion of processing by the phase.

The READR routine has records copied from SYSIPT into the input buffers, and from there copies the records, one at a time, into the read-in work area. If the SOURCE option is specified, each source record is copied into the print file buffer, for inclusion in the printed source listing, immediately before the next record is read.

If the source program is coded in either BCD or 48-character EBCDIC, the input to Phase EA consists of the output from either Phase BA or Phase CA. This output is passed to Phase EA in record format on text pages. Records are paired, the source record preceding the 60-character record. Each 60-character EBCDIC record is copied into the read-in work area. If the SOURCE option is specified, each source record is copied into the print file buffers for inclusion in the source listing.

As each record is read in it is translated, byte for byte, from 60-character EBCDIC into the compiler internal code. This code is shown in figure 5.88. Each character of all identifiers and constants is translated into internal code. All operators, including two character operators (e.g., ||, ->), are replaced by single code bytes.

The translated record is scanned from the left. Comments (identified by their inclusion within /* */ characters), are replaced by blanks. All remaining characters are considered to be part of the PL/I program and control is passed to the STARTA routine for processing of the records as described later.

When a record has been processed, the processing routines call the READR routine as required to read-in and translate further records. Within the read-in work area, records are concatenated so that they form a continuous stream of text. If there is insufficient space for a record containing part of a current statement to be read in, existing text within that statement is moved to the start of the read-in work area, so that the entire work-area page is available for the statement. If a statement, other than a DECLARE, DEFAULT, or ALLOCATE statement, cannot then be contained within the read-in work area, it indicates that the maximum permissible statement size has been exceeded: an error message is issued, the statement is ignored, and compilation continued. DECLARE, DEFAULT, and ALLOCATE statements can be split at any comma not contained within parentheses, and spanning of pages by these statements is permitted.

If a statement containing a string item cannot be contained in the work-area, a quote character is inserted in front of the first semicolon (if any) found within the string. The remainder of the string item is considered to belong to another statement.

Detection of any invalid characters not contained by quote characters results in deletion of the containing statement.

Statement Numbering

The limits of statements are recognized by the appearance of semicolons. All statements, including compound statements, statements included in a compound statement, block- and group-delimiting statements, and null statements, are numbered as they are copied into the read-in work area. THEN clauses are not regarded as statements for numbering purposes. Diagnostic messages created throughout compilation refer to these statement numbers. The number of the first statement in each record is printed in the source listing.

Statement Headers

To enable statements to be easily identified, and basic information about each statement to be readily accessable, a 6-byte statement header is created and inserted before each statement. Information is inserted in the various fields of the statement header during processing by this phase. The format of statement headers varies slightly according to the type of statement; the statement header may be followed immediately by the body of the statement, or other information such as a label list or a chain field may be inserted. On completion of processing by this phase, the text is organized in a format referred to as Type 1 text. The general structure of a statement in this format is shown in figure 2.10.

Bytes	Content	Use	
0	SL or SN	Marker to indicate labeled or unlabeled statement.	<
1	Code byte	Indication of statement type, e.g., IF, GOTO, assignment.	Statement header
2-3	Statement number	Number of source statement.	
4-5	Prefix options	Bits set to indicate conditions (e.g., SIZE, OVERFLOW) that are enabled for the statement.	<
6...	Label list	If the first byte indicates a labeled statement, these bytes show the length of the label list and the length and name of each label, e.g., LAB:L: would be shown as seven bytes containing 7 3 LAB 1 L.	
	Chain field	A 17-byte chain field is inserted in all PROCEDURE, BEGIN, and ON statements.	
	Statement body	The statement body with statement elements in source order. Keywords are represented by code bytes, identifier names are preceded by a length byte, and constants are preceded by a marker and a length byte.	
	Semicolon	End-of-statement marker.	

Figure 2.10. General format of a statement in Type-1 text

Prefix Processing

Each PL/I item is tested for the presence of a colon, indicating that it is a statement prefix. There are two types of prefixes, condition prefixes and label prefixes. Condition prefixes must precede label prefixes.

Label prefixes are placed immediately after the statement header. Each label name is preceded by its length. If there is more than one label, the total length of the label list is inserted before the first label length.

When condition prefixes are detected, bits are set in bytes 4 and 5 of the statement header to indicate the conditions or options that are enabled for that statement. Exceptions to this are the CHECK and NOCHECK condition prefixes, which may have argument identifier lists. These two condition prefixes are treated as separate statements, and are inserted before the main statement with their own statement headers. Thus, the source statement:

```
(NOOFL, SUBRG, CHECK (A,B)):(NOCHECK(C)):LAB1:LAB(3):A,B,C=1;
```

would appear in the output from Phase EA as:

```
CHECK statement header (A,B);

NOCHECK statement header (C);

ASSIGN statement header, label list length, 4LAB1 6LAB(3)1A 1B 1C=1;
```

The prefix option field in the ASSIGN statement header would contain flag settings for the NOOFL and SUBRG conditions, plus any applicable default or inherited options. The prefix option fields in the CHECK and NOCHECK statement headers would contain zeros. If CHECK and NOCHECK condition prefixes precede a PROCEDURE or BEGIN statement, the block level and count are inserted in the prefix option bytes.

Keyword Identification

Each identifier in a statement is treated as a potential PL/I keyword. Phase EA contains tables that contain every PL/I keyword handled by the phase. These tables are searched for a keyword that matches the identifier. If a match is found, and if the identifier is in a position that makes it a valid keyword, it is replaced by a predefined 1-byte or 2-byte code (see figure 5.88).

Keywords are classified according to their context in the source program, e.g., statement identifier, verb, ON-condition. Keyword tables are organized on the basis of class of keyword and length of keywords within that class. Within each classification, keywords are grouped in tables containing keywords of similar length. The format of keyword table entries is shown in the following sample of entries in the tables of verb-class keywords of three bytes length:

```
STL3  DC   X ' number of keywords in this table'
      DC   X'0E170D' (keyword in internal code)
      DC   AL1(END) (replacement code)
      DC   X'0D0C15' (keyword in internal code)
      DC   AL1(DCL) (replacement code)
```

Two levels of directory are used when searching the keyword tables. When a statement analysis routine calls the KYWD routine, it passes a numeric argument indicating the class of the potential keyword. This indicates the first level of directory to be used to find the address of the second level directory. The second level directory indicates the address of a keyword table for the same class and length as the potential keyword.

When a matching keyword table entry is found, the replacement code is returned to the caller routine. If a matching entry cannot be found, a code of X'FF' is returned to the caller routine.

Because there are more than 256 PL/I keywords, some of them are replaced by a 2-byte code. These keywords can always be recognized by their classification, and the first character (always X 'D9') is not shown in the keyword tables.

Verb Identification

When the first identifier that is not enclosed by parentheses and is not followed by a colon is found, it indicates that all prefixes to a statement have been processed. This identifier is treated as a

potential verb keyword and the keyword tables are searched for its replacement code.

If a matching verb keyword is found, the replacement code is inserted in the second byte of the statement header to indicate that statement type. The routine that processes the particular type of statement is called to analyze and process the body of the statement.

If the identifier is found not to be a verb, it is assumed to be the left-hand side of an assignment statement. The ASSIGN code is inserted in the statement header and the appropriate routine is called.

If the first identifier is a verb, analysis of the statement may reveal that the statement is an assignment statement, e.g., GOTO = 3; or PROC(A,B,C)=4;. Detection of the assignment character (=) will result in reprocessing of such a statement.

Statement Analysis

The syntax of each statement is checked for its compliance with the syntax defined for that type of statement within the PL/I language. As described above, the statement type is determined according to the presence of certain verb keywords or operators, and an appropriate routine is called to analyze and process each type of statement.

The analyzing routine scans the statement body for the presence of mandatory and optional items. In general, these items consist of keywords that apply only to the particular type of statement, and optional arguments applying to the keywords. Keywords can often appear in variable order. When a keyword is found, it is replaced by its internal code, and a subroutine is called to process any arguments present. These optional arguments can be in the form of:

- A constant.
- A simple identifier e.g., NAME.
- A subscripted or qualified identifier e.g., NAME.A(3) ->Q.
- An expression, which consists of one or more of the above items, plus operators and parentheses.

These items are converted to a series of identifiers and constants, separated by operators and delimiters, by the following subroutines used by all statement analysis routines:

Item type	Subroutine name
Arithmetic constant	ACONST
String constant	SCONST
Identifier	IDENTR
Qualified identifier	SQUID
Expression	EXP

When the syntax has been checked, the statement body is output, after the statement header and label list, in source-program order as follows:

Keywords - replaced by 1-byte or 2-byte code.

Identifiers - the identifier name, preceded by a field containing its length value.

Constants - the constant, preceded by a field containing its length value, preceded by a 1-byte constant-marker.

Delimiters - the delimiter for an IF clause is "THEN". The delimiter for a THEN clause followed by an ELSE clause is ";ELSE". In all other cases the delimiter is a semicolon.

This is the basic format of Type-1 text.

Statement Error Handling

The handling of syntax errors within statements depends upon features peculiar to the statement type. Errors are mainly handled by deletion of the part of the statement in error. Depending upon whether the remaining part of the statement retains the sense of the program, subject to checking by later compiler phases, deletion is kept to a minimum. If cascading of errors throughout the program is likely, the erroneous statement is replaced by a null statement, thus enabling labels to be retained. In a few cases of simple errors, an attempt is made at correction. The following example indicates typical methods of handling errors in a statement for which the valid syntax can be represented as:

```
VERB KEYWORD1 KEYWORD2(NAME) KEYWORD3(EXP);
```

If an error is found within "NAME" or "EXP", the complete KEYWORD option is deleted.

If one or more parenthesis around "NAME" or "EXP" is missing, it is usually inserted.

If "KEYWORD1" is mandatory but is missing, the statement is replaced by a null statement.

If "KEYWORD2" is optional but is in error, only "KEYWORD2(NAME)" is deleted.

If "VERB" cannot be identified, the statement is replaced by a null statement.

Program Block-structure Checking

Blocks are numbered in order of their appearance in the source program. The block structure of the program is checked for the logical occurrence of end-of-program and end-of-file indications.

A push-down stack is used to check the block structure. The stack consists of a series of entries, one for each statement, or IF, THEN, and ELSE clause, that is significant in the program block structure. As each item is entered in the stack it is allocated a block number. The format of a stack entry is:

Symbolic field name	Field length	Field content
STCODE	1 byte	Code for type of entry (e.g., PROC)
STBKC	1 byte	Block number
STPO	2 bytes	Prefix options
STLLTH	1 byte	Length of label
STLBL	31 bytes	Label (if any)

The STCODE, and the stacking/unstacking action, for various statement types is as follows:

Statement Type	STCODE	Stacking Action
PROC	80	Make new stack entry
BEGIN	81	Make new stack entry
DO	82/84	Make new stack entry
END		Delete from the stack all entries for the block just ended.
ON	20	Make new stack entry
On-unit	A0	Make new stack entry
IF	41	Make new stack entry
THEN	42	At THEN clause change STCODE of IF entry to STCODE of THEN.
ELSE	44	STCODE of top entry must be THEN: delete this entry.

After each statement is processed, the top entry in the stack is checked. THEN entries are deleted if no ELSE clause is found. Errors such as 'IF - THEN - END;' are checked for, and are corrected by inserting NULL statements in appropriate places. RETURN statements inside BEGIN clauses are deleted.

If an end-of-file indication (/*) is detected before the stack entries indicate the logical end-of-program, END statements are inserted to enable compilation to continue. If the stack entries indicate the logical end-of-program before the end-of-file indication is detected, the current END statement is deleted and the remaining source statements are processed.

If the push-down stack overflows, it indicates that a compiler limitation on nesting of blocks has been exceeded, and compilation is terminated.

Chaining of Nested Blocks

To assist Phase EE in the de-nesting of blocks, a 17-byte chain field is inserted between the statement header or the label list, and the body of the statement for each PROCEDURE, BEGIN, and ON statement. The format of these chain fields is shown in figure 5.61.

ON statements are chained in a special way. If an ON statement is associated with SYSTEM, or is associated with a NULL on-unit, it is not treated as a block statement. All others are treated as a BEGIN block on-unit, even though they may be single statements. In the latter case, the chain field is inserted in the ON statement body; the following BEGIN statement (if any) has no chain field. If there is no BEGIN block, the END statement pointer is set to X'FF'.

The following example illustrates chaining of statements in the output from Phase EA.

The source statements:

```
1.  A:PROC(P,Q);
2.  B:ON SUBRG BEGIN;
3.    ON OFL(CHECK(X)):GOTO L;
4.  L:END
5.  E:ENTRY
6.  (CHECK(Y)):BEGIN;
7.  END A;
```

would be chained on output from Phase EA as shown in figure 2.11.

The example of chaining illustrates the following features that are used during processing by Phase EE:

1. Every block-heading statement contains a pointer to the END statement for that block, thus enabling Phase EE to skip all statements within a block. The exception is on-units that consist of a single statement, and therefore have no END statement. In such cases, Phase EE skips the next statement, plus any CHECK and NOCHECK statements attached to the on-unit.

2. The chain of next-block references passes through dummy statement headers embedded within ON statements. These dummy headers have a unique code byte, X'9B' (ON-BEGIN) which Phase EE recognizes. The next-block chain points to the first CHECK or NOCHECK statement preceding a block so that these statements are included in a block. In the case of ON-BEGIN statements, the CHECK and NOCHECK identifier lists follow the ON statement.

3. The ENTRY statement chain fields are only used in PROCEDURE blocks; BEGIN blocks cannot contain ENTRY statements, (for uniformity they contain an unused chain field).

4. The chain field of each block header is preceded by a 2-byte field containing values for the block nesting-level and block-count. Phase EA inserts the total block-count for the program in the XBLKCT field in XCOMM. When the block-count maintained by Phase EE reaches the value in XBLKCT, it indicates that the end of the next-block chain has been reached. For this reason, the next-block chain field in statement 6 in the example is shown as not-used.

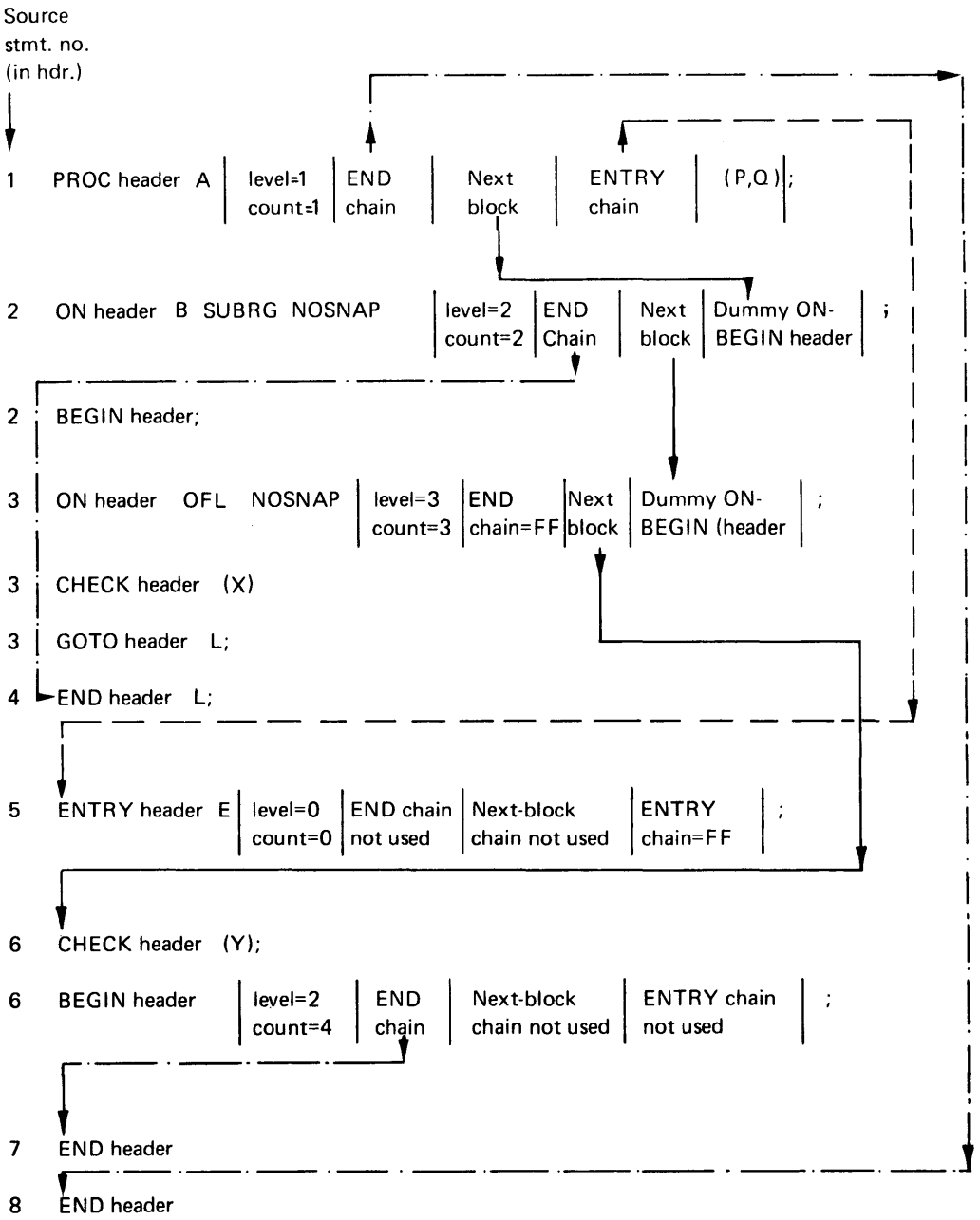


Figure 2.11. Chaining of statements in the main text stream output from Phase EA

SYNTAX ANALYSIS - PASS 2 (PHASE EE)

Phase EE continues the analysis of the source program started by Phase EA. In doing so it performs the following functions:

1. Completes the analysis of statement types partly processed by Phase EA, i.e., PROCEDURE, BEGIN, and ENTRY statements.

2. Analyzes, and converts to Type-1 text, the following types of statement:

ALLOCATE	REWRITE
DELAY	DELETE
STOP	UNLOCK
EXIT	OPEN
DISPLAY	CLOSE
CALL	LOCATE
READ	DECLARE
WRITE	DEFAULT

3. Builds a new main text stream in which the statements are written in de-nested block order (i.e., all statements within a block appear in a contiguous text stream preceding all statements within a contained block).

4. Builds a second text stream containing types of statements for which easy access is particularly required by phases in the dictionary build stage.

5. Scans for stream-oriented input/output statements (i.e., GET, PUT, and FORMAT statements) to determine whether the next phase to be loaded is Phase EI, which analyzes such statements, or Phase GA.

PHASE INPUT

The input to this phase consists of the main text stream generated by Phase EA. The input is written on text pages, chained in sequence.

The text consists of a contiguous sequence of statements in source-program order. Each statement is numbered and preceded by a statement header. Block-heading statements have chain fields preceding the statement header or inserted in the body of the text. Statements analyzed by Phase EA are in Type-1 text format. Other statements are in the form of source-program statements translated into compiler internal code, with all keywords represented by a one or 2-byte code.

PHASE OUTPUT

Output from the phase consists of two streams of text, the main text stream and a second text stream known as the dictionary text stream. In both text streams, all statements (except stream-oriented input/output statements) are in Type-1 text format.

In both text streams the statements are written in de-nested block order, e.g., if block A contains block B, all the statements in block A appear before the first statement in block B.

Some statements inserted in the dictionary text stream are also retained in the main text stream. The content of each statement may vary between the two text streams.

The format of items in both text streams output by Phase EE are shown in section 5: "Data Area Layouts."

The Main Text Stream: a new text stream is built during processing by Phase EE, and the text pages input from Phase EA are discarded on completion of the processing of all text.

The new main text stream contains no chain fields. All valid PL/I statements in the source program are written in the text stream in de-nested block order. Statements are written in Type 1 text format, with the following exceptions:

1. PROCEDURE, BEGIN, and ENTRY statements are represented by statement headers, labels, and prefix options only. In this form they indicate the positions of entry points.

2. GET, PUT, and FORMAT statements are not processed by this phase, and appear in the same format as at input to the phase.

BEGIN blocks are repositioned as required for de-nesting, and a compiler-generated label is placed before each one so that it can be identified in the CALL statement that is used to replace the BEGIN statement in its original in-line position.

DECLARE and DEFAULT statements are not included in the main text stream output.

The Dictionary Text Stream: To simplify scanning in the dictionary build stage, PROCEDURE, ENTRY, BEGIN, DECLARE, DEFAULT, LOCATE, and ALLOCATE statements are copied into a separate text stream. These statements are written in de-nested block order and chained for rapid access (see figure 5.72). PROCEDURE and ENTRY statements associated with contained blocks are included with statements in the containing block (in which they are known in the PL/I sense).

Each block is preceded by a 28-byte block header containing five chain fields (see figure 5.71). The first chain field contains a forward pointer to the next block. The other four chain fields, in order of appearance, contain chains for PROCEDURE and ENTRY, DECLARE, DEFAULT, and LOCATE and ALLOCATE statements within the block. These statements appear in source-program order and are chained backwards. BEGIN and ON-BEGIN statements appear with the compiler label generated for identification in the main text stream, and are chained with ENTRY statements.

To enable entry points in the main external procedure block to be handled by the same routines as entry points in the rest of the program, a header for an imaginary block of zero nesting level is inserted at the beginning of the dictionary text stream.

PHASE OPERATION

The input consists of a contiguous sequence of text in source program order. When this text is read sequentially, a read-ahead technique that employs the XBRIC macro is used, enabling page-handling requirements to be determined in advance. When chains are followed, and scanning of the text becomes non-sequential, it is sometimes necessary for the scan to shift forwards or backwards to a location within another text page. Such cases are handled by the JUMPTXT routine and special-case coding in the XBRICM routine. The XTXRF macro is used for non-sequential scanning of text. Input pages are given the status DISCARDED on completion of all processing by the phase.

Statement Analysis

The syntax of statement types processed by this phase is checked by special analyzing routines for each type of statement. The method of checking and the handling of errors is similar to that described for Phase EA.

De-nesting of Contained Blocks

To assist in the resolution of names during the dictionary build stage, contained blocks are de-nested so that all statements within a block are contiguous on output from the phase. The chains set up by Phase EA are used for this purpose. The two output streams are built simultaneously as statements are scanned and processed.

The output streams built by this phase are illustrated in the following example, showing output in the main text stream and in the dictionary-text stream. The source program example used is similar to that used in the Phase EA description. A few statements have been added for illustration purposes.

The following source statements:

```
1.  A:  PROC(P,Q);
2.      DCL BB FIXED;
3.      B: ON SUBRG BEGIN;
4.         DEFAULT RANGE(X) FLOAT;
5.         ON OFL(CHECK(X)): GOTO 1;
6.      L: END;
7.      E: ENTRY;
8.         DCL CC STATIC, AA CHAR (5)CTL;
9.         (CHECK(Y)): BEGIN;
10.        ALLOCATE AA CHAR 3;
11.    END A;
```

would appear in the main stream output from Phase EE in de-nested block order as follows:

```
1.  PROC-header level=1 count=1 A;
2.  ON-header B SUBRG NOSNAP FF0002;
7.  ENTRY-header level=1 count=1 E;
9.  CALL-header FF0004;
12. END-header ;
3.  ON-BEGIN-header level=2 count=2 FF0002;
5.  ON-header OFL NOSNAP FF0003;
6.  END-header L;
5.  CHECK-header level=3 count=3 (X);
5.  ON-BEGIN-header level=3 count=3 FF0003;
5.  GOTO-header L;
5.  END-header ;
9.  CHECK-header level=2 count=4 (Y);
9.  BEGIN-header level=2 count=4 FF0004;
10. ALLOCATE-header;
11. END-header ;
```

Note the following features shown in the example:

1. Block nesting-level and count values are retained in the block headers to assist name resolution by later phases.

2. CHECK and NOCHECK headers contain block nesting-level and count values for use by Phase KT.

3. Special entry-names, consisting of FF00 and a 2-digit block-count number, are generated for BEGIN and ON-BEGIN blocks.

4. DECLARE statements (statements numbers 2 and 8) and DEFAULT statements (statement number 4) are not required in the main text stream: they are included in the dictionary-text stream.

5. ALLOCATE statement headers only appear in the main text stream to mark the statement position (statement number 10). The identifier and its attributes (if any) are included in the dictionary text stream.

6. There are no chains in the main text stream output from this phase.

The same source program statements would result in entries being made in the dictionary text stream as shown in figure 2.12.

Note the following features illustrated in the example:

1. Block nesting-level and count values are inserted in entry statements. For other types of statement, these values are shown in the block header.

2. A block header is created for the on-unit block created by the expansion of source statement number 5 (see main text stream example). The ON-BEGIN header in this block is treated as being part of the containing block, and therefore all chains in the block header, except the next-block chain, are set to X'FF'. The block header is generated for block-nesting level and count information in the dictionary build stage.

Determination of Next Processing Phase

If stream-oriented input/output statements are found in the text stream, they are copied into the output stream without change. A flag is set in the phase work area to indicate the presence of such statements. When the end-of-program indication is detected, the setting of this flag is tested. If it is set, an XPST macro is issued to indicate that Phase EI is to be loaded. If it is not set, the XPST macro specifies that Phase GA is to be loaded.

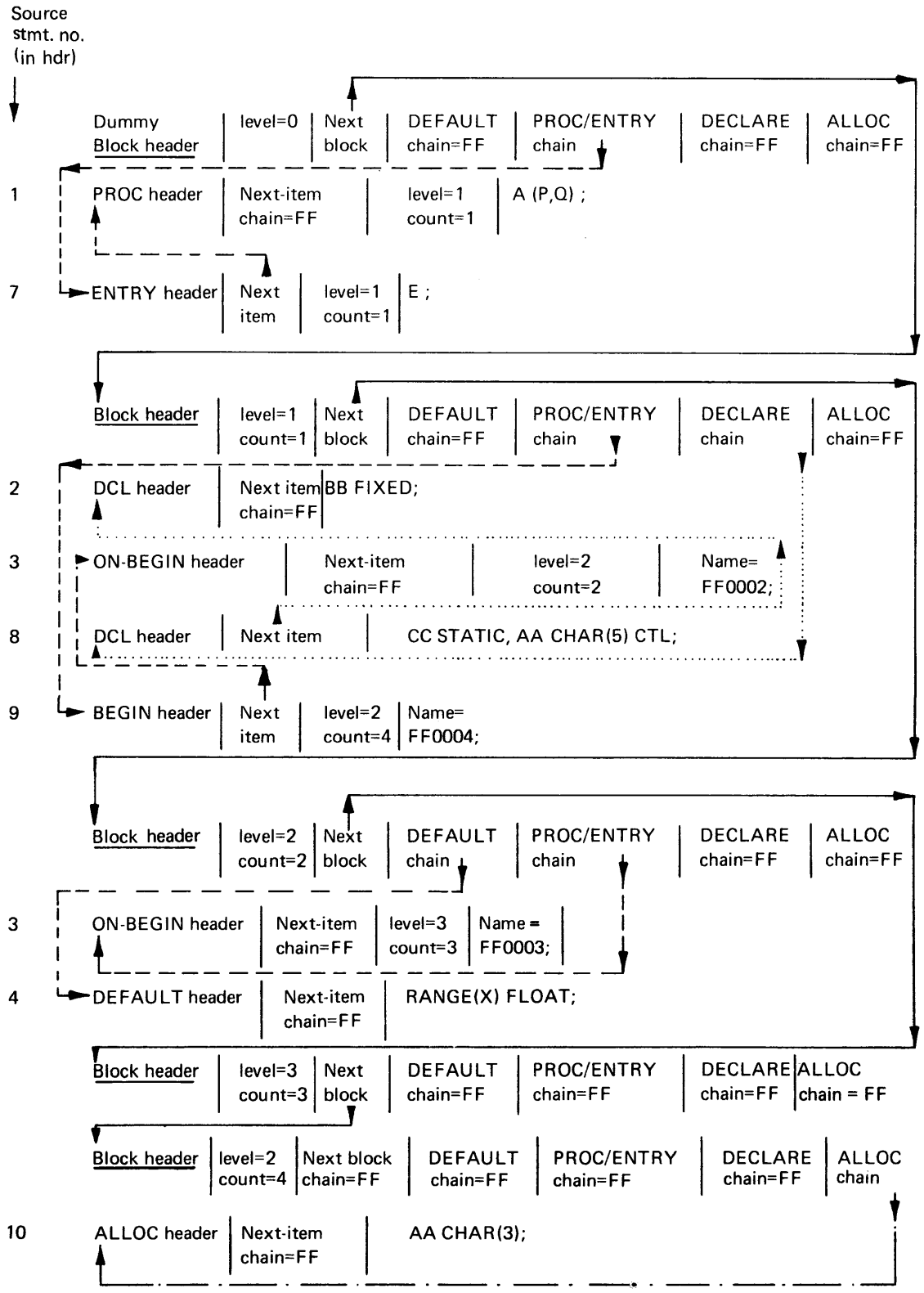


Figure 2.12. Chaining of statements in the dictionary text stream output from Phase EE

SYNTAX ANALYSIS - PASS 3 (PHASE EI)

Phase EI is only loaded and executed if the source program contains GET, PUT, or FORMAT stream oriented input/output statements. These statements are checked for correct syntax, and output in Type 1 text format. Diagnostic messages are issued if errors are detected.

While the stream I/O statements are being processed, the following functions are performed to assist processing by Phase II:

1. The contents of DO specifications within data lists are re-ordered.

2. Markers are inserted at each end of format item iterations.

PHASE INPUT

Only the main text stream output from phase EE is scanned. The dictionary text stream is not scanned or accessed.

PHASE OUTPUT

The output consists of the main text stream in which all statements appear in Type 1 text format in deblocked order. Only syntactically correct statements are included.

PHASE OPERATION

Every statement in the main text stream is scanned. Statements other than stream-oriented input/output statements are copied unchanged into the output stream. If a GET, PUT, or FORMAT statement is detected, appropriate routines are called to process them.

Stream oriented input/output statements can contain three main types of item:

Options e.g., PAGE, SKIP, etc.,

Data lists

Format lists

GET and PUT statements can contain all three types of item. FORMAT statements contain only format list items. The relationship between the three types of statement enables common routines to be used for much of their processing.

A file entry is inserted by default if not specified in a PUT or GET statement.

Format List Iterations: Special markers are inserted in format lists to indicate the beginning of the format list, and the limits of iteration and repetition factors within the list. The markers are:

Symbolic name	Code	Indication
FMATLST	X'C8'	Format list follows
FIT	X'AC'	Precedes format iteration factor
FITE	X'AB'	Ends the format iteration factor

DO Specifications: The specifications contained in DO statements within data lists are re-ordered so that they appear in a format similar to iterative DO statements. The clause 'BY 1' is inserted in DO specifications by default if no BY clause is specified. Parentheses around DO specifications are removed and a special marker (X'38'), which has the same priority as a semicolon, is inserted at the end of each DO specification. A scratch text page is used as a temporary storage area when processing DO specifications.

The processing described above is illustrated in the following example. The source statement:

```
PUT EDIT (P,Q,((AR(I,J)DO I=1 TO 20) DO J=3 TO 4)) ((X)A,3B,(3*X)(4 B,A),A));
```

would appear in the output from Phase EI as follows:

```
PUT-header EDIT(P,Q, DO J=3 TO 4 BY 1 '38' DO I=1 TO 20 BY 1 '38'
AR(I,J) ENDDO;ENDDO;)
FMATLST(FIT(X)(fmtA, FIT 3 fmtB FITE FIT(3*X)(FIT 4 fmtB FITE fmtA)
FITE, fmtA)FITE) FILE(SYSOUT);
FILE(SYSOUT);
```

THE DICTIONARY BUILD STAGE

The dictionary build stage consists of four phases: GA, GI, GE, and GM. Each phase is invoked for all compilations. These phases build a dictionary in which entries describe all identifiers and some constants that appear in the text streams output from the syntax analysis stage. The purpose of the dictionary is to provide easy access to any of the information that is currently known about a particular identifier at the time of its appearance in any data area. Later phases of the compiler may use the dictionary to obtain this information, or to insert information that is determined during processing.

The dictionary is built by extraction of information during several passes of the text streams against the phases that comprise this stage. In the final pass, each item for which a dictionary entry (or entries) has been made is replaced by a reference to its dictionary entry. This reference is accompanied by a brief description of the item's most important features, thus reducing the need to access the dictionary for regularly used information.

DICTIONARY SECTIONS

The dictionary is divided into four named sections, the names, variables, general, and storage dictionaries. The names dictionary is built completely in this stage. The main structures of the variables and general dictionaries are built in this stage, but many additional entries may be made by later phases. The storage dictionary is built in the storage allocation stage.

Note: The formats of the various dictionary sections, and of the entries they hold, are shown at section 5: "Data Area Layouts."

The Names Dictionary

An entry is made in the names dictionary for each unique identifier in the text, including compiler-generated names for BEGIN-blocks. The entries have fixed alignment and variable length, (see figure 5.6).

Each entry in the names dictionary contains a reference to its associated entry in either the variables or general dictionary. The level and count numbers of blocks containing the identifier are also included. Entries for structure members contain references to the name of the immediately containing structure. All entries are connected by hash chains.

Searching the Names Dictionary (Hashing): Each time an identifier is processed by a phase in this stage, the names dictionary must be searched to find whether an entry for that identifier already exists. Because an identifier may appear many times in a source program, a hashing technique is used to avoid repeated sequential searches of the dictionary.

When a name in the text is read, an Exclusive-Or operation is performed, byte for byte, on that name. After each exclusive-or operation, the two hexadecimal digits in the hash-result-byte are interchanged. The value of the final hash-result-byte is doubled. The resulting value indicates an offset in a 512-byte hash table. This table consists of a number of 2-byte dictionary references, each of which is the head of a hash chain

linking existing entries. The chain referenced by the hash result is searched for an existing entry for the name. If there is no entry, the name is added to the chain.

The scope of declarations in PL/I allows identical names to refer to different identifiers. Names with the higher block counts are arranged nearest to the head of each chain, so that the first name found, if it is known in the current block, will be the identifier required.

A structure member can be referred to by various qualified names. In checking the identity of a qualified name, the right-most name in the qualifications is searched for. If found, the containing-structure name referred to in its entry is compared with the next left qualification name. This process is repeated so that each member of the qualified name and the structure are compared. If any structure name does not agree with the corresponding qualifying name, the containing structure name is then compared with the same qualification. If the structure and the qualifications end simultaneously, the name has been found. If the structure ends before the qualifications end, the name found is not the required name. If the qualifications end before the structure ends, the qualification is incomplete (but may be valid), and the hash-chain search is continued. If no completely qualified reference is found, and if there is more than one incomplete but valid reference, these references are ambiguous and an error message is generated.

The Variables Dictionary

An entry is built in the variables dictionary for each variable in the text. Each entry contains code indicating the attributes of the variable, and a reference to the corresponding entry in the names dictionary.

All variables dictionary entries are of fixed length (40 bytes), and all entries have the same format. The usage of some of the fields in these entries varies at different stages of the compilation (see figure 5.7).

Entries for variables aggregates contain references to corresponding entries in the general dictionary. Entries for structure members are contiguous in order of declaration (after LIKE has been expanded). Each structure member is allocated a sequence number (one for a major structure, zero for a non-structure member). Each entry for a structure member contains the sequence number of its containing structure.

The General Dictionary

The general dictionary is used throughout the compilation to store information about a wide variety of items. Entries in this dictionary section are of variable length but have a fixed alignment.

Phases in the dictionary build stage make entries in the general dictionary for the following items:

Label Constants: All entries for label constants are grouped at the beginning of the dictionary. This enables their references to be used as offsets in bit vectors that are used for flow tracing in the optimization stage. The entries for label constants are not made during the first pass, but space is reserved for them according to a value set in the XNLAB field in XCOMM during the syntax analysis stage.

Label lists: Each label list entry contains a bit vector indicating the label constant values that can be assigned to a label variable. Label list entries are referenced by label variable entries in the variables dictionary.

Block Headers: Entries for block headers follow the space reserved for label constants. An entry is made for each block in the source program.

Entry Constants: Entry constants that appear as prefixes on PROCEDURE or ENTRY statements, or are generated by the compiler for BEGIN blocks, have entries which contain a reference to their associated entry in the names dictionary, a list of the attributes applicable to the RETURNS attribute, and a variable-length list of the references of variables dictionary entries for any parameters used on the entry point. External entry constants have similar entries but, instead of having a list of parameter dictionary entries, they contain the head of a chain linking all relevant parameter-descriptor entries in the general dictionary.

Parameter Descriptors: Entries for parameter-descriptors are built from entry declarations. They are similar to variables dictionary entries (without names dictionary references). If the descriptor is omitted, a dummy entry is created. The descriptors are chained via their names dictionary reference fields.

Generic Entry Points: An entry for a generic entry-point name contains references to every entry for the generic family, and also to entries for the associated WHEN arguments.

Constants: Integer constants of value less than 10**7 are held in text. Dictionary entries are made for all other constants, each entry containing the DED and the internal representation of the constant.

File Constants: An entry for a file constant contains the file attributes, a reference to an entry for any ENVIRONMENT attributes declared, and a reference to the associated names dictionary entry.

ENVIRONMENT Attributes: Files declared with the ENVIRONMENT attribute have an associated entry containing the ENVIRONMENT options. Names and constants are replaced by 6-byte text references.

Aggregate Tables: Entries for structures and arrays contain details of the structuring or bounds.

PICTURE Specifications: An entry is made for each different PICTURE specification in the text. Each entry contains the picture specification (checked for errors) and the DED implied by the picture.

DEFAULT Specifications: An entry is made for each different DEFAULT specification appearing in the text. The format of these entries is similar to that for entries in the variables dictionary.

When implicit declarations are processed, the appropriate entry is selected and copied into the variables dictionary. When contextual declarations are processed, dictionary entries are made by merging the contextual attributes with information in the appropriate DEFAULT specification entry.

Overflow Entries: When the fixed format of dictionary entries cannot accommodate information required in unusual situations, additional entries known as overflow entries are built. The information in such entries depends upon the source of reference.

EXPLICIT DECLARATIONS (PHASE GA)

Phase GA performs the following three main functions:

1. Builds dictionary entries for all explicitly declared items, except label constants. All relevant valid attributes, whether they are applied by explicit declaration or by default, are indicated in the entries. The following attributes, which have appended information, are not processed completely:

   a. string lengths that are expressions

   b. dimensions attributes

   c. label-variable value lists

   d. INITIAL attribute

   e. DEFINED attribute

   f. BASED attribute

   g. OFFSET attribute

   References to these attributes are placed in the appropriate variables dictionary entries so that they can be easily found by later phases.

2. Sets up DEFAULT statement information for use by the phases that process contextual declarations (Phase GI) and implicit declarations (Phase GM). As this phase is not aware of the names of items that are declared implicitly, sample implicit-declaration entries are built in the general dictionary, one for each different default specification. A directory is built to assist later phases in the stage to access these entries.

3. Diagnoses multiple declarations, conflicting attributes, and unresolvable LIKE attributes.

PHASE INPUT

Phase GA scans the dictionary-text stream output from the syntax analysis stage. This file contains PROCEDURE, ENTRY, BEGIN, DECLARE, DEFAULT, LOCATE, and ALLOCATE statements. The main text stream is not used by this phase.

PHASE OUTPUT

Output from the phase consists of:

1. The main text stream and the dictionary-text stream, neither of which are changed by this phase.

2. Names dictionary entries for all identifiers that are declared explicitly, except label constants.

3. Variables dictionary entries for all variables that are declared explicitly.

4. General dictionary entries for block headers, entry points, generic entry points, parameter descriptors, file constants, environment attributes, picture specifications, and all default specifications.

5. A scratch page containing the hash table and a directory to the default specification entries in the general dictionary.

PHASE OPERATION

The dictionary-text stream input is scanned, block by block, in order of appearance in the block-header chain. Within each block, the chains set up by Phase EE are used to scan and process items in the order of DEFAULT statements, PROCEDURE, ENTRY, and BEGIN statements, and DECLARE statements. ALLOCATE and LOCATE statements are not processed by this phase.

Use of Tables, Lists, and Directories

To assist in the correct application of attributes to identifiers, Phase GA creates and/or uses the following tables, lists and directories:

The Default Directory

The LIKE Directory

The Structure Table

The Factor-level Stack

The Attribute Collection List

The Attribute Tree

Each of these items exists only within the phase.

THE DEFAULT DIRECTORY: Because DEFAULT statement specifications can be applied to any declaration, a directory, which contains the text reference of the attributes associated with any DEFAULT RANGE specification, is built.

The default directory is divided into sections corresponding to block levels. When the directory is used by routines within the phase, it is scanned in the opposite direction to that in which it is built, so that entries for the innermost level of known blocks are scanned first.

At the end of processing by Phase GA, and when entries for sample implicit declarations have been built in the general dictionary, a new default directory is built for use by other phases. This new directory which is part of the phase output, provides direction to entries in the general dictionary instead of to items in the text.

THE LIKE DIRECTORY: When a structure or structure member with the LIKE attribute is detected, information for resolving this attribute may not be available. To assist in the application of this attribute when all other declarations in the block have been seen, a LIKE directory, similar in organization to the DEFAULT directory, is built.

Entries in this directory contain the statement number and the factor level at which the attribute appeared, and the text references of the start of the major structure declaration.

THE STRUCTURE TABLE: This table is built for the following purposes:

1. To enable the true level of a structure member (as opposed to the declared level) to be determined.

2. To enable the sequence number of the containing structure to be determined.

3. To enable the containing-structure chain in the names dictionary to be set up.

4. To enable the inheritance of alignment declarations.

When a structure is found in a declaration, an entry is made in this table for each true structure level (in level order). Each entry contains the declared level, the true level, the sequence number, the name reference, and declared alignment of the last structure member, at that level, that was processed.

When a structure member is processed, the table is scanned from the deeper levels upwards until an entry is found with a declared level deeper than that of the current structure member. The true level for the current member is one more than that in the entry found. Thus, a structure declared as follows:

```
DCL 1A, 3B, 5C, 4D, 2E;
```

acquires true levels as if it was declared as follows:

```
DCL 1A, 2B, 3C, 3D, 2E;
```

THE FACTOR-LEVEL STACK: This stack is used to enable efficient scanning of declarations containing factored attributes. The stack is built at the same time as the attribute collection list described below.

When a DECLARE statement is scanned, a stack entry is made for each depth of factoring (up to a maximum of 16). Each entry is six bytes long. The first byte indicates the structure level applicable at that factor level, and avoids the necessity for a backwards scan to determine the inheritance of a structure level from a containing factor level. The other five bytes contain the text reference of the end of the attribute list applicable to the factor level. This information enables repeated scanning of attributes at outer levels of factoring to be avoided.

THE ATTRIBUTE COLLECTION LIST: When an explicit declaration of an identifier is detected, a list is made of all its explicitly declared attributes. Storage is allocated so that entries can be made, at predetermined offsets, for every attribute in the PL/I language, plus some pseudo attributes (e.g., I-N,ARITHMETIC) used for processing purposes. The compiler internal code for each attribute indicates the offset in the list at which a relevant entry is to be made.

Each entry is two bytes long. The first byte contains a factor-depth value, representing one more than the number of factor levels between the attribute and the identifier to which it is applied. This value is updated as commas and parentheses are detected in the statement.

The second byte is a flag byte, used to give the following indications:

1. The attribute is derived from a default specification.

2. The attribute is implied (e.g., the ENTRY attribute is implied in the statement: DECLARE X RETURNS (FIXED);).

3. The attribute is to be applied. This indication is set according to the result of checking all the declared attributes in the list against the attribute tree described below.

THE ATTRIBUTE TREE: The primary purpose of the attribute tree is to provide a guide to the selection of default attributes (from DEFAULT statement or standard default specification) when the attributes list in an explicit declaration is incomplete (e.g., the FLOAT attribute is

applied to the statement: DCL X DEC;). The tree is also used to detect conflict between declared attributes (e.g., DCL X FIXED FLOAT;). The 'tree' consists of a list of 2-byte entries. In each entry, the first byte contains a code for a particular attribute. The second byte either indicates the offset in the list of another 'branch' to be examined if the attribute is selected for application, or is null, indicating the end of a 'branch'. Each 'branch' can usually be associated with a particular class of attributes, such as storage class or data type.

When the attribute collection list entries for a declaration have been made, an interpreting routine accesses the relevant entries in the attribute tree. By following the branches indicated, the compatibility of attributes is checked and the 'attribute is to be applied' flags in the attribute collection list are set. If attributes are found to conflict, the appropriate attributes are not applied and diagnostic messages are issued. Branches containing attributes required by DEFAULT specification are then scanned, and attributes applied by selection. If no suitable DEFAULT specification exists, the standard default attribute is applied.

Sequence of Processing

The dictionary-text stream is scanned, block by block. Within each block, statement chains are scanned in the order of DEFAULT statements, PROCEDURE and ENTRY (and BEGIN) statements, and DECLARE statements. The ALLOCATE statement chain is not scanned by this phase.

DEFAULT Statements: DEFAULT statement specifications can apply to many declarations. For this reason, the DEFAULT statement chain is scanned, and the default directory is built, at the start of processing of each block.

Entry Points: In order that names on entry points internal to a block can be associated with any DECLARE statement references to them, statements in the PROCEDURE/ENTRY statement chain are scanned next. Dictionary entries are made for items in this chain as follows:

Names dictionary entries are made as required.

Entry-point entries are made in the general dictionary for each statement in the chain.

Block-header entries are made in the general dictionary for all PROCEDURE, BEGIN, and ON-BEGIN statements in the chain.

Parameters associated with statements in the chain are detected, even though the parameters are not in the same block. Names dictionary entries only are made for the parameters at this stage.

If there is a RETURNS option on a statement, entries are made for the attributes in the attributes collection list. Default attributes are added and the attributes are then selected and applied to dictionary entries.

DECLARE Statements: Statements in the DECLARE chain are processed in sequence, although factoring of attributes causes some switching in the scanning sequence. The factor-level stack is used to avoid unnecessary rescanning of factored attributes.

When a name is found, its attributes are scanned and added to the attribute collection list. If there are any outer factor levels of attributes that have not yet been seen, a scan ahead is made and those attributes are collected.

To determine whether an item is a major structure, a minor structure, or a base element, the structure level of the next declared name is checked. This may be found either after the next comma, or by checking the inheritance of a factored structure level by use of the factor level table. When the declared attributes have been collected, the default directory is scanned, backwards from the last entry, for a range specification that covers the current name. When an appropriate entry is found, the referenced text item is accessed and attributes on the specification and all containing factor levels are collected. The directory scan is stopped at the first change of block following detection of an applicable range specification entry.

The names dictionary is then scanned for any previous declaration of the name. If any entry is found, a diagnostic message is generated. If no entry is found, a new entry is made and the hash chain is extended.

The attribute tree is then used to select the attributes in the attribute collection list that are to be applied. The selected attributes indicate the type of dictionary entry required. The appropriate routines are called to make and initialize the general and variables dictionary entries, and to set bits indicating the applied attributes. The size and alignment of a variable is determined, and the information entered so that Phase GE can determine whether aggregate table entries can be commoned.

Attributes Processing

Each attribute that can appear in the attribute collection list has two processing routines associated with it. The functions of the first of these routines are:

1. Marking the attribute in the attribute collection list.
2. Taking special action for any value, or other appendage, on the attribute.
3. Bumping the scan to the next attribute when processing is completed.

The other routine applies the attribute to a particular dictionary entry. Bits are set to indicate the applied attributes. In some cases, references to attributes to be applied by Phase GE are inserted.

Special features of attribute processing are described in the following paragraphs.

Precision, Scale, and Fixed String Lengths: The values of these appendages to attributes are converted to binary values by the first processing routine for the particular attribute.

The LIKE Attribute: Special problems associated with processing of the LIKE attribute are:

1. The appearance of the LIKE attribute in a structure declaration means that reference must be made to another structure declaration. The declaration for this other structure may not be seen until later in the scan.
2. Dictionary entries for members of a structure are required to be consecutive. For this reason, it is not possible to make entries for some members of a structure and later add entries for members to which the LIKE attribute applies.
3. If the LIKE attribute is applied by copying dictionary entries for previously-declared structures, the declared attributes (except

inherited alignment) will also be carried over. If the substitution is made across blocks, the default attributes that are applicable may vary. Therefore, only information about true-level structure numbers can be extracted from other dictionary entries.

To assist in handling these problems, the text reference of the relevant declaration is inserted in the dictionary entry for each major or minor structure. When a LIKE attribute is found in a declaration, an entry which contains the text reference of the major structure is made in the LIKE directory. Any completed entries for that major structure in the variables dictionary are deleted and the spaces marked as re-usable; names dictionary entries are retained.

When all declarations in a block have been processed, the LIKE directory is scanned and LIKE-structure declarations are re-accessed.

These declarations are processed in the normal way except that when a use of the LIKE attribute is seen, the name referred to is replaced by the appropriate dictionary reference, and the declaration in the text of the corresponding structure is accessed via the dictionary entry. The structure referred to in the LIKE clause is then processed and the appropriate default attributes and alignment requirements are applied.

The ENTRY Attribute: Declarations containing the ENTRY attribute present a special problem because the associated parameter descriptor list effectively contains other declarations. In particular, an entry declaration can appear within the parameter descriptor list of an entry point declaration.

To simplify problems of recursion, parameter descriptors are not processed at the time when the ENTRY declaration is seen. Instead, a dictionary entry, which refers to the parameter descriptor entries, is built and added to a chain of such entries in the general dictionary. When all declarations in the block have been processed, this chain is scanned and parameter-descriptor declarations are processed. Nested ENTRY declarations automatically cause a new chain of dictionary entries to be started. The process is repeated until no new chains are created.

The PICTURE Attribute: PICTURE specifications are checked for validity by the PICTURE attribute processing routines. Applicable precisions, scales, and string lengths are calculated and inserted in the PICTURE specification entries in the general dictionary. Identical specifications share a common dictionary entry.

Attributes Processed by Phase GE: BASED, OFFSET, GENERIC, DEFINED, and INITIAL attributes, and any variables that have subscripts containing dimensions or adjustable string lengths, are processed by Phase GE. The routines in Phase GA which select and apply these attributes insert references in the dictionary entries to enable Phase GE to access the required information in the text.

The dictionary entry for the item contains a 3-byte field which refers to the text page containing the declaration. Other fields in the entry contain the offsets, within the text page, of the declarations of the applicable attributes. If all the attributes appear in the same page as the items declarations, the text reference can be obtained from the page field and offset field in the dictionary entry. If, as a result of DEFAULT statements, the attributes appear in a page other than that containing the declaration, an overflow entry is made in the general dictionary. This entry contains the text reference of the attribute and a reference to the offset field. A flag byte is set to indicate the use of an overflow entry.

The ENVIRONMENT Attribute: The attribute lists of files declared with the ENVIRONMENT option may contain other names. When all declarations for the block have been processed, names and constants in dictionary entries associated with the ENVIRONMENT option are replaced by 6-byte text references. ENVIRONMENT option entries in the general dictionary which contain names that cannot be resolved at this time, are chained together for ease of accessing by Phase GI.

CONTEXTUAL DECLARATIONS (PHASE GI)

Phase GI detects all items that are declared by their context in the PL/I source program, and builds entries for them in appropriate sections of the dictionary.

PHASE INPUT

Input to the phase consists of the main text stream and the dictionary text stream, as output from the syntax analysis stage. Entries built by Phase GA in the names, general, and variables dictionaries are accessed as required when possible contextual declarations are found during a sequential scan of the two text streams.

The default directory built by Phase GA, which contains references to sample entries created in the general dictionary for each DEFAULT RANGE specification, is accessed as required for application to contextual declarations.

PHASE OUTPUT

The main text stream and the dictionary text stream are not changed during processing by this phase. Additional entries are made in the names, general, and variables dictionaries as follows:

General dictionary entries for contextually declared file names, label names, and programmer-defined CONDITION names.

Variables dictionary entries for items declared contextually with the TASK, EVENT, POINTER, or AREA, attributes.

Names dictionary entries for contextually-declared built-in function names, and for every item for which an entry is made in the variables or general dictionaries. An additional entry is made in the hash table for each new entry in the names dictionary. The default directory is not changed by this phase.

PHASE OPERATION

Detection of Contextual Declarations

A single sequential scan is made of the main text stream and the dictionary-text stream. The contextual declaration of an item is indicated by the presence of various PL/I keywords (e.g., TASK, BASED, SET), subscripts lists, or the locator-qualifier symbol, ->, and by the position of the identifier relative to any of these indications. For example:

```
DCL B BASED (P);
```

and

```
P-> Q = A;
```

In each of these statements, the variable P is contextually declared as a pointer variable.

The name being contextually declared may be remote from the indication of its context. For example:

```
DEFAULT RANGE (T) BASED;
.
.
.
CALL X TASK (ADDR(P-> Q)->T);
```

In the above statements, T is contextually declared as a task variable. Between the name and the keyword indicating its context are two other contextual declarations:

1. ADDR is contextually declared as a built-in function because it is followed by a subscript list.

2. P is contextually declared as a pointer variable because it is followed by the pointer symbol.

Within valid PL/I syntax, the only contextual declarations that can appear between an identifier and its context indicator are locator qualifiers and built-in function names. In these cases, no other context can appear between the indicator and the name.

In such situations, Phase GI avoids the use of read-ahead technique by maintaining a record of context in a simple 2-level stack. In the example, the context-indicator keyword ADDR when its subscript list is detected, and the context POINTER is applied to the identifier P when the -> symbol is detected. TASK is unstacked and applied to the first identifier that is not within a subscript list and is not a locator qualifier, (T in this example).

When the NAMERTN routine determines that a name is in a position that could constitute a contextual declaration, the hash chains in the names dictionary are searched for any previous explicit or contextual declarations of the name that are currently known. If such a declaration is found, the previously declared attributes are checked for compatibility with the current context. If there are any incompatibilities, diagnostic messages are generated.

If no previous declaration is found in the dictionaries, the current appearance of the name constitutes a contextual declaration, and this is processed by the appropriate routine within the phase. In most cases, the routine CONTEXT allocates the processing to a particular routine which then makes suitable dictionary entries. In the case of identifiers with the FILE attribute, the entire processing is done by the CONTEXT routine. In the case of contextually declared built-in function names, processing is done by the N13 and N13A routines which first check that the subscript list does not apply to an ENTRY statement label or a subscripted array name, and then checks against the table of built-in function names to ensure that the information in the dictionary entries is correct.

Labels: If a statement is preceded by a label or labels, this is indicated in the Type-1 text statement header. The labels are processed by the SL1 routine which makes entries for them in the general dictionary. Corresponding entries are made in the names dictionary.

Scope of Contextually Declared Names: A name declared contextually in an inner block has scope similar to that of a name declared explicitly in an outer block. When the scanning routine detects the start of a new source-program block in either the main or dictionary text streams, a list of known blocks, KNOLST, is updated. This enables the XSRCHR routine to check the scope of names when searching the hash chains.

Application of Default Attributes: Any appropriate default attributes that are specified must be applied to contextual declarations. To facilitate this, Phase GA creates, in the general dictionary, a set of sample dictionary entries appropriate to each specified DEFAULT range. Before Phase GI makes an entry in the variables dictionary for a contextual declaration, the appropriate processing routine calls the DFTRTN routine. DFTRTN scans the default directory passed by Phase GA for any sample entry in the general dictionary containing the default attributes specified for the appropriate range of variables. If such an entry is found, it is copied into the variables dictionary and the processing routine in Phase GI merely overwrites those default attributes that conflict with the context of the variable.

The default attributes specifying scope, alignment, and area length, may be dependent upon the attributes specified by context, e.g.,

```
DEFAULT (RANGE(A) INTERNAL UNAL, RANGE(B)) CHAR VALUE(AREA(7));

SIGNAL CONDITION (A1);

ALLOCATE X IN (A2) SET (A3);

SIGNAL CONDITION (B1);

ALLOCATE X SET (B3);
```

On the basis of default attributes specified by Phase GA, identifiers in the example within RANGE(A) and RANGE(B) have the attributes CHAR(1) UNALIGNED INTERNAL. By context, the following attributes are applied:

A1 is CONDITION INTERNAL

B1 is CONDITION EXTERNAL

A2 is AREA (7)

A3 is POINTER UNALIGNED

B3 is POINTER ALIGNED

In order that the correct attributes can be applied, Phase GA includes in each sample default entry information to indicate to Phase GI:

1. The area length to be applied to an area variable.
2. Whether system default alignment is to be applied.
3. Whether system default scope is to be applied.

Using this information, Phase GI sets up area lengths, and overwrites the sample attribute entries for scope and alignment as required by context.

Application of ENVIRONMENT Attributes: When the GITRT routine detects the end-of-program marker, and when dictionary entries have been made for all items declared contextually in the main and dictionary text streams, the ENVI and ENVERR routines are called if there are any identifiers in the source program declared with the ENVIRONMENT attribute. These routines scan the ENVIRONMENT chain built in the general dictionary by Phase GA. Dictionary entries are made for any contextually declared names found in the various format lists.

When all processing is completed, the GIEND and GIOUT routines ensure that all text and dictionary pages are allocated their appropriate status before calling the control phase to pass control to Phase GE.

DECLARATION_EXPRESSIONS_(PHASE_GE)

Phase GE continues the process of building dictionary entries for explicitly or contextually declared variables. New entries are built in the general dictionary, and existing entries in the variables dictionary are modified in some cases. Items processed by this phase are:

1. Variables declared with BASED, OFFSET, DEFINED, INITIAL, and GENERIC attributes.
2. Variables that appear in ALLOCATE statements.
3. Array and structures, for which aggregate-table entries are build in the general dictionary.
4. Label variables, for which value-list entries are build in the general dictionary.

This phase also builds a second text stream, known as the declaration-expressions_file. This text stream is built by the generation of statements containing information that cannot be suitably held in dictionary entries. In general, this information consists of expressions resulting from declarations, e.g., adjustable array bounds, adjustable string lengths, expressions resulting from INITIAL attribute assignments. In building statements for inclusion in this text stream, items declared with the DEFINED attribute are examined and the type of defining is identified, i.e., simple defining, iSUB defining, or string-overlay defining.

PHASE INPUT

Input to the phase consists of:

1. The variables dictionary.
2. The general dictionary.
3. The names dictionary and hash table.
4. The default directory.
5. The dictionary-temt stream.

The main text stream is not accessed by this phase.

PHASE OUTPUT

Output from the phase consists of:

1. The general dictionary, containing aggregate-table entries, label variable value-list entries, and entries for generic entry points and WHEN clauses.
2. The variables dictionary, with some entries modified.
3. The names dictionary and hash table.
4. The default directory.
5. The declaration-expressions file.

The dictionary text stream is discarded during processing by this phase. The main text stream is not altered by this phase.

PHASE OPERATION

Sequence of Processing

A single scan is made of every entry in the variables dictionary, and parts of the general dictionary, testing for entries for variables with attributes that require processing by this phase. As it is desirable that statements output in the declaration-expressions file should appear in a similar order to items in the main text stream (i.e., in deblocked order, outer block first), the scanning sequence is arranged so that entries relating to items in outer blocks are seen and processed first. According to the features of items detected in this scan, various routines are called to perform the requisite processing as described in later paragraphs.

Because the sample implicit declaration entries, created in the general dictionary by Phase GA, apply as though the declarations were made in outermost block, these entries are scanned first. The chain that links these entries also links entries for entry points that have parameter-descriptor lists, and these items are processed as they are seen in the scan. Contextual declaration entries made in the variables dictionary by Phase GI also apply as though the declarations were made in the outermost block, and these entries are scanned next. The scan then switches back to the beginning of the variables dictionary, in which entries made by Phase GA appear in the required order.

Throughout the scan, a list of known blocks is maintained and updated as changes of blocks are detected. Wherever a change of block occurs in the scan of the variables dictionary, the scan is switched to the general dictionary. All block-header entries for blocks that are known are searched for a dictionary reference, inserted by Phase GA, indicating the head of a chain of overflow entries created for items with the GENERIC attribute. Entries for all items in the chain are processed before the scan switches back to the first variables dictionary item in the next block.

On completion of the processing of variables dictionary entries, the chain of ALLOCATE and LOCATE statements in the dictionary-text stream is scanned and processed..

During the scan, dictionary entries are created or changed, and declaration-expression file statements are generated and output, as described in the following paragraphs.

Construction of Aggregate Tables

A skeleton aggregate table is built in the phase work area. When a dictionary entry for a variable that is an array or structure member is seen, information about dimensions and structure level are copied into the appropriate fields of the skeleton table.

The information about the variable is obtained by accessing the declaration in the dictionary text stream indicated in the page and offset fields in the dictionary entry. If the YV2FL flag in the dictionary entry is set, the page and offset fields refer to the relevant overflow entry in the general dictionary, and information is copied from there instead of the text.

When all available information has been copied into the skeleton table, the table is copied into the general dictionary, unless an identical table with which it can be commoned already exists in a dictionary entry. The format of aggregate table entries is shown in figure 5.11.

To assist in the commoning of tables, the following chains are maintained:

Fixed-extent structures

Fixed-extent arrays

AUTOMATIC adjustable-extent structures

AUTOMATIC adjustable-extent arrays

Aggregate tables that cannot be commoned

New chains for aggregates with adjustable extents are started on entry to each block.

Unstructured Arrays: The skeleton of the aggregate table is built up in the work area, the text information concerning the array's dimensions is accessed, and details of the lower and upper bounds of each dimension are put into the aggregate table. The phase then determines which of the five chains (see above) this aggregate table should be placed in, and scans that chain to see whether it already contains an aggregate table entry identical to the one it has just built up in workspace. If an identical entry is found, that reference is inserted into the variables dictionary entry being processed, and the table in workspace is discarded. If however, the whole of the chain is scanned without finding an identical aggregate table entry, the new table is copied into the next space in the general dictionary, and is also added to the front of the relevant chain. The reference of the new general dictionary entry is placed in the variables dictionary entry. The scan of variables entries is then resumed.

Structures: A separate aggregate table is built up for each element of the structure, to avoid the necessity of having a work area large enough to hold aggregate tables for a complete structure. A table (TAB) is maintained in the work area to indicate the general dictionary reference of the last aggregate table inserted (or commoned) at a given level, and also the total number of dimensions in the structure up to that level, including any inherited from higher logical levels. Text information concerning dimensions is processed as for arrays, and the bounds are inserted in the work-area table with an indicator to show the level from which each dimension was inherited. For major structures, however, only the basic part of the aggregate table is copied into the general dictionary (or compared, if commoning). When a base element is encountered, any bounds information is added to the work-area table, and the entire table, showing all bounds contained in and inherited by the base element, is copied into the general dictionary. When the last member of a structure has been added to the dictionary (or commoned), the information on inherited dimensions is cleared from the table (TAB).

When the aggregate table for a major structure is ready for copying into the dictionary, the appropriate chain is scanned for an identical major structure aggregate table entry. If such a table is found, it is assumed that subsequent elements of the structure currently being processed will also common. The aggregate tables for subsequent elements are built up in the usual way, and each is compared with its corresponding element in the structure against which commoning is being attempted. If the comparison fails, the chain scan is resumed for any other major structure against which commoning may be possible. If such a structure is found, its subsequent elements are each compared with the corresponding element in the structure against which commoning has failed, and if the comparisons are all successful, then the element in the work area is compared with the next uncommoned element. If this is

successful, TAB and all chaining fields are updated, the variables dictionary entries are reset to point to the newly commoned structure, and processing goes on to the next variables dictionary entry.

If the comparisons fail, and the scan of the chain ends without any further commonable major structure being found, or if the structure against which commoning is being attempted finishes before all elements of the current structure have been commoned, then it is necessary to create an entirely new series of entries in the general dictionary for the current structure. This is done by copying, into new spaces in the dictionary, all of the previously commoned elements of the structure against which commoning has now failed, updating the internal chaining as necessary. When all such elements have been copied, the aggregate table in the work area is moved into the next dictionary space and the new structure is added to the appropriate chain.

Building the Declaration-expressions File

The term 'declaration expression' is applied to any expression that appears in an explicit or contextual declaration, and to expressions generated by the compiler to assign the required value to a variable declared with the INITIAL attribute.

If a variable is associated with a declaration expression, it is not possible to build a dictionary entry describing that variable until the expression has been analyzed and evaluated. The information required to do this may not be available until later in the compilation process, or until execution time. Phase GE therefore generates special Type-1 text statements to contain and identify these expressions, and passes them to later phases of the compiler in a separate text stream. The previous second text stream, known as the dictionary-text stream, is discarded after processing by this phase. The new second text stream is referred to as the declaration-expressions file.

The types of declaration expression for which statements are generated are as follows:

Array bound expressions

Adjustable string length expressions

Locator qualifier expressions

DEFINED item expressions

INITIAL value assignment expressions

The scanning sequence is arranged so that items are output in the declaration expressions file in deblocked order, outer block first. The exceptions to this sequence are array-bound expressions, for which statements may be commoned with those for earlier identical expressions, and expressions arising from ALLOCATE statements, which all appear at the end of the file. All expressions corresponding to a single aggregate table appear in contiguous statements, even though they may refer to more than one variable.

In general, the expression is analyzed and a routine appropriate to the type of expression is called. This routine generates a Type-1 text statement which contains the expression, and identifies its type by inserting a compiler-generated verb in the statement header. The following types of statement are generated:

Identifying Name	Expression Type
AGGASSN	Array bound assignment.
STRL	String length value assignment.
INASSN	INITIAL value assignment.
INASSN2	INITIAL value assignment - ARRAYS.
PTS	Based pointer expression.
CFFA	AREA to which CFFSET attribute applies.
POSX	Adjustable POSITION attribute expression.
DSUBS	iSUB defining - associates base and subscripts with defined variable.
MAP	DEFINED (simple defining) - base to be evaluated during compilation.
LDASN	String overlay or simple defining - locator of defined item and address of base item to be evaluated during execution of prologue code.
PTSD	String overlay or simple defining - locator of defined item and address of base item to be evaluated during execution of main program code.

Processing Array-bounds Expressions

Each expression that is used to define the bounds of an array is given a unique identifying number. When the work-area aggregate table for that array is tested for commoning with an aggregate table entry in the general dictionary, the expression is also tested for commoning. If the test succeeds, the reference of the common aggregate table entry and the number of the common expression are inserted in the variables dictionary entry for the variable. If the commoning tests fail, a new aggregate table entry is made in the general dictionary and a new declaration expression file statement is generated.

Processing Variables with the DEFINED Attribute

When a variable with the DEFINED attribute is scanned, a dictionary entry for the base item name is searched for. If an entry for an explicit or contextual declaration of that name cannot be found, an implicit declaration entry is made by copying the default attributes from the appropriate sample implicit declaration entry built by Phase GA. The reference of the dictionary entry for the base item is copied into the variables dictionary entry for the defined item.

Routines are then called to analyze the declaration and determine whether it applies to simple defining, string overlay defining, or iSUB defining. For each type of defining, another routine is called to determine whether the address of the base item can be evaluated by later phases during compilation, during execution of the object module prologue code (e.g., AUTOMATIC adjustable extent expressions), or during execution of the object module main text code (e.g., CONTROLLED or adjustable extent subscripts). The routine then generates a declaration-expression-file statement by inserting values into the

Statements in the ALLOCATE/LOCATE statement chains in the dictionary text stream are scanned, block by block. Within each statement, each allocated or located name is replaced by a 6-byte descriptor containing

fields of an appropriate skeleton statement in the phase work area, and copies the statement to an output text page.

Processing Variables with the GENERIC Attribute

When Phase GA processes DECLARE statements, it creates overflow entries which each contain the text reference of a variable declared with the GENERIC attribute. The entries for each item within a block are chained, and the head of the chain is stored in the YHNXB field of the appropriate block header entry in the general dictionary. This enables Phase GE to check the scope of names when processing GENERIC items.

Phase GE scans the chain of overflow entries for each block. For each entry, the contained text reference is used to access the declaration for the generic name in the dictionary text stream. The names dictionary is then searched for an entry for that name, and the dictionary reference is inserted in the overflow entry.

The scan of the declaration is continued and an overflow entry is made for each WHEN clause. Each entry contains the dictionary reference of the item named in the WHEN clause, plus the attributes specified. A forward chain connects all WHEN clause entries for a generic name. Thus, two WHEN clause entries, one for E1 and one for E2, would be made for the following declaration:

```
DCL G GENERIC (E1 WHEN (FLOAT,COMPLEX),
               E2 WHEN (PICTURE '99V9'));
```

Masks are built and used to check that the attributes declared in each attribute list are valid within the PL/I language, and do not conflict with each other. If a WHEN clause argument is a PICTURE specification, the dictionary reference of the PICTURE specification entry, built by Phase GA, is inserted in the WHEN clause entry.

Processing Label Variables

Phase GE builds a value list entry in the general dictionary for each label variable declared with an argument list. This entry contains a list of the label constant values that can be assigned to the label variable.

Because entries for label constants are all made at the beginning of the general dictionary, the value list for a label variable can be represented by a bit vector. The setting of the nth bit in a value list indicates that the label constant at the nth entry in the general dictionary can be assigned to the label variable.

The dictionary reference of the value list is inserted in the variables dictionary entry for the label.

Processing ALLOCATE and LOCATE Statements

When all declarations have been processed by Phase GE, a separate CSECT, containing routines that process ALLOCATE and LOCATE statements, is entered. These routines differ from those that process declarations, in that information is gathered in a diffferent manner, and output is in a format applicable to the generation of executable statements rather than prologue code. The main processing routines within the phase are called to generate aggregate tables, and bound, string length, and INITIAL assignment statements.

its dictionary reference. If no dictionary entry resulting from an explicit or contextual declaration can be found, the sample implicit declarations created by Phase GA are used to create implicit declaration entries.

A descriptive table for the allocated or located name is built in the phase work area, using the same format as that for a variables dictionary entry. Within this table, the declared attributes are merged with the ALLOCATE/LOCATE statement attributes.

For level-one allocated or located names, an ALLOC or LOCATE statement is generated in the declaration expression file. The format is:

```
|SN2|ALLOC|5-byte chain field|6-byte operand|
```

No prefix options are included in the statement header. They are contained in the statement in the main text stream.

All the ALLOCATE and LOCATE statements for a block in the declaration expressions file are connected by a backwards chain. As a backwards chain is used to connect the statements in the dictionary text stream, backwards chaining in the output from Phase GE enables subsequent phases to read the statement in deblocked source program order. The chain header is inserted in a overflow entry in the general dictionary, referenced from the block header entry.

The main phase routines are called to process the descriptive table for the name in the work area. If the name identifies an aggregate, an aggregate table is built and copied into the general dictionary. No attempt is made to common ALLOCATE or LOCATE statement aggregate tables. Declaration expression file statements, indicating evaluation when the code generated for the ALLOCATE or LOCATE statement is executed, are generated and output. Control is then returned to the special ALLOCATE/LOCATE statement routines.

The declared extents are then compared with the allocated or located extents. To enable optimization of subscript calculations, etc., only those extents that change on allocation or location are marked as being adjustable. The extents of external items or CONTROLLED items that are passed as arguments, are always marked as adjustable.

On completion of this processing, control is passed to Phase GM.

IMPLICIT DECLARATIONS AND NAMES RESOLUTION (PHASE GM)

Phase GM completes the building of dictionary entries for items declared in the source program. Items for which dictionary entries are made or modified by this phase include:

1. Implicitly declared names (i.e., names for which no explicit or contextual declaration has been found).
2. Constants that cannot be held in text in their literal form.
3. Attributes of files specified in OPEN statements.
4. PICTURE items in format lists.
5. Explicitly declared non-contextual built-in functions.

Each identifier in the main text stream and the declaration expressions file is replaced by a 6-byte field that contains a brief description of the item and the reference of its main dictionary entry.

Some statements in the declaration expressions file are modified or merged into the main text stream at appropriate places.

PHASE INPUT

Input to Phase GM consists of:

1. The main text stream as output from the syntax analysis stage.
2. The declaration-expressions file, output from Phase GB
3. The names, variables, and general dictionaries.
4. The hash table and default directory.

PHASE OUTPUT

Output from Phase GM consists of:

1. The main text stream, consisting of statements in Type-1 text format, in which the following changes have been made:
   a. Each name (or constant for which a dictionary entry has been made) is replaced by a 6-byte field containing a brief description of the item and the reference of the main dictionary entry for the item.
   b. Constants for which no dictionary entry has been made are replaced by a 6-byte field containing a description of the item (i.e., data type, scale, precision, length, etc.) and a binary representation of the constant value.
   c. For some structure references, a structure element descriptor (STRUD) for each base element is inserted in the text.
   d. Each array item is followed by an aggregate table reference (ATR).
   e. Each item in an argument list is preceded by a descriptor for its corresponding parameter (PARD).

2. The names, general and variables dictionaries, in which entries have been made for implicitly declared variables, SELECT statement constants, and for some constants.

3. The declaration-expressions file, in which some statements have been modified or replaced, and from which statements arising from ALLOCATE or LOCATE statements have been removed and merged into the main text stream.

PHASE OPERATION

Sequence of Processing

The main text stream is scanned sequentially. As each item is identified, processing action is taken as described in the following paragraphs. Processing action includes the building of a new main text stream. Each input text page is discarded when processing of its contents is completed.

When processing of the main text stream is completed, the declaration-expressions file (if it exists) is scanned sequentially. Items for which action is required are processed as they are detected, and the modified version of the file is built on new text pages.

Implicit Declarations

When a name is found in the text, the XSRCHR routine is used to scan the dictionary for an entry for that name. As dictionary entries have been made for all items declared explicitly or contextually, any name that has no dictionary entry at this stage is considered to be declared implicitly. The default directory is used to find a sample implicit declaration entry (built in the general dictionary by Phase GA) that is applicable to the item. The sample entry is then copied into the variables dictionary. An entry is made in the names dictionary and added to the hash chain.

If the implicitly declared name is associated with a declaration-expression in the second text stream, the entry made in the variables dictionary is added to a chain. This chain links all copies of similar implicit-declaration entries to the sample entry in the general dictionary from which they were copied, and provides easy access for possible later processing.

When dictionary entries for an implicitly declared name have been made, the name is processed as though existing entries had been found by the XSRCHR routine.

Resolution of Names

When a name is found during the scan of the text, the XSRCHR macro routine is used to find the relevant entries in the names and variables dictionaries. The reference of these entries are inserted in the last two bytes of a 6-byte field that is used to replace the name at every place where it appears in the text. The first byte of this field contains a code byte which identifies the type of data represented by the name (see figures 5.29 to 5.38). Information inserted in the remaining bytes of the 6-byte reference varies according to the type of item it replaces. (The formats of various 6-byte references are shown

at section 5: "Data Area Layouts.") The 6-byte references for some types of data item include a 3-byte data-element-descriptor (DED), the formats of which are shown in figure 5.59. Thus, for many items, the 6-byte reference contains sufficient information to enable rapid scanning of the text without accessing the dictionary entries for every identifier. A text code, DREF, is placed in front of every 6-byte reference, to indicate the presence of the reference during text scanning.

For some items, additional information is inserted in the text after the 6-byte reference. Such items are dealt with as follows:

1. Array items - a reference (ATR) is inserted indicating the appropriate aggregate table entry in the general dictionary. The ATR is preceded by a marker (ATRM).

2. Structure items - a structure descriptor (STRUD) is inserted after most structure references. The STRUD contains a 5-byte descriptor for each base element. The beginning and end of the STRUD are indicated by 1-byte markers. Structure descriptors are not inserted if the structure appears in a check list, a data list, a FROM or INTO clause, or as an argument to a built-in function.

3. Subroutine calls and function references -- a 2-byte field containing the number of items in the argument list is inserted after the 6-byte reference. Except where the statement calls an external procedure for which no ENTRY declaration has been made, a reference to the appropriate parameter-descriptor entry in the general dictionary is inserted in front of each item in the argument list. Each parameter descriptor reference is preceded by a PARD code byte.

In order to reduce the number of dictionary-page input/output operations involved in name-resolution processing, a push-down stack (NAMSTK) is built and maintained in phase working storage. Each entry in NAMSTK is 32 bytes long, and can contain a name of up to 27 characters preceded by a two-byte field containing its length, and also the reference of the corresponding variables-dictionary entry. The number of entries in NAMSTK is indicated in NAMSTL.

Entries are made in NAMSTK for unqualified variables which have names of not more than 27 characters; for qualified names the limitation depends upon the number of members in the name. In addition to the overall name-length value inserted in a two-byte field preceding the name in the stack entry, each member of the name is preceded by a one-byte field containing the member-name length. Thus an entry for a name "A.B" will contain "0003 01 A . 01 B". Where subscripts are involved, an entry is made in NAMSTK only if the last member of a qualified name is subscripted (for example, "A.B(5)"); no entry is made if subscripts appear within the name (for example, "A(5).B").

When a name is seen during the text scan, NAMSTK is searched from the most recent entry. If an entry for the name is found, processing is performed without searching the names dictionary. If there is no entry for the name in NAMSTK, XSRCHR is used to search the names dictionary, information is inserted in the text, and a new entry is made in NAMSTK. Only the 64 most-recently-used names are held in the stack.

Resolution of Constants

When a constant item is found in the text, processing varies according to the value of the constant.

Decimal integer constants with value less than 10**7, binary integer constants with value less than 2**32-1, character constants of less than

four characters, and bit constants of less then 32 bits, do not have dictionary entries made. These items are converted to the required data form (binary or string) and held in text as a 6-byte field which also contains the original precision or length of the constant. The dictionary code byte for such items is null.

A chain of general dictionary entries is created to enable Phase ID to perform optimization of SELECT statement constructs more easily. The entries are known as Select Optimization Tables (SOTs) and are chained in SELECT statement number order from an anchor in XCOM. A SOT is created each time a SELECT statement is found in the text, and a stack of information required in the SOT is maintained. An entry is made in the stack for each level of SELECT construct. A new entry is created in the stack (SELECT stack or SELSTACK) when a SELECT statement header is found in the text and the appropriate entry is updated when a SELECT expression, WHEN expression, or OTHERWISE statement header is located. At an ENDSELECT statement the SOT created for the corresponding SELECT statement is updated with the contents of the entry at the top of the SELSTACK and a new entry created in the stack.

Dictionary entries are made for constants with values too large to be held in text. The precision and scale of the constant is inserted in the 6-byte reference, together with the dictionary reference.

Argument Lists and Subscripts

When a reference to a programmer-defined function is processed, arguments have to be distinguished from subscripts so that any parameter descriptors can be correctly positioned before items in the argument list.

A scan-ahead technique is used whenever a function reference is seen. Subscripts are distinguished from arguments by comparing the number of subscripts with the number of declared dimensions. Any outstanding parenthesized items are assumed to be arguments. The problem is illustrated in the following example:

```
DCL A W(10),2 X(3,7),3 Y(10),4 Z ENTRY;

CALL W(5).X(2).Y(3,5).Z(P,Q);
```

In this case, the function name W.X.Y.Z is resolved as soon as W is seen. A scan-ahead is then made of the whole function reference, ignoring the qualifications ".X", ".Y", and ".Z". The subscripts are compared with the number of subscripts in the declaration, and merged into the subscript list (5, 2, 3, 5). The item (P,Q) is thus recognized as an argument list.

Because subscript lists and argument lists can contain nested subscripts or arguments, a stack is set up whenever a function reference is detected. An entry is made in the stack for each argument level. Each entry contains details of the parenthesis level (to distinguish intervening subscript levels), the number of subscript lists at this level preceding the argument, and details of where the parameter descriptor for the next argument can be found (parameter descriptors for internal function references are obtained from the parameter text references, and for external references are obtained from the dictionary entry).

PTSD Statements: These statements contain information about defined items that cannot be addressed until reference at execution time. They are changed to PTS statements with the format:

```
PTS header,   ADDR(Base item);
```

LDASN Statements: These statements set up locators for defined items that cannot be addressed until execution of prologue code. They are modified to the format:

```
LDASN-header, Lccator = ADDR(Base item);
```

On completion of processing, control is passed to Phase IA.

Built-in Function Declarations

Built-in functions can be declared in a number of ways. If a function is explicitly declared with the BUILTIN attribute, Phase GA builds a dictionary entry with an incomplete description. If a built-in function is declared contextually, Phase GI either completes an entry built by GA, or builds a complete dictionary entry if there has been no explicit declaration.

The PL/I language allows a limited number of built-in functions to be declared explicitly but without a contextual declaration, or for the BUILTIN attribute to be applied by use of a DEFAULT statement without either an explicit or contextual declaration. Phase GM contains a list of such functions, together with the attributes that must be applied. When one of these function names is detected, the attributes are applied to the incomplete dictionary entry (if one exists) or used to build a new dictionary entry. The name is then resolved in the text.

Merging ALLOCATE and LOCATE Statements

Phase GE generates statements in the declaration expressions file that are associated with ALLOCATE or LOCATE statements in the main text stream. Phase GE also builds a chain of general dictionary overflow entries which contain the text references of the relevant statements in the declaration expressions file.

When Phase GM scans an ALLOCATE or LOCATE statement in the main text stream, it uses the chain of entries in the general dictionary to access the statements in the main text stream, immediately after the ALLOCATE or LOCATE statement. As the statements are copied, implicit declaration entries are built, and names are resolved, as for all other statements.

Processing Declaration-expressions Statements

When the scan of the main text stream is completed, the declaration expressions file is scanned. Names are resolved, and dictionary entries are made for implicit declarations, in a similar manner to processing of the main text stream.

The special processing of statements performed by Phase GM depends upon whether the information they contain is to be evaluated and used during compilation, during execution of prologue code, or on reference to the item during execution of the compiled code. All declaration expressions file statements that contain information about a particular item are contiguous.

Where such statements contain information that may be required as reference during execution, the dictionary entry for the item is made to point at the first of the statements. This enables all such statements relating to a particular item to be readily accessed during compilation. Particular types of statements are processed as follows:

AGGASSN and STRL Statements: These statements contain information about adjustable extent items that will require mapping in the prologue code. A MAP statement is generated to precede the first statement for each item.

PTS and OFFA Statements: If the information contained in these statements consists of a single unsubscripted unqualified name, the dictionary reference of that name is inserted in the dictionary entry for the variable to which the statement applies. The statement is then deleted.

The compilation process can be considered to consist of two main operations. The first of these operations consists of checking the validity of the PL/I source program, and organizing the internal representation of the source program into readily accessible and processable data forms, i.e., the main text stream and the dictionary. The second operation consists of determining the object code that is required to represent the source program for execution purposes, and generation of that code in the form of a relocatable object module.

When the compilation process is considered in this way, the four phases that are grouped in the expression analysis and text formatting stage, (Phases IA, ID, IE, and II), perform functions which continue and complete the first operation. These functions include:

- Merging of the text into one text stream.
- Expansion of data aggregates into individual elements.
- Matching of corresponding arguments and parameters, including creation of dummy arguments if required.
- Resolution of expressions and determination of any data-type conversions that are required.

The last phase in the stage, Phase II, changes the organization of the text stream from the format used during analysis of the source-program content (Type-1 text) into a format more convenient for indicating the object code required (Type-2 text).

MERGING OF DECLARATION EXPRESSIONS (PHASE IA)

The prime function of this phase is the repositioning of declaration expressions and MAP operators, passed to the phase in the declaration expressions file, at appropriate places in the main stream. While performing this function, the phase also performs the function listed below:

1. Constructs locator chains for all references to based variables.
2. Detects all references to DEFINED items, and introduces information about the defined base into the text.
3. Analyzes the extent to which storage for REFER structures will require remapping during execution, and generates extent-expressions and MAP operators in the correct sequence.
4. Analyzes all array INITIAL assignments.
5. Performs special processing on references to certain built-in functions and pseudovariables.
6. Processes ALLOCATE and FREE statements.

PHASE INPUT

Input to the phase consists of the main text stream and the declaration-expressions file, both in Type-1 text format.

The variables and general dictionaries are accessed.

PHASE OUTPUT

Output from the phase consists of the modified main text stream only: the declaration expressions file is discarded when the information it contains has been merged into the main text stream. No dictionary entries are created by this phase.

Although the main text stream output from this phase consists mainly of statements in Type-1 text format, the text generated when some statements are merged from the declaration expressions file differs from this format. Phase IA generates some text in pseudo-text-table format, where each pseudo text table is 20 bytes long and consists of a 2-byte operator and three 6-byte operands. The term "pseudo text table" is used because of the close similarity between this format and the format of the text tables into which all text is translated by Phase II. The text generated by Phase IA may also contain compiler-generated temporary operands, and a particular type of temporary operand known as qualified-name temporary operands (referred to in the published listings and throughout this manual as Q-temps). An example of the use of these text features is the text generated by this phase to represent the source statement:

```
DECLARE A(M);
```

This would appear in the text output from Phase IA as:

```
Statement    OFFS       8          A      Q-temp.n Q-temp.n ASSN M;
  header  (2 bytes) (6 bytes) (6 bytes) (6 bytes)

                      Pseudo text table              Type-1 text
```

where the function of the statement is to set the value of the upper bound of the array A (as represented by Q-temp.n), to the value M. The constant value "8" indicates the offset of the upper-bound field from the start of the object-time descriptor for the array A.

The generation and use of Q-temps and text tables is more fully described in the description of Phase II.

PHASE OPERATION

Repositioning of Declaration Expressions

The X2STRM field in XCOMM is accessed to see if a declaration-expressions file has been created in the dictionary build stage. If so, Phase IA repositions items contained in that file at appropriate places in the main text stream. The processing required to merge this information is performed for one procedure block at a time, starting with the outermost block and repeating the processing for each block in sequence. For some items, e.g., element INITIAL assignments, merging consists of copying a particular expression from the declaration-expressions file into the appropriate place in the main text stream. For other items, e.g., locator-qualifier items, it may be necessary to reproduce an expression in several different places in the main text stream.

For each procedure block, the first step in processing is a sequential scan of the declaration-expressions file statements related to that particular block. During this scan, only those statements which contain expressions that can be evaluated during execution of prologue code are accessed and processed. Such statements are:

AGASSN statements

INASSN statements

STRL statements

MAP statements

Processing is performed so that these items (if present) are inserted in the main text stream, immediately following the relevant PROCEDURE, BEGIN, or ON-BEGIN statement, in the following order:

1. INASSN statements for fixed-length items with no dependencies (i.e., the INITIAL specification does not contain any variable which itself requires initialization).
2. AGGASSN and STRL statements.
3. MAP statements.
4. INASSN statements for adjustable-extent items with no dependencies.
5. INASSN statements for fixed-length items with dependencies.
6. INASSN statements for adjustable-extent items with dependencies.

AGASSN statements, STRL statements, and INASSN statements for fixed items are copied directly into the appropriate places in the main text

stream. MAP statements and INASSN statements for adjustable items are chained in the declaration-expressions file for later processing. This processing is performed on completion of the sequential scan for the block, when the chains of statements created in the declaration-expressions file are accessed. MAP statements are inserted in the main text stream in order of decreasing alignment requirements. An ACCUM marker is then generated to indicate that storage must be allocated for the items with adjustable extents. Any INASSN statements referring to adjustable-extent items are then inserted after the marker.

A sequential scan is then made of the main text stream statements contained in the procedure block. Processing, including the repositioning of items still remaining in the declaration-expressions file, is performed during this scan as described in the following paragraphs. The entire process is repeated for each procedure block in turn.

Construction of Locator Chains for Based Variables

Processing is performed to ensure that each based variable in the text, whether it is explicitly qualified or not, is accompanied by its relevant locator qualifier. Because locator qualifiers can themselves be based, it is sometimes necessary to construct a chain of qualifiers in order to define the correct generation of a based variable. As offset locators can be based, and can be associated with areas that are also based, more than one logical chain may be required to define a particular based variable. By use of a special stack, and by recursive calling of the text-scanning routine, chains are nested so that only one locator chain is generated for each reference to a based variable. During processing by this phase, offset qualifiers are converted into references to the POINTER built-in function, with appropriate arguments. The stack set up for the building of a locator chain is named LOCSTK. The stack is constructed and used on a last-in, first-out basis. The various types of entries that may be made in LOCSTK are listed below:

Entry type	Code byte	Length (in bytes)	Indication when unstacking
COMMA	(X'30')	1	Comma (between offset and area in reference to POINTER b.i.f.).
RPAR	(X'37')	1	Generate right parenthesis to terminate reference to POINTER b.i.f.
SCOLON	(X'34')	1	End of a section of the stack; stop unstacking and continue scanning text.
SCLN2	(X'38')	1	End of a section of the stack; stop unstacking and continue scanning text.
AREA	(X'4B')	6	Text reference of an area expression.
SN	(X'FC')	6	Text reference in main text stream.
SN2	(X'FB')	6	Text reference in declaration expressions file.
(Other)		6/4	6-byte operand if an area, 4-byte operand if a pointer or offset. (Precision and scale for a locator are not stacked.)

If a locator is found during the scan of text, a preliminary read-ahead scan is made of the locator chain, if one exists. If this scan detects an offset in a qualifying position, a reference to the POINTER built-in

function is generated in the output text stream, followed by a left parenthesis in readiness for later building of the argument list. The offset in the text is flagged to indicate that, when the main scanning routine encounters the flagged offset, the relevant area must be identified for use as the second argument in the built-in function reference. If more than one offset is found, nested built-in function references are generated. When the look-ahead scan reaches the end of the locator chain, control is returned to the main scan, which is pointing at the first locator in the chain, and the main processing of the locator chain is started.

If the main scanning routine finds an offset that was flagged during the preliminary scan, the flags are removed and entries are made in LOCSTK in the order: SCLN2, RPAR, associated area (or text reference of the area in the declaration expressions file.) The scan is then continued.

If an unqualified based variable is found, a semicolon is stacked in LOCSTK, and the dictionary entry for the based variable is accessed to determine its qualifier, which is also stacked. If the qualifier is in turn based, then its qualifier is stacked, and the process is repeated as many times as is necessary.

Detection of a non-based variable causes unstacking action down to the top SCOLON or SCLN2 in LOCSTK. Each qualifier is unstacked and copied to the output text stream, and a PTS operator is generated after each one unless it is followed by a SCLN2 entry in LOCSTK. When unstacking reaches a SCOLON or SCLN2 entry in LOCSTK, that entry is deleted and scanning of the text is restarted.

If a locator is an expression, its reference in the main text stream is stacked and the main scanning routine is called recursively to scan the expression in the declaration-expressions file. The expression is then processed, with stacking, unstacking and text generation as required. When the end of the expression is reached, control is returned to the unstacking routine, and the stacked text reference is used to position the scanning routine for resumption of its text scan.

When a saved text reference of an area is found during unstacking, it is replaced by the text reference of the current position of the scan, and the scanning routine is called recursively to scan the statement in the declaration-expressions file which contains the area expression. When the end of the expression is reached, control returns to the unstacking routine, and the stacked reference is used to position the resumed scan.

Locator chains built by the processing described are illustrated in the following examples:

1. Assuming the following declarations:

```
DCL BV BASED(OF),
    OF OFFSET(A),
    A AREA;
```

   The statements:

```
X = BV; or X = OF-> BV;
```

   become:

```
X = POINTER(OF,A)-> BV;
```

2. Assuming the following declarations:

```
DCL BV BASED(OF),
    OF OFFSET(A)BASED(P1),
    P1 POINTER BASED(P2),
    P2 POINTER,
    A AREA;
```

   The statement:

```
X = BV + 1;
```

becomes:

```
X = POINTER(P2->P1->OF,A)->BV + 1;
```

3. Assuming the following declarations:

```
DCL BV BASED(OF1(OF2->J)->OF3),
    OF1(6) OFFSET(A1(2)) BASED(P1),
    OF2 OFFSET(A2),
    OF3 OFFSET(A3),
    J BASED,
    A1(6) AREA,
    A2 AREA,
    A3 AREA BASED(P2),
    P2 POINTER BASED(P1),
    P1 POINTER;
```

The statement:

```
X = BV + 1;
```

becomes:

```
X = POINTER(POINTER(P1->OF1(POINTER(OF2,A2)->J),
     A1(2))->OF3,P1->P2->A3)->BV + 1;
```

Repositioning DEFINED Statement Information

When a reference to a defined item is detected during the scan of a procedure block, the routine SC5 is invoked to access the declaration-expressions file for a statement containing relevant information. Such information is usually held in POSX statements, which contain adjustable POSITION attribute expressions for string overlay defining, and DSUBS statements, which associate the base and subscript of an iSUB defined variable. The information contained in these statements is merged with information in the main text stream. For example, the statement:

```
DCL A(5,6), B(4) DEFINED A(2*1SUB,1SUB);
```

appears in the main text stream output from Phase IA as:

```
B DSUBS A(2*1SUB,1SUB) RPAR2
```

where the DSUBS and RPAR2 operators act as parentheses around a special subscript list.

Generation of Structure-mapping Information

Based structures with adjustable extents (i.e., REFER structures) may require mapping of storage at the time they are referred to during execution of the statement code. Mapping cannot be completed prior to this stage, because the mapping depends upon which generation of the structure is being used by the REFER item. Routine HOWFA examines the position of the referenced structure-item in relation to adjustable items in the structure. It also examines the alignment requirements of various elements of the structure, and thus determines how far a structure must be mapped for any reference at execution time.

The adjustable extents (string lengths or array bounds) of a REFER structure are contained in a preceding unsubscripted base element of the same structure. Therefore not all references to elements of the structure result in a requirement for remapping of the structure. Requirements for remapping are illustrated in the following example. For a structure declared as follows:

```
DCL 1 A BASED(P),
    2 B,
    2 C,
    2 D(X REFER(B)),
    2 E(Y REFER(C)),
    2 F;
```

Any references to D, E, or F would result in the structure having to be remapped, but references to B or C would not.

At execution time, each reference to a REFER structure that necessitates remapping will result in the creation of a descriptor for the structure in question. To enable this, Phase IA generates a RESDES operator, which indicates to phases in the storage allocation stage that storage must be reserved for this descriptor. Each RESDES operator has a unique identification number to identify the relevant descriptor. Information is also inserted in the text to indicate mapping code which must be generated. This information consists of OFFS pseudo text tables, locator descriptor assignments, and MAP operators. The operands associated with the MAP operators supply the descriptor number (applied to the RESDES operator) and the reference of the start and finish points for the mapping operation.

Processing of Array INITIAL Assignments

In addition to the previously mentioned processing of INITIAL assignments (i.e., the repositioning of the assignments), Phase IA further analyzes INITIAL assignments to array items. As a result of this analysis, IASSN (iteration assignment) operators are generated to indicate multiple assignments. The use of these operators is shown in the following example. The statement:

```
DCL A(N) INIT((I+J) (2) (K),(M*N)4));
```

indicates that the first two elements of the array A are to be initialized to value K, the next (M*N) elements are to be initialized to the value 4, and that this sequence is to be repeated (I+J) times. After analyzing this declaration, Phase IA generates two iteration assignments:

```
A IASSN (K);
```

and

```
A IASSN 4;
```

The number of the elements of the array A that are to be initialized to the value (K) or 4 are supplied separately in array iteration descriptors generated by the phase. Each array iteration descriptor is preceded by an AID operator and followed by an ENDAID operator.

Special Processing of Built-in Functions

Whenever a reference to certain built-in functions or pseudo-variables is found, the argument list is examined to see if processing by this phase or later phases can be simplified. The particular items examined are references to the STRING, UNSPEC, DIM, HBOUND, and LBOUND built-in functions, and the REAL and IMAG pseudo-variables.

An example of the processing performed is where a single variable is passed as an argument to the UNSPEC built-in function. As the result returned by UNSPEC is a bit-string representation of the argument, the

reference to the function can be deleted and the required result obtained by modifying the DED of the argument so that it indicates a bit string of the appropriate length.

Other examples of special processing are in cases of references to the DIM, HBOUND, or LBOUND built-in functions with based-variable arguments. In such cases, the required information about the arguments can be found in the appropriate Aggregate Table entries in the general dictionary. Use of this information can avoid the need for the creation of locator chains by this phase.

This phase also detects the erroneous case where two or more arguments are supplied to an ADDR built-in function.

Processing of ALLOCATE and FREE Statements

ALLOCATE statements, together with any associated AGASSN, STRL, and INASSN statements, are merged from the declaration-expressions file into the main text stream by Phase GM. Phase IA repositions the associated statements in their correct positions relative to the ALLOCATE statement, similar to the repositioning of statements in the prologue. If storage mapping operations are required in connection with an ALLOCATE statements, RESDES and MAP pseudo text tables are generated as required.

If the SET and IN options applicable to the ALLOCATE statement have not been specified explicitly, Phase IA will imply the correct locator or area variables as required by the PL/I language.

If a FREE statement is found for which an IN option has not been specified explicitly, the implied option is generated. If the storage for a REFER structure is being freed, mapping code consisting of RESDES and MAP pseudo text tables, together with assignments to set up object-time descriptors, is generated to determine the storage to be freed.

MATCHING OF DATA-AGGREGATE ARGUMENTS (PHASE ID)

Phase ID performs four major functions. These are:

1. Examination of arguments specified in procedure calls and function references. If these arguments are data aggregates, or if the parameters of a procedure are data aggregates, the arguments and parameters are checked for matching of dimensions and attributes. Where necessary, (e.g., where an argument is an expression), a temporary operand is created to represent the argument in a format that is acceptable to the procedure or function (i.e., a dummy argument is created).

2. Examination of operands following INTO, FROM, and LIST options in input/output statements and any event variables in WAIT statements. If such an operand is an array-expression or an array cross-section, a temporary operand is generated.

3. Conversion of LEAVE to GOTO.

4. Processing of SELECT groups.

All references to subscripted identifiers are checked for validity, and for compatibility with their declarations.

PHASE INPUT

Input to the phase consists of main text stream in Type-1 text format. The variables dictionary and the general dictionary are accessed.

PHASE OUTPUT

Output from the phase consists of the main text stream in Type-1 text format, modified as follows:

1. Data aggregates that are used as arguments in procedure calls, in references to programmer-defined or built-in functions, or as operands in some input/output statements, are assigned in certain circumstances to aggregate temporary operands that are created by this phase.

2. All aggregate parameter descriptors are removed from the text.

3. RESDES and MAP pseudo text tables are generated wherever an aggregate temporary operand is generated, to enable later phases to reserve and map the necessary storage.

4. If aggregate temporary operands with adjustable bounds are generated, OFFS pseudo text tables and Q-temp. assignments are generated to set the values of the bounds in the relevant descriptor.

5. Statements containing invalid subscripted items are deleted.

6. LEAVE statements have been converted to GOTO statements to the end of the appropriate DO-group.

7. The output text stream for a SELECT group can be in one of two forms depending on whether the group will be executed using branch tables at execution-time.

8. For "optimized" SELECT groups a secondary text stream containing the branch tables required at execution-time. The branch tables are built by phase ID in the same format as phase PA builds STATIC entries for label constants (that is, 14 byte entries).

An entry is made in the variables dictionary for each aggregate temporary operand that is created, and a corresponding aggregate-table entry is made in the general dictionary.

PHASE OPERATION

Sequence of Processing

The main scanning routine, QSCAN, scans the main text stream sequentially, looking for the following items:

1. Any reference to a built-in function that is followed by a left parenthesis (indicating the presence of an argument list).
2. The appearance of an ANO code byte (X'3C'), indicating an argument list following a procedure call or function reference.
3. The appearance of an INTO, FROM, or LIST option in an input/output statement.
4. A WAIT statement.
5. A CALL statement or option.
6. Any subscripted identifier.
7. LEAVE, ITDO, ENDIT, and END statements.
8. SELECT, WHEN, OTHERWISE, and END (cf SELECT) statements.

Items not included in the above list are copied to the output text stream unaltered. When one of the listed items is found, a branch is made to an appropriate routine for processing of the item.

The routine ARGLIST is branched to if any of items 1, 2, or 3 is found. This routine checks for the presence of data aggregates in arguments, parameters, or operands. If none are found, control is passed to OUTARG and the item is copied unaltered to the output text stream. If an argument is an expression that contains an aggregate, or if the corresponding parameter is an aggregate, an aggregate temporary operand is created to replace the argument. If an argument is a data aggregate that is not contained in an expression, it is checked for matching with the relevant parameter. If matching is satisfactory, the argument is copied to the output text stream. If an argument and parameter do not match, an aggregate temporary operand is created and text is generated to assign the argument to the temporary operand. When an aggregate temporary operand is created, an appropriate entry is made in the variables dictionary and an aggregate table entry is made in the general dictionary. Data-aggregate operands in input/output statements do not require matching, but an aggregate temporary operand may be created where the operand is an expression, an array cross-section, or a subscripted structure. If the event variable in a WAIT statement is an array cross-section, a temporary operand is created. When the required text has been generated, control is returned to QSCAN and the text scan is continued.

If the operand of a CALL statement or option is an aggregate, the statement is deleted and a diagnostic message is generated.

When QSCAN finds a reference to a subscripted identifier, a branch is made to the QSUBS routine. This routine checks that all items in the subscript list are data elements, and also checks that the number of items in the subscript list is equal to the number of dimensions declared for the identifier. If a subscript list is invalid, the containing statement is deleted and a diagnostic message is generated. In performing this function, QSUBS may process an identifier that has already been processed by the ARGLIST routine as part of an argument.

Matching of Arguments and Parameters to Programmer-defined Procedures

Calls and function-references to programmer-defined procedures are processed by the ARGLIST routine. If the procedure is internal, each argument is preceded by a reference to a dictionary entry for the associated parameter descriptor. Parameter descriptors may not be available for calls and references to external procedures. A reference in text to a parameter descriptor is identified by a preceding PARD code byte (X'3D'). In the case of an array parameter, the descriptor is followed by the reference of its aggregate table in the general dictionary (ATR), identified by a preceding ATRM marker (X'39'). Structure parameter descriptors are followed by a list of structure element descriptors, (STRUDs), one for each base element, and the start and end of the STRUD list is indicated by STDMK (X'3A') and ESTMK (X'3B') code bytes. Array and structure arguments also have similar associated information and markers.

The subroutine SCANEXP is called to examine each argument in turn. It passes information to the ARGLIST routine in two 1-byte fields, KTEMP and EXPRTYPE. Flags set in these fields indicate such things as whether the argument is an expression, whether it is connected, whether it is an array or a structure, etc. If the argument is an expression, an aggregate temporary operand is generated as described in later paragraphs. If the argument is not an expression, the argument and parameter are compared as follows.

In the case of an array parameter, the DEDs of the parameter and argument are compared. In the case of a structure parameter, the structure element descriptors of the parameter and argument are compared. If the DEDs or STRUDs do not match, an aggregate temporary operand is generated. If the DEDs and STRUDs match, then a further check is made to see if the parameter and argument have the same aggregate table reference. If the ATRs are different, the dimensions and bounds of the argument are compared; if they match, an aggregate temporary operand is required, but if they do not, the statement is deleted. However, if the DEDs or STRUDs match, and both have the same ATR, no temporary operand is required. Control is passed to OUTARG, which copies the argument to the output text stream and deletes the parameter descriptor reference and the parameter and argument ATRs or STRUDs.

Checking Arguments to Built-in Functions

When a reference to any built-in function is accompanied by an argument list, the ARGLIST routine passes control to the ABIF routine. Each argument is examined to see if it is a structure; structure arguments are only valid in references to the STRING, ALLOCATION, or ADDR built-in functions. If a structure argument is found in a reference to any other built-in function, a diagnostic message is generated. If a reference to a built-in function that is not an aggregate-manipulation built-in function is valid, control is returned to ARGLIST and the function reference is copied unaltered to the output text stream.

The arguments to an aggregate-manipulation built-in function are checked for compatibility with the requirements of the PL/I language. For example:

- A data-aggregate argument to the STRING or ADDR built-in function must be a connected array or structure, and must not be an expression.
- Arguments to array-manipulation built-in functions must be arrays.

- A reference to the POLY built-in function must have two arguments, the first of which must be a single-dimension array, and the second must be a data element or a single-dimension array.

If the argument list is invalid, the statement is deleted. If the flags set by SCANEXP indicate that a temporary operand is required, or if the attributes or precisions of aggregate arguments differ from those required, an aggregate temporary operand is created as described in later paragraphs.

If the phase has to modify the argument to an ADDR built-in function, then it will output a two-argument ADDR built-in function - the first argument being the modified input argument and the second being the original input argument.

Processing of Operands in Input/Output Statements

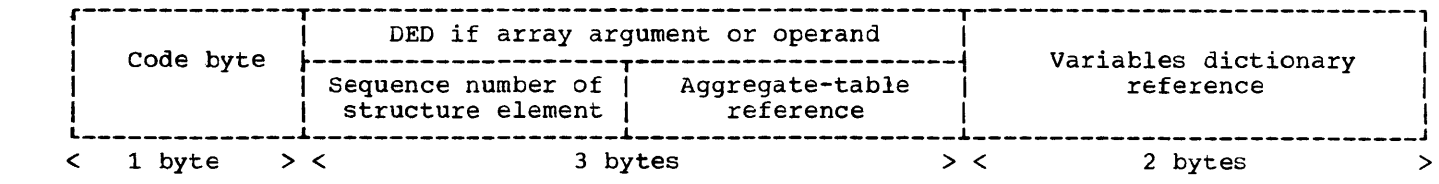
The ARGLIST routine is called to examine operands following the INTO, FROM, or LIST option in an input/output statement, and to replace the operands with aggregate temporary operands if required. An aggregate temporary operand is required in the following circumstances:

- If a LIST option is followed by an array cross-section or by an expression containing an array.
- If an INTO or FROM option is followed by a subscripted structure.

If these circumstances exist, an aggregate temporary operand is created as described in following paragraphs. In all other circumstances, control is passed to OUTARG and the operand is copied to the output text stream.

Generation of Aggregate Temporary Operands

The TEMP1 routine creates aggregate temporary operands in the circumstances described in preceding paragraphs, and in most cases generates text to assign the relevant argument or operand to the aggregate temporary operand. Unlike most of the temporary operands generated by the compiler, which are identified by a sequential number, aggregate temporary operands are identified by a reference to the variables dictionary entry that is made when each aggregate temporary operand is created. The general format of an aggregate temporary operand, which is similar to that for a 6-byte reference to a variable operand, is as follows:



When an aggregate temporary operand and its variables dictionary entry are created, an aggregate table entry is also made in the general dictionary. These aggregate tables are not commoned. The aggregate table reference for a structure temporary operand is included in the operand, but for an array temporary operand it is inserted in the text, preceded by an ATRM marker. Structure temporary operands are followed in the text by a STRUD list and markers. All aggregate temporary operands are preceded by RESDES and MAP pseudo text tables, and those with adjustable bounds also have OFFS pseudo text tables and Q-temp. assignments.

An aggregate temporary operand of the general form described is generated in the following circumstances:

- If the DEDs of an argument and a parameter do not match, or if an argument to a built-in function is not the required data type.
- If a parameter is declared with the CONNECTED attribute and the corresponding argument is not connected.
- If an argument to an array-manipulation built-in function is unaligned.
- If arguments to the POLY built-in function differ in mode or precision.
- If an argument, or an operand following PUT LIST, is an expression.

(Other situations where a special form of aggregate temporary operands are generated are described later.)

An example of situations where aggregate temporary operands are generated is a source program containing the following statements:

```
DCL A FLOAT,
  1 B,
    2 B1 CHAR,
    2 B2 BIT;

CALL E(A,B);

E:  PROC(X,Y);
  DCL X(6) FLOAT,
  1 Y,
     2 Y1 CHAR,
     2 Y2 FIXED;
```

In this example:

1. A data element, A, is used as an argument in a call to a procedure that has an array parameter, X.
2. The data type of the structure element B2, in the structure argument B, differs from the data type of the structure element Y2 in the structure parameter Y.

   The format of the source statement:

```
CALL E(A,B);
```

   on input to Phase ID can be represented as follows:

```
CALL E(PARD pard-X ATRM ATR A, PARD pard-Y STRUD-list B STRUD-list);
```

Aggregate temporary operands would be generated, and text generated to assign the arguments A and B to them. The text that would appear in the output from Phase ID can be represented as follows:

```
CALL E(RESDES MAP AGGASSN T1 ATR ASSN A;
T1, RESDES MAP AGGASSN T2 STRUD-list ASSN B STRUD-list; t2);
```

This text sequence represents three statements:

```
T1 = A;

T2 = B;

CALL E(T1,T2);
```

where the aggregate temporary operands have the same data type, dimensions, and structuring as the corresponding parameters X and Y.

A slightly different form of temporary operand is generated when an expression is used as an argument and no corresponding parameter descriptor is available, or when an expression is used in a PUT LIST statement. In such cases, although an aggregate temporary operand which reflects the dimensions and structuring of the result of the expression can be generated, the DED of the temporary operand cannot be completed by Phase ID. Instead, the first byte of the DED is set to X'41' to indicate that completion is required by a later phase. This form of temporary operand is referred to as a chameleon temporary operand.

Another form of aggregate temporary operand is created in the following circumstances:

- If an argument is an array cross-section or a subscripted structure, provided that, if there is a parameter descriptor, the argument matches it.
- If an operand following INTO or FROM is a subscripted structure.
- If an operand following LIST or WAIT is a cross-section of an array that does not contain structures.

In these circumstances, the argument or operand can be considered to be defined or overlaid on an existing data aggregate, and therefore no storage allocation is required for it. The temporary operand generated in such cases is referred to as a no-storage temporary operand. The variables dictionary entry created for a non-storage temporary operand is referred to as a reduced descriptor.

Processing LEAVE Statements

Phase ID keeps a stack of DO-groups, the stack being 50 levels deep. Each entry in the stack is made on encountering a DO statement. Entries are "popped-off" the stack when ENDDO and ENDIT text bytes are found. (Phase EA has expanded multiple END statements into separate ENDs and has created a syntactically correct set of DO-END pairs.)

Each entry in the stack consists of the following:

2 bytes - primary statement label of the DO statement. This is set on seeing the DO statement. (This element is initialized to a null value when the stack is pushed.)

2 bytes - Generated Statement Label (GSL) number allocated at the LEAVE statement (this element is initialized to a null value at the DO statement and completed at the LEAVE statement).

Action taken by Phase ID is as follows:

At the DO Statement: A stack entry of primary statement labels, if any, is created, together with a null GSL number. The stack index is then incremented.

At the LEAVE Statement: If the LEAVE statement has an identifier the primary label is picked up from the DREF of the identifier, and the stack is scanned to find this statement label. A check is made to see if a GSL number has already been allocated (by a previous LEAVE for this DO-group), and if so an SN GOTO (in place of the LEAVE) is generated with target GSL number equal to the number found in the stack entry just accessed. If a GSL number has not already been allocated, a GSL number is generated, the stack entry just accessed is completed with this GSL number, and the SN GOTO (in place of the LEAVE) is generated.

If the LEAVE statement has no identifier the top entry of the stack (the stack entry for the immediately containing DO-group to be left) is accessed. A check is made to see if a GSL number has been allocated, and then processing is continued as described above.

At the ENDIT and ENDDO Statements: The top of the stack is accessed and if the entry has a GSL number allocated a GSL text table is generated with this number after the ENDIT or ENDDO. The stack is then "popped-up". If the entry has a null GSL number the stack is just "popped-up".

Processing SELECT groups

Phase ID contains routines to process the following Statement Number (SN) tables:

SN SELECT

SN WHEN

SN OTHERWISE

SN ENDSELECT

SN MAP

SN CHECK/NOCHECK

The routines are accessed after XSTAT and STMTYP have been set up and before output of the statement header.

Action taken by Phase ID at each of the SN tables is as follows:

At SN SELECT: A new entry in the SELECT stack is created and the text reference of the SELECT statement saved. The dictionary 'SELECT OPTIMIZATION' entry is then accessed followed by updating of the dictionary reference slot from the chain slot in the dictionary entry. If on scanning the expression is found to be scalar the expression is assigned to a temporary. If not scalar, message 644 is issued. Output text is then built using the SELECT optimization dictionary entry.

At SN WHEN: The text reference of the WHEN statement and each expression is scanned for being scalar. If scalar, output consists of the correct text stream according to the opt flag in the dictionary. Otherwise, message 644 is issued and output built.

At SN OTHERWISE: Output consists of the correct text stream according to the opt flag in the dictionary.

At SN ENDSELECT: If "OTHERWISE SEEN" flag is set OFF in dictionary entry the text for "OTHERWISE RAISE ERROR" is issued. The SELECT stack is then "popped-up".

At SN MAP for WHEN (SN00): SN00 is reset to SN MAP, and output is produced that will be put out during processing of SN WHEN.

At SN CHECK/NOCHECK: The text reference and statement number are saved.

EXPANSION OF DATA AGGREGATES (PHASE IE)

Phase IE examines all data aggregates in the text, and where necessary, generates text which expands the aggregates into elements. Much of the work of the phase is involved with the processing of data aggregate assignments. These assignments may stem directly from source program statements, or may be assignments generated by Phase ID in the course of aggregate argument matching or processing of some built-in functions. Array assignments are effectively converted into element assignments by placing them in appropriate do-loops. Structure assignments are expanded into individual base element assignments and, if the base elements are dimensioned, they are placed in appropriate do-loops. Some aggregate assignments are tested for possible optimization, whereby the assignment can be performed by MVC instructions without expansion into a number of individual element assignments. Phase IE indicates the assignments for which this is possible, so that they can be correctly processed by Phase II and KE.

PHASE INPUT

Input to the phase consists of the main text stream in Type 1 text format. Each statement header is examined in order to identify statements processed by the phase. Each of these statements is examined for the presence of data-aggregate operands (including aggregate temporary operands generated by Phase ID).

The general dictionary is accessed. The names and variables dictionaries are accessed if the text contains structure BY-NAME assignments.

PHASE OUTPUT

Output from the phase consists of the main text stream in Type 1 text format, modified as follows:

- Data aggregates in the text are expanded into individual elements or, if the aggregate is dimensioned, code bytes are inserted to indicate where do-loops are required to perform the expansion.
- Aggregate assignments are expanded into individual element assignments. Where an aggregate assignment can be performed without such expansion, the statement-type code byte in the statement header is changed to SOASSN.
- Temporary operands are generated and inserted in the text, as described in the details of processing.

No dictionary entries are made by this phase.

PHASE OPERATION

Sequence of Processing

The input text stream is scanned sequentially. When a statement header is found, the routine TSTPROC examines the statement-type code byte. If the statement is a PROC, ENTRY, ON, or BEGIN statement, the entire statement is copied to the output text stream unaltered. For other

types of statement, the operands are examined by appropriate routines, and copied to the output unaltered unless they are either an operand in an aggregate assignment or an aggregate operand in an input/output statement.

If the statement-type code byte indicates that the statement is an assignment statement or a stream-oriented input/output statement, the first operand is examined by the routine FRST. If this operand is a structure, the routine STRUCASS is branched to; if the operand is an array, the routine ARAS is branched to. If the first operand is not a data aggregate, it is copied to the output text stream unaltered, and any other operands are examined in turn. Similarly, for statements other than the statement types mentioned, each operand is examined in turn. If any of these operands is a procedure call or a function reference with an argument list containing an aggregate assignment (as generated by Phase ID), control is passed to either STRUCASS or ARAS according to whether the aggregate temporary operand is a structure or array.

The routines STRUCASS and ARAS perform the main processing functions of the phase. Information about the first operand is extracted from appropriate dictionary entries and stacked. If the statement is an assignment statement, the right-hand side of the assignment is compared with the stacked information for possible optimization. Text is generated to indicate either optimized assignment or expansion of the aggregate operands by use of DO-loops.

Processing of Assignments

The first operand (left-hand side) of an assignment is referred to as the master-operand. By examination of the master operand, each assignment is classified as belonging to one of three types:

- Element assignments, when the master operand is an element variable.
- Array assignments, when the master operand is an array.
- Structure assignments, when the master operand is a structure.

Processing Element Assignments: Element assignments are copied to output text stream unaltered, except when the right-hand side of an assignment contains a function reference with an argument list which itself contains an aggregate assignment (generated by Phase ID). In this case, the aggregate assignment is always expanded, as described in following paragraphs.

Processing Array Assignments: The aggregate table entry for the master operand is accessed and, for each dimension, an entry is made in a stack, DIMSTAK. Each entry contains the values of the upper and lower bounds (if known), and flags which are set if either of the bounds is adjustable. The entries in DIMSTAK are used for comparison when checking the validity of other operands, and when testing for the possibility of executing the assignment without expansion into individual element assignments. Full comparisons to test for the possibility of the optimized form of assignment are only carried out when the following conditions exist:

1. The master operand is a connected array with fixed bounds, and is not a compiler-generated aggregate temporary operand

or

   the master operand is a cross-section of an array with connected elements.

2. The right-hand side of the assignment is not an expression, and is a connected array with fixed bounds, a cross-section of an array with connected elements, an element variable, or a constant.

In checking whether these conditions are satisfied, a check is made to see if the source operand has the same aggregate table reference as the master operand (target). If not, the aggregate table entry for the source operand is accessed, and the bounds of each dimension are compared with the appropriate entry in DIMSTAK. If the dimensions and bounds match, the operand DEDs are compared. If the DEDs match, the assignment can be performed without expansion, and this is indicated by changing the code byte in the statement header to SOASSN (X'7C'). The assignment is copied to the output text stream unaltered, except that the ATRs (and ATRMs) of the operands are deleted (unless an operand is followed by subscripts).

Examples of aggregate assignment source statements which do not require expansions are shown below:

```
   DCL A(3,4,5),
       B(3,4,5),
       C(4),
       X DECIMAL FLOAT;
   .
   .
   .
   A=B;
   A=C(2);
   A=X;
   B=1;
   A(2,*,*)=B(2,*,*);
(But not A(*,*,2)=B(*,*,2);)
```

If the conditions described are not satisfied, the assignment is expanded into a series of individual element assignments by the generation of appropriate do-loops. Phase IE generates markers to indicate the start and end of the required do-loops. It also generates temporary operands which are used to identify the loops, and to which values are assigned by Phase KE so that they can be used as loop control variables. This processing starts when entries for each dimension of the master operand are made in DIMSTAK. For each do-loop that is required, a temporary operand (which requires no storage and therefore requires no dictionary entry) is generated. Each temporary operand is identified by a unique number, and this identifying number is inserted in the appropriate entry in DIMSTAK. If it is found that the assignment does not require expansion, the temporary-operand numbers are not used.

If the assignment is to be expanded, the SASSN code byte is retained in the statement header, and a DOTMP marker (X'9F') is generated to indicate where the start of a do-loop is required. The validity of the dimensions and bounds of the first operand on the right-hand side of the assignment is checked by comparison with the entries in DIMSTAK for the master operand. If they are valid, text representing the master operand, the assignment operator, and the first source operand is generated. Each operand identifier is followed by subscripts, which are either derived from the source program statements or, in the case of array assignments, are asterisks or the loop-control temporary operand. If the right-hand side of the assignment is an expression, remaining operands are in turn checked for validity by comparison with the master operand, and appropriate text is generated. When all operands have been processed, an ENTMP marker (X'9E') is generated to indicate the end of the do-loop. Examples of the text generated are shown in the following examples:

If identifiers are declared as follows:

```
DCL A(3,4,5),
    B(3,4,5),
    C(3,4,5),
    X DECIMAL FLOAT;
```

then A = X+5; becomes:

```
    DOTMP t1;
    A(*,*,t1)=X+5;
    ENTMP t1;
```

```
    A(1,*,2)=B(3,*,4); becomes:

    DOTMP t2;
    A(1,t2,2)=B(3,t2,4);
    ENTMP t2;
```

```
A =B+C(2,3,4); becomes:

    DOTMP t3;
    A(*,*,t3)=B(*,*,t3)+C(2,3,4);
    ENTMP t3;
```

Where an assignment includes an array cross-section with more than one asterisked dimension, one do-loop is generated for each group of contiguous asterisks. (This does not necessarily imply that the elements are contiguous in storage.) This is illustrated in the following example:

```
DCL P(5,6,7,8,9),
    Q(5,6,7,8,9)FIXED BIN;

P(*,*,5,*,*)=Q(*,*,2,*,*); becomes:

DOTMP t1;
DOTMP t2;
P(*,t1,5,*,t2)=Q(*,t1,2,*,t2);
ENTMP t2;
ENTMP t1;
```

Processing Structure Assignments: The aggregate table entry for the master operand is accessed, and for each member up to the first base element, an entry is made in a stack, STSTAK. Each 8-byte entry contains details of the level of the structure member, and its dimensions (if any). If a base member is dimensioned, an entry is made im DIMSTAK for each dimension. The right-hand side of the assignment is then scanned. If it is a single structure, comparisons are made to check for the possibility of an unexpanded assignment. If it is an expression, the assignment must be expanded and comparisons are only made to check for validity.

Comparisons are made, one operand at a time, with the corresponding entries in STSTAK and then with the corresponding entries in DIMSTAK. If the first base elements match, and the assignment does not contain an expression, the corresponding STRUDs of all base members are compared. If the STRUDs match, the assignment is either suitable for optimization or is invalid because of variations in structuring. A flag, OPTSW, is set to indicate that expansion is not required, and further entries are made in STSTAK and DIMSTAK for the purpose of checking the structuring of the two operands. If all checks are satisfactory, the code byte in the statement header is changed to SOASSN (X'7C') and the assignment is copied unaltered to the output text stream. If the STRUDs do not match, or if the assignment contains an expression, the assignment must be expanded. The structure members of the first operand on the right-hand side, down to the first base element, are checked for validity by

comparison with the master operand entries in STSTAK and DIMSTAK. The process is repeated until all base elements have been checked and the individual assignments generated. Where a structure assignment is expanded, the SASSN code byte is retained in the statement header. Text output by the phase is illustrated in the following examples:

```
DCL 1 R,
      2 S,
        3 T,
        3 U;
.
.
.

R=R+1;
```

is expanded into:

```
T=T+1;
U=U+1;
```

If a structure is dimensioned, temporary operands, and DOTMP and ENTMP markers are generated to indicate do-loops, as shown in the following example:

```
DCL 1 S(2),
      2 T(3,3),
        3 U(4),
        3 W;
.
.
.
.
S=S+1;
```

is expanded into:

```
DOTMP t1;
DOTMP t2;
DOTMP t3;
U(t1,*,t2,t3)=U(t1,*,t2,t3)+1;
ENTMP t3;
W(t1,*,t2)=W(t1,*,t2)+1;
ENTMP t2;
ENTMP t1;
```

The foregoing example illustrates the following features of optimization of do-loops:

1. Where an inherited dimension is common, only one do-loop is generated.

2. Where a structure member has more than one dimension, only one do-loop is generated.

Processing Structure BY NAME Assignments

If the TSTPROC routine detects a statement containing a structure assignment with the BY NAME option, the BYNAME routine is branched to. This routine calls the BYNASN subroutine to process the assignment. The processing is similar to that for other structure assignments. Entries are made in STSTAK for each member down to the first base element. Entries are made in DIMSTAK for each dimension of a dimensioned member. Aggregate operands on the right-hand side of the assignment are then compared in turn with the entries in STSTAK, to detect any similarly

qualified name at the same level of structuring. Where matching names are found, element assignments are generated, with indications for do-loop expansion where a base element is dimensioned.

Processing Aggregates in Stream I/O Statements

If the TSTPROC routine detects a statement header for a GET or PUT statement, the statement is processed in a similar manner to assignment statements. The operands in the data list are examined in turn for the presence of data aggregates, and a branch is made to either the STRUCASS routine or the ARAS routine if a structure or array that requires expansion is found. The structures are expanded into a sequence of base elements, and arrays are expanded by the generation of surrounding do-loops .

If an expression containing aggregates is found, the first aggregate in the expression is used as the master operand. If an aggregate with greater extents is found later in the expression, it is then treated as the master operand, and the expression is reprocessed. Examples of the processing performed are shown in the following examples:

```
DCL 1 X,
      2 XA,
      2 XB,
    1 Y,
      2 YA,
      2 YB,
    P(5),
    Q(5);


.
.
.
.
GET EDIT (X) (A(1));
```

is expanded to:

```
GET EDIT (XA,XB) (A(1));
```

```
PUT EDIT (X+Y)(A(1));
```

is expanded to:

```
PUT EDIT (XA+YA,XB+YB)(A(1));
```

The statement:

```
PUT LIST (P+Q);
```

will appear in the input to Phase IE as an assignment to an aggregate temporary operand, generated by Phase ID. After expansion, this assignment becomes:

```
PUT LIST (DOTMP t1;
T1(t1)=P(t1)+Q(t1);
ENTMP t1; T1);
```

The processing performed on various forms of stream-oriented input/output statements is summarized in figure 2.13.

Processing of Based Structures

If a based structure that is preceded by a locator chain (created by Phase IA), appears in an assignment, or in a data list in a stream-oriented input/output statement, the locator chain is required to precede each base element in the output text stream when the structure is expanded into base elements. To avoid such repetition a pointer temporary operand (shown in examples as "p.temp.") is generated and the locator chain is assigned to it by use of a PCASSN operator. This assignment is generated before the first base element or base-element assignment. Each subsequent base element in the expansion is preceded by the pointer temporary operand. This is illustrated in the following example:

```
DCL 1 A BASED(P),
    2 B,
    2 C,
    2 D;
DCL P POINTER BASED(Q),
    Q POINTER BASED(R);
.
.
.
A=1;
```

On input to Phase IE, this assignment statement is represented by:

```
R PTS Q PTS P PTS A ASSN 1;
```

On output from Phase IE, the assignment to the pointer temporary operand, and the expansion of the structure assignment appear as follows:

```
R PTS Q PTS P PCASSN p.temp.1 PTS B ASSN 1;
p.temp.1 PTS C ASSN 1;
p.temp.1 PTS D ASSN 1;
```

Type of operand	Type of input/output statement					
	Data directed		List directed		Edit directed	
	GET	PUT	GET	PUT	GET	PUT
Single element variable or constant	Unaltered	Unaltered	Unaltered	Unaltered	Unaltered	Unaltered
Elemnt expression	Invalid	Invalid	Invalid	Unaltered	Invalid	Unaltered
Single array	Unaltered	Unaltered	Unaltered	Unaltered	Subscripted and enclosed in do-loop/s	Subscripted and enclosed in do-loop/s
Array expression	Invalid	Invalid	Invalid	Expanded (aggregate temporary generated by Phase ID)	Invalid	Expression expanded with do-loop/s
Array cross-section	Invalid	Subscripted and enclosed in do-loop/s	Unaltered (no-storage temporary generated by Phase ID)	Unaltered (no-storage temporary generated by Phase ID)	Subscripted and enclosed in do-loop/s	Subscripted and enclosed in do-loop/s
Single array-element	Invalid	Unaltered	Unaltered	Unaltered	Unaltered	Unaltered
Single structure	Unaltered	Unaltered	Expanded into individual base elements	Expanded into individual base elements	Expanded into individual base elements (Subscripted and enclosed in do-loop/s if dimensioned)	Expanded into individual base elements (Subscripted and enclosed in do-loop/s if dimensioned)
Structure expression	Invalid	Invalid	Invalid	Expanded into individual base elements	Invalid	Expanded into individual base elements

Figure 2.13. Summary of stream I/O data processing performed by Phase IE

Processing Aggregates in Procedure Calls and Function References

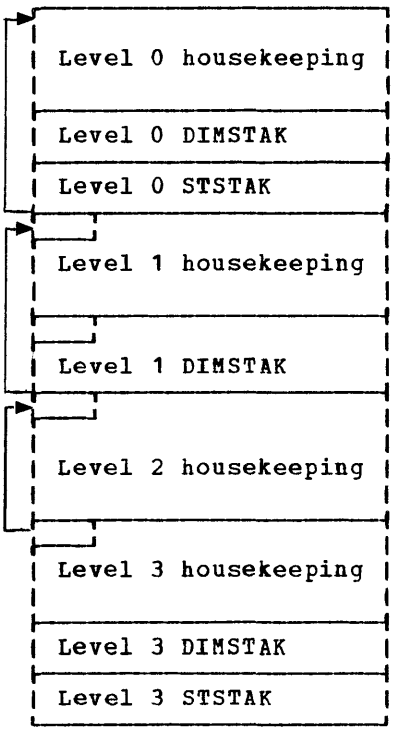
Where data aggregates are used as arguments in procedure calls and function references, they will have been examined by Phase ID, and assignments to dummy arguments (aggregate temporary operands) will have been generated in some cases. When these assignments are processed by Phase IE, no attempt is made to avoid expansion into individual elements. Aggregate arguments (including assignments) are expanded into elements in all cases, except where they are used in references to the STRING, ALLOCATION, or ADDR built-in functions, in which case they are copied to the output text stream unaltered.

Because function references may be nested in argument lists, a nested stacking system is used. The subroutine FUNEST sets up a stack, DSTAK, when a procedure call or function reference with an argument list is found. As the argument list is scanned, a section of DSTAK is built for each level of nesting. If more than one section is required, a backwards pointing chain is created in the stack. This chain enables each section to be readily accessed when the stack entries are later processed from the innermost level of nesting to the outermost level.

Each nesting-level section of DSTAK consists of a housekeeping area and, if the argument at that level contains an aggregate, a structure stack (STSTAK) and/or a dimensions stack (DIMSTAK). If a particular argument level does not contain an aggregate, information about the argument will be contained in the housekeeping section of DSTAK for that level, and STSTAK and DIMSTAK will not be set up in that section. The basic structure of DSTAK is illustrated in the following example.

```
   DCL E1,E2,E3,E4 ENTRY;
   DCL 1 STA(6);
         2 M1,
         2 M2,

       1 STB(6) LIKE STA,

       (P,Q)(4,4),
       I,J FIXED DECIMAL;
    .
    .
    .

LAB:CALL E1(STA+E2(P+Q+E3(I*J+E4(STB+5))));
           ^      ^      ^      ^
           |      |      |      |
           |      |      |      |
           |      |      |      |
      level0 level1 level2 level3
```

Basic structure of DSTAK for CALL statement at LAB.



When DSTAK has been set up, expansion of aggregates is carried out, starting at the innermost level of argument. Where there are structures in an outer level of argument, the inner levels are processed for each base member of the outer level.

EXPRESSION ANALYSIS AND TEXT TRANSLATION (PHASE II)

Phase II examines every statement in the main text stream and performs the following functions:

1. Translates the sequential input (Type 1 text) into a bracket free format consisting of fixed length text tables (Type 2 text). Within these text tables, statements and statement elements are organized in a standard form that enables ease of processing by later phases of the compiler.

2. Analyzes all expressions, breaking them down into components most suitable for evaluation. Temporary operands are created to hold the results of intermediate expression evaluation or operand data-type conversion.

3. Generates qualified-name temporary operands (Q-temps.) to facilitate handling of subscripted variables and based variables.

4. Analyzes built-in function references. The analysis includes checking the validity of arguments, generation of dummy arguments as required, and determination of the function result data-type.

5. Checks that all non-aggregate arguments in procedure calls and function references match their corresponding parameters.

6. Performs generic selection of procedure entry points.

PHASE INPUT

Input to the phase consists of the main text stream in Type 1 text format. Within this text stream, blocks have been de-nested and within each block statements are retained in source-program order. Each statement is preceded by a statement header which identifies the type of the statement. Within each statement, its components (operators and operands) are retained in source-program order, but additional operators, operands, markers, and pseudo-text tables have been inserted by preceding phases.

Some entries in the general and variables dictionaries may be accessed by the phase.

PHASE OUTPUT

Output from the phase consists of a completely rebuilt main text stream in which components of text are organized in a series of fixed length (32-byte) text tables. A statement-header text table is inserted in front of each statement to hold general information about the statement. The body of each statement consists of a series of text tables in which operators and operands are organized in fixed-length fields in predetermined order. All parentheses are removed from the text. This format of text is referred to as Type-2 text.

During analysis and translation of the text, special operators, temporary operands, and compiler-generated labels are generated as described in the description of phase operation.

Some variables dictionary entries for aggregate temporary operands may be changed during processing by this phase.

Translation to Type-2 Text

One of the most significant functions of Phase II is the translation of the entire main text stream from a continuous stream of operators, operands, and control information (Type-1 text) into a format that is most suitable for further processing. The format used is referred to as Type-2 text, and consists of a series of fixed length (32 bytes) tables. With a few exceptions, such as the text tables used for statement headers or PROC and ENTRY statements, a standard arrangement of fields is used in the text tables for holding operators, operands, and general information in the form of flags, chains, etc. The basic format of a Type-2 text table is shown below:

Field content	Field length in bytes
Operator	2
Operand 1	6
Operand 2	6
Operand 3	6
Statement number	2
General information (flags and chains)	10

Each text table is classified by the first byte of the operator. The operator code bytes used in Type 2 text are shown in figure 5.89. Some types of text tables may be further classified by a code byte in the second byte of the operator. This is illustrated in figure 5.92, which shows the various types of text tables and the use of the operand fields. The order of the operator and operands in the text tables is the sequence in which algebraic operations are performed. For example, the statement:

```
A = B + C;
```

is translated into:

PLUS	B	C	A
(Operator)	(Operand 1)	(Operand 2)	(Operand 3)

Note that the Operand 3 field contains the result of evaluation of the Operator-Operand 1-Operand 2 triple. In some types of text table, some of the operand fields may not be used. For example, the statement:

```
A = B;
```

is translated into:

ASSN	B	null	A
(Operator)	(Operand 1)	(Operand 2)	(Operand 3)

During translation of the text expressions are analyzed, and in some cases a temporary operand is generated, e.g., to hold the result of intermediate evaluation of an expression or where data-type conversion of an operand is required. These temporary operands, which require no storage and therefore require no dictionary entry, are uniquely identified by a sequential number, and are represented in the following examples by "Tn", where "n" is an identification number. For example, during the translation of the statement:

```
A = B + C - D;
```

a temporary operand is required to hold the result of an intermediate evaluation of the expression. The expression is translated as follows:

```
PLUS     B     C     T1

MINUS    T1    D     A
```

Temporary operands created during this phase are local temporaries, i.e., their use is confined to within a single statement.

The translation technique used by this phase is based upon the allocation of various priority levels to operators used in the text. Figure 2.14 shows a list of some of the more commonly used operators and the priorities assigned to them. A complete list of operators and their priorities can be found in the table PRITAB in the published listing of Phase II.

Operator	Meaning	Priority Level
LPAR	Left parenthesis	32 (2)
PPLUS, PMINUS	Prefix plus/minus	28 (27)
NOT	Logical NOT	28 (27)
POWER	Exponentiation	28 (27)
DIV, MULT	Divide/Multiply	26
PLUS, MINUS	Infix plus/minus	25
CONCAT	Concatenate	24
GT, EQ, LT, GE, LE, NE	Comparison operators	23
AND	Logical AND	22
OR	Logical OR	21
TO, BY	DO specification operator	16 (15)
DOEQ	DO specification operator	14
DO, ITDO	DO specification operator	13
COMMA	Data-element separator	11
IF	IF statement operator	8
THEN	IF statement operator	7 (0)
ELSE	IF statement operator	6 (1)
ASSN	Assignment	5
GOTO	Branching operator	3
ENDO, ENDIT	DO loop delimiter	2
SCOLON	Statement delimiter	1
RPAR	Right parenthesis	0

Note: This is not a complete list of the operators used. Acomplete list may be found in the table PRITAB in the published listing of Phase II.

Figure 2.14. Priority levels of operators commonly used in text translation

A pushdown stack, MSTACK, is used in the translation process, on a last-in, first-out (LIFO) basis. MSTACK can be considered as a temporary storage table consisting of two sections; one section being used for operators and the other for operands.

For various reasons, such as the free format permitted in some PL/I statements, a number of operators have two priority levels. These are shown in figure 2.14 where the value shown in parentheses is the priority applied to the operator when it is in MSTACK. The other priority applies to the operator when it is in the input text. An example of the need for dual priorities is in the following statements:

```
DO I = 1 TO 10 BY 2;
```

can also be written as:

```
DO I = 1 BY 2 TO 10;
```

To enable correct processing of such statements, the operators TO and BY each have two priority levels.

The input stream is scanned sequentially. At the beginning of each statement, a statement header text table is generated. MSTACK is not used for this purpose. The format of a statement header text table is shown in figure 5.90.

The body of the statement is scanned step by step, each step consisting of an operator-operand pair. Register 1 is used as a pointer as the text is scanned. MSTACK is initialized with a special bottom-of-stack marker, which is a null operator with priority equal to that of a statement delimiter (semi-colon). One result of this is that an operand which has no associated operator, (e.g., the operand A in the statement A = B), can be stacked under the operator-operand pair concept. The initial operand is paired with the null operator.

As each operator-operand pair is scanned, the priority of the operator in text is compared with the priority of the top operator in MSTACK. If the operator in text has the higher priority, then the operator-operand pair in text is added to the top of MSTACK. If the two operators have equal priority, or if the top operator in MSTACK has the higher priority, then unstacking action takes place.

The unstacking action initially consists of the removal of the top operator and the top two operands from MSTACK. This action is controlled by routine TG7, which examines the unstacked operator and calls other routines, appropriate to the operator type, to perform further processing as required. In many cases, such processing merely involves the placing of information in the appropriate fields of a text table to be generated. In other cases, more involved processing is required. The processing required when an arithmetic operator is unstacked is described below.

When an arithmetic operator is unstacked, routine TG7 calls the expression-analysis routine, EA1. This routine detects any operand data-type conversions that are required, and determines the data type of the result. The result forms the third operand in the text table that is then generated. The result operand is also placed at the top of MSTACK to replace the lower of the two unstacked operands. Scanning of the statement, with appropriate stacking and unstacking action, continues until the whole statement has been processed. The statement delimiter (semicolon) has a very low priority and causes unstacking action until MSTACK is empty.

An example of the translation of a statement containing an expression is shown in figure 2.15. Throughout the example the data types of the variables involved have been assumed to be such that no data-type conversions are required. Note that the processing stage numbers shown in the example have no significance except for illustrative purposes.

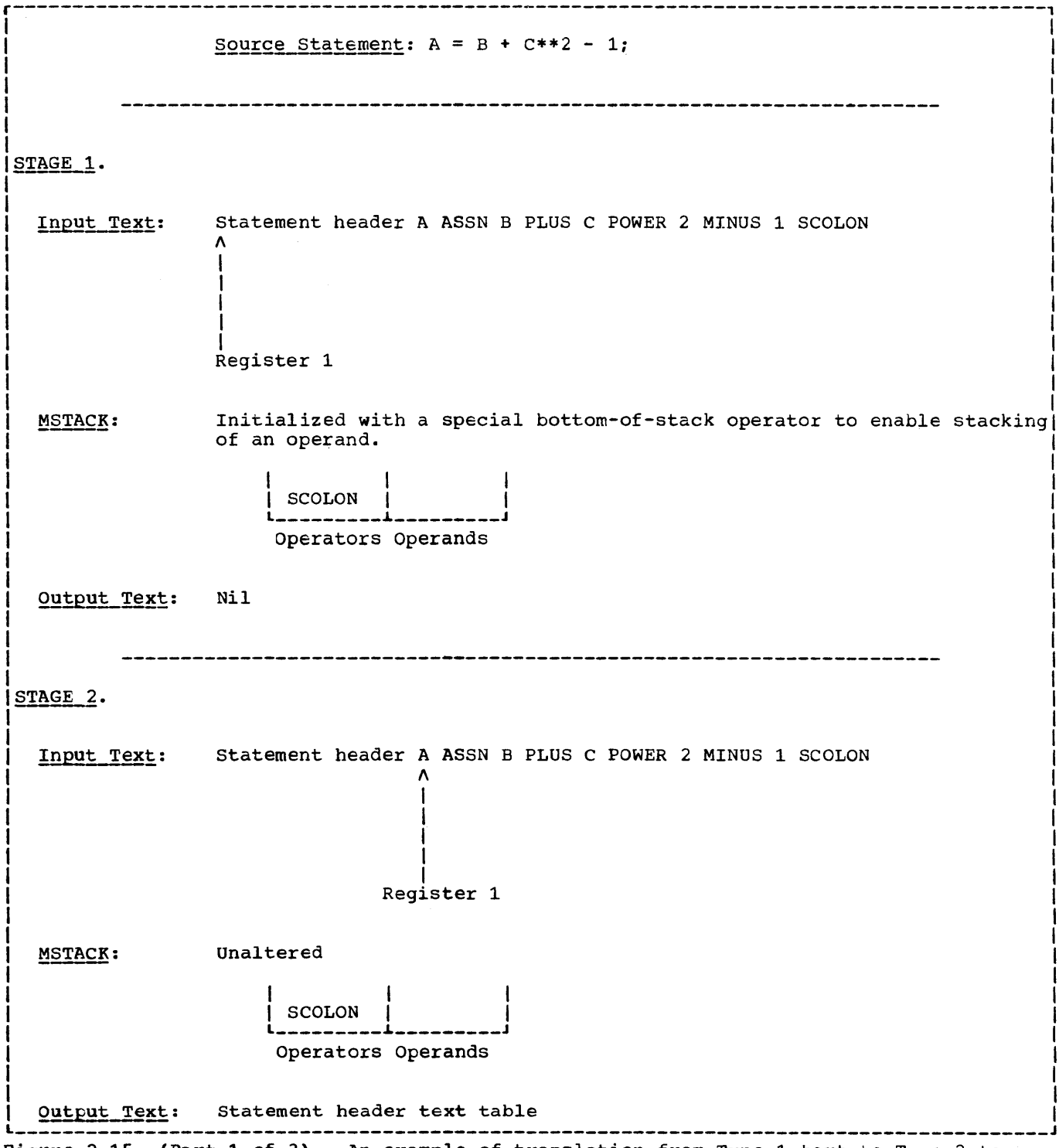


Figure 2.15. (Part 1 of 3). An example of translation from Type-1 text to Type-2 text

STAGE 3.

Input Text: Statement header A ASSN B PLUS C POWER 2 MINUS 1 SCOLON

Register 1.

MSTACK:

Operators	Operands
POWER	2
PLUS	C
ASSN	B
SCOLON	A

Output Text: Statement header text table (as before).

Successive comparisons of operator priorities, where ASSN priority > SCOLON, PLUS priority > ASSN, and POWER priority > PLUS, cause the operator-operand pairs to be stacked as shown. No operator priority has caused unstacking or text table generation at this stage.

STAGE 4.

Input Text: Statement header A ASSN B PLUS C POWER 2 MINUS 1 SCOLON

Register 1

MSTACK:

Operators	Operands
PLUS	T1
ASSN	B
SCOLON	A

Output Text: Statement header text table

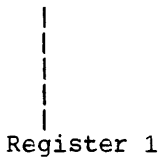
(Operator)	(Operand 1)	(Operand 2)	(Operand 3)
POWER	C	2	T1

Comparison of the priority of the MINUS operator and the priority of the POWER operator causes unstacking and the generation of a POWER text table.

Figure 2.15. (Part 2 of 3). An example of translation from Type-1 text to Type-2 text

STAGE 5.

Input Text: Statement header A ASSN B PLUS C POWER 2 MINUS 1 SCOLON



MSTACK:

Operators	Operands
MINUS	1
ASSN	T2
SCOLON	A

Output Text: Statement header text table

(Operator)	(Operand 1)	(Operand 2)	(Operand 3)
POWER	C	2	T1
PLUS	B	T1	T2

Comparison of the priority of the MINUS operator with the priority of the PLUS operator causes further unstacking and the generation of a PLUS text table. The MINUS 1 operator-operand pair are then stacked.

STAGE 6.

Input Text: Register 1 points at the beginning of the next statement header.

MSTACK: Comparison of the priority of SCOLON with operators in MSTACK causes all items in MSTACK to be unstacked.

Output Text: Translation of the statement to Type 2 text is completed when the MINUS and ASSN operator-operand pairs are unstacked. No intermediate result temporary operand is needed for the result of the MINUS operation, and the variable A is used in the third operand field.

Statement header text table

(Operator)	(Operand 1)	(Operand 2)	(Operand 3)
POWER	C	2	T1
PLUS	B	T1	T2
MINUS	T2	1	A

Figure 2.15. (Part 3 of 3). An example of translation from Type-1 text to Type-2 text

Translation of DO and IF Statements

Labels are generated by the compiler, and inserted in the text to indicate the start and end of each do-loop, and possible branching

points in IF statements (e.g., if an ELSE clause is present in an IF statement, a GOTO operator is inserted at the end of the THEN clause, with a compiler-generated label indicating the branching point). The use of these labels is illustrated in the descriptions of the phases that process the statements in the statement processing stage.

Each compiler-generated label is given a unique identifying number. Labels are represented in descriptions by the characters 'CLn', where n is the identifying number.

To enable compiler-generated labels to be applied in the correct sequence during translation of DO and IF statements, two special stacks, DOSTACK and IFSTACK are used for temporary stacking of the labels.

Expression Analysis

As each statement is translated into Type-2 text, expressions within the statement are analyzed to determine the intermediate operations required in the evaluation of the expression. Text tables are generated to enable these operations to be performed, and temporary operands are created to represent the results of the intermediate operations (as shown in the example of text translation).

Temporary operands are also created when the data types of operands in an expression require conversion before the expression can be evaluated, e.g., if a character-string operand is used in an arithmetic operation, the operand must be converted to an arithmetic data-type.

As each operator/operand pair is unstacked from MSTACK, the routine EA1 is called to determine whether the operand requires data-type conversion. If conversion is required, a CONV text table is generated for later processing by Phase KX. The target operand of the CONV text table is a temporary operand, and this operand replaces the original operand in the text table for the operator concerned. The exact result type, including scale and precision, is then evaluated. The use of CONV text tables is illustrated in the following example:

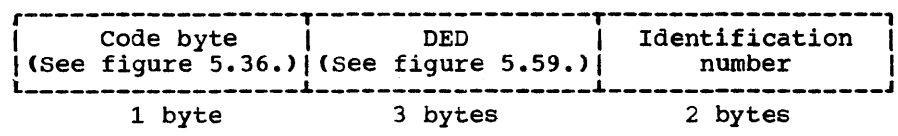
```
DECLARE  X BIT(6), A FIXED DEC, B FIXED BIN,

         C CHAR(2), D BIT(3);

         X = A + B < C | D;
```

Translation and analysis of the expression results in the following text tables being generated:

```
CONV        A        -        T1
PLUS        T1       B        T2
CONV        C        -        T3
LT          T2       T3       T4
OR          T4       D        X
```

T1 and T3 are the results of the conversion of operands A and C to binary data. T2 and T4 are temporary operands generated to hold the results of intermediate operations in evaluation of the expression.

Each temporary operand has a unique identification number, and is represented in text as:



The code byte illustrates:

1. Whether the temporary operand is:

   a. local - its use is confined to within one statement.

   b. global - it may be used in several statements.

2. Whether the temporary operand can reside in a register.

3. The last usage of the temporary operand.

Phase II completes the DED (data element descriptor) for all the temporary operands that it creates. For example, where temporary operands represent intermediate results, the data type will be that of operand 1 and 2, while scale and precision will be evaluated according to the rules of PL/I for the operation performed.

Qualified-name Temporary Operands (Q-temps.)

Whenever a qualified item, such as an array-element or a based variable, is referred to, a means of referring to the qualifying information is required. During text translation and during processing by other phases, special temporary operands called qualified-name temporary operands are generated to replace such qualified items. These temporary operands are referred to in the published listing and in the following descriptions as Q-temps. A Q-temp. contains a description of the data type of the item referred to, and provides a means of identifying the qualifying information by chaining to the text table in which the Q-temp. was created. Each Q-temp. is uniquely identified by a sequential number. The formats of Q-temps. are illustrated in figures 5.52 and 5.55.

When an array element is referred to, it is necessary to evaluate the subscript in order to calculate the offset of that particular element from the start of the array. When a statement containing such a reference is translated and analyzed, a special stack, SFSTACK, is used in conjunction with MSTACK. SFSTACK is used to hold the array reference, the index temporary operand (used in subscript evaluation), and the type of the subscript (e.g., iSUB defined). Use of SFSTACK enables multiple and nested subscripted-variable references to be processed in such a way that an NDX table, in which the requisite Q-temp. is created, is generated immediately prior to use of the subscripted variable. The generated Q-temp. then replaces the array variable in MSTACK.

When a based variable is qualified by one or more pointers, the based variable will be preceded by a PTS operator in the Type-1 text input to the phase. During translation, the qualifications are examined and one or more PTSAT text tables are generated. A Q-temp. which replaces the based variable reference is created in the final or only PTSAT table.

The use of Q-temps. is illustrated in the following example. The source statement:

```
P -> X = A(I);
```

where

P is a pointer,

X is a based variable,

A(I) is a subscripted array reference,

is translated by Phase II into the following text-table sequence:

PTSAT	S1	X	QT1
SUBS1	I	A	T2
NDX	T2	A	QT3
ASSN	QT3		QT1

The following features are shown in this example:

1. QT1 represents the based variable X, qualified by the pointer P.

2. T2 is a temporary operand which contains the result of the subscript calculation, i.e., the offset of the array element.

3. QT3 represents the Ith element of the array A. QT3 is created by its appearance in the Operand 3 field of the NDX text table. At any subsequent appearance of this Q-temp., it is possible to determine which variable (and which part of the variable) is referenced, by examining the text table in which the Q-temp. was created.

Analysis of Built-in Functions

Each reference to a built-in function is analyzed, and the number and data types of arguments are checked for validity. BIF text tables are generated to contain the arguments and the function result, with a code byte to indicate the built-in function name.

When a built-in function reference is found in the input text, it is stacked in MSTACK with a BIF operator. It is also stacked in SFSTACK and information in this stack is used when checking the validity of built in function references.

Each argument is stacked in MSTACK with a phase-generated BARG operator. When operator-priority comparisons cause the BARG operator-operand pairs to be unstacked they are held in a special save area until the BIF operator-operand pair is unstacked and the validity of all the arguments can be checked.

The routine TGCAFU is used for processing all function references (and CALL statements), but this routine calls the BIFE routine to process built-in function references. BIFE matches the information against a table within the phase called FUNTAB. FUNTAB contains a list of all built-in functions in the PL/I language, together with the number and data type of the arguments that can be used, and the data type of the function result. If it is found that any argument requires conversion, the subroutine ACHE is called to handle the conversion. The subroutine FRED (Function Result Determination) examines the special result descriptor (SRD) in the built-in function operand to determine the data type of the result to be returned by the function.

When all the argument testing and result determination is completed, control is returned to the TGCAFU routine. This routine generates BIF text tables in which the arguments are held in the Operand 1 and 2 fields. If there are more than two arguments, more than one BIF text table is generated. The function result is held in the Operand 3 field of the last BIF table.

Argument and Parameter Matching

Phase II compares each data-element argument with its corresponding parameter descriptor, if one exists. (Phase ID processes data-aggregate

arguments and parameters.) If the function entry point is INTERNAL, or if parameters are described in the declaration of an EXTERNAL entry point, Phase GM will have inserted a PARD operator and a parameter descriptor before the corresponding argument in the text. Argument operands are stacked in MSTACK with ARG operators, and parameter descriptor operands are stacked with PARD operators. When these operator-operand pairs are unstacked, the MATCH routine is called to check the matching of each argument with its corresponding parameter.

If the attributes of an argument and its corresponding parameter are different, a dummy argument is created if possible, together with a WARNING diagnostic message. If the attributes of an argument differ from those of its corresponding parameter to such an extent that conversion of the argument is impossible, then the statement is ignored. The MATCH routine also creates dummy arguments whenever constants are passed as arguments.

Selection of Generic Entry Points

Phase II selects the correct entry point from a generic family by comparing the argument list associated with a generic name with each WHEN descriptor list in turn. The process involves matching of complete argument and descriptor lists as well as matching of individual arguments and descriptors. The argument list is built by making one or more entries in MSTACK; expression arguments are evaluated prior to their inclusion in MSTACK. The SELECT routine is then called to access the WHEN descriptor lists in the general dictionary, chained entries created by Phase GE, and to compare them in turn with the argument list in MSTACK. Individual arguments and descriptors are compared in turn, starting with the first (lowest) entry in MSTACK. The subroutine BARD is called to convert the argument into a 2-byte format, for ease of comparison with the 2-byte descriptors in the dictionary entry.

The first descriptor list that matches the argument list indicates the entry point to be selected. However, it is then necessary to perform the normal argument matching process. The process differs from that described for non-generic argument matching because there are no parameter descriptors in the text, and the appropriate dictionary entries must be accessed. Dummy arguments are generated as required, and ARG text tables are generated as each successful comparison is completed.

Text Handling Features Organized by Phase II

When Phase II translates the main text stream into Type 2 text it incorporates a number of features that enable ease of handling of this form of text by later phases. These features are mentioned briefly here; their usage is described later.

The last four bytes of each 32-byte text table contain two 2-byte chain fields, as shown in figure 5.93. These chain fields can be used to indicate the offset from the start of a page of the preceding or following text table in a logical chain. On output from this phase, the value of these fields is always zero. At the beginning of each page, following the 16-byte page header, a 32-byte overflow-page index table is inserted. This table, the format of which is shown in figure 5.3, is used in connection with the text-table chain fields mentioned above, but is not used by this phase.

The statement header tables which are generated before the text tables that comprise each statement (see figure 5.90), contain three statement-type chain fields. These chain fields are used by Phase KA to

link statements of similar types. Their contents affect the sequence of phase loading, as described in section 3.

To allow for known expansion of the text by later phases, NULL text tables are generated to follow certain types of text tables. For example, DATAE text tables, which form part of some input/output statements, are followed by five NULL text tables. These text tables are overwritten during processing by Phase KQ.

STATEMENT PROCESSING STAGE

The main function of phases grouped in the statement processing storage is to determine the object code that is required to represent the PL/I source program, and to modify the text stream so that the code requirements are clearly indicated to phases in the storage and register allocation stage, and the final assembly stage.

Throughout processing by phases within the stage, the text stream consists of a series of 32-byte text tables (Type-2 text). On input to the stage, the text bears a close relationship to the PL/I source program. On output from the stage, the text bears a close relationship to the machine code required in the object module. One or more text tables in the text output from this stage can indicate one or more machine instructions that must be generated by a code generation phase, or can indicate processing required by phases in the storage and register allocation stage. The following example indicates the type of processing performed by phases in this stage.

Previous processing of the following PL/I source program statements:

```
DCL A CHAR(10), B CHAR(20);
.
.
.
A = SUBSTR(B,2,10);
```

results in the text input to the statement processing stage containing the following text tables:

BIF(SUBSTR)	B	2	null
BIF(SUBSTR)	10	null	A
(Operator)	(Operand 1)	(Operand 2)	(Operand 3)

After processing by phases in this stage, the text contains the following text tables:

OFFS	1	B	Q-temp.n
MOVE	Q-temp.n	10	temp
ASSN	temp	null	A
(Operator)	(Operand 1)	(Operand 2)	(Operand 3)

These text tables indicate to one of the code generation phases that the following machine instruction must be generated for inclusion in the object module:

```
MVC A(10),B+1
```

The stage consists of fifteen phases, Phases IK, KA, IM, IQ, KE, KI, KT, KL, KM, KQ, KV, KK, OC, OX, and KX. Most of these phases process only those text tables that are connected with certain PL/I language features, and ignore the rest of the text. To enable this processing to be carried out most efficiently, the first phase in the stage (Phase KA) examines the whole of the text stream and sets up chains linking statements of similar type. These chains enable some of the phases that process particular types of statements to examine only those statements contained in the relevant chain.

Some phases in this stage are loaded and executed for every compilation. Other phases are only loaded and executed if the PL/I source program contains certain features. Whilst scanning the entire text stream, Phase KA determines which phases are required, and sets up information to indicate the required sequence of phase loading. One phase, Phase IK, is only loaded and executed if either or both of the ATR and XREF

compiler options has been specified. The sequence of phase loading is also affected if OPT(TIME) has been specified. In this case, the four phases of the global optimization stage are loaded and executed after Phase KV, and before any of the last four phases of this stage.

TEXT HANDLING DURING THE STATEMENT PROCESSING STAGE

Apart from differences in the functions of phases, an important difference between phases in the statement processing stage and phase in preceding stages is the methods used for handling text. The Type-1 text format used in the main text stream prior to output from Phase II consists of a continuous stream of text, containing items of various lengths. Each phase reads and processes its input text stream, and builds a new text stream into which items that are not processed by the phase are copied unaltered, and new or modified items are inserted, as the output text stream is built. Phase II translates the text stream into Type-2 text format, which consists of a series of fixed-length text tables. This format allows text tables to be deleted, or new text tables to be inserted in the required logical sequence in the text stream, without completely rebuilding the text stream in each phase.

During translation of the text into Type-2 text format, Phase II sets up chain fields in each text table, and in the overflow-page index table that is created in each text page. As Phase KA reads the text stream it acquires a number of additional text pages, known as overflow pages, at the rate of one overflow page for every four existing text pages. Provision is made for a further seven overflow pages for each existing page to be acquired, one at a time, if necessary. These pages are used to hold any additional text tables that are created in the statement processing stage (and subsequent stages). New text tables created on these overflow pages are linked into the logical sequence of text tables by use of the forward chain field (IFCHN) in the text tables and the ISPLB field in the overflow-page index tables.

If the logical sequence of text tables coincides with their physical sequence, the values in the text table forward chain fields remain zero and the text scanning routines follow the normal sequence of scan. No values are set in text table chain fields to link the last text table on a page to the first text table on the next page in the original text stream. If the content of a text table is to be changed, it is overwritten in situ. If a text table is deleted from usage, this is done by changing its operator code byte to NULL (X'79'). In some cases, e.g., when a series of NULL text tables occurs, the XLINK macro is used to bypass some text tables and move the scan to the next active text table in the logical sequence. This is done by inserting the offset, from the start of the page, of the next logically sequential text table into the last one and a half bytes of the forward chain field of the text table from which the scan is to be stepped.

If a new text table is created, it is inserted in the first available space in an overflow page. The logical sequence of scan is maintained by inserting its offset, from the start of the overflow page, into the last one and a half bytes of the forward chain field of the preceding logical text table. The page containing the new text table is identified indirectly by inserting a value, in the range 8 to F, in the first half-byte of the same forward chain field. This value is used to index the relevant 3-byte field in the ISPLB field of the overflow page index table on the current page. Each 3-byte field can contain the track address of a text page. As the text of any group of four pages can overflow onto up to eight overflow pages, there are eight of these fields, identified by names from REF0 to REF7. The value '8' in the first half byte of a forward chain field indicates that the track address of the relevant page can be found in the REF0 field of the overflow page index table, and the values '9' to 'F' have similar correspondence with the REF1 to REF7 fields. The same system is used to

link from text tables in overflow pages back to text tables in the original text stream pages, or to text tables on other overflow pages. The system is illustrated in figure 2.16.

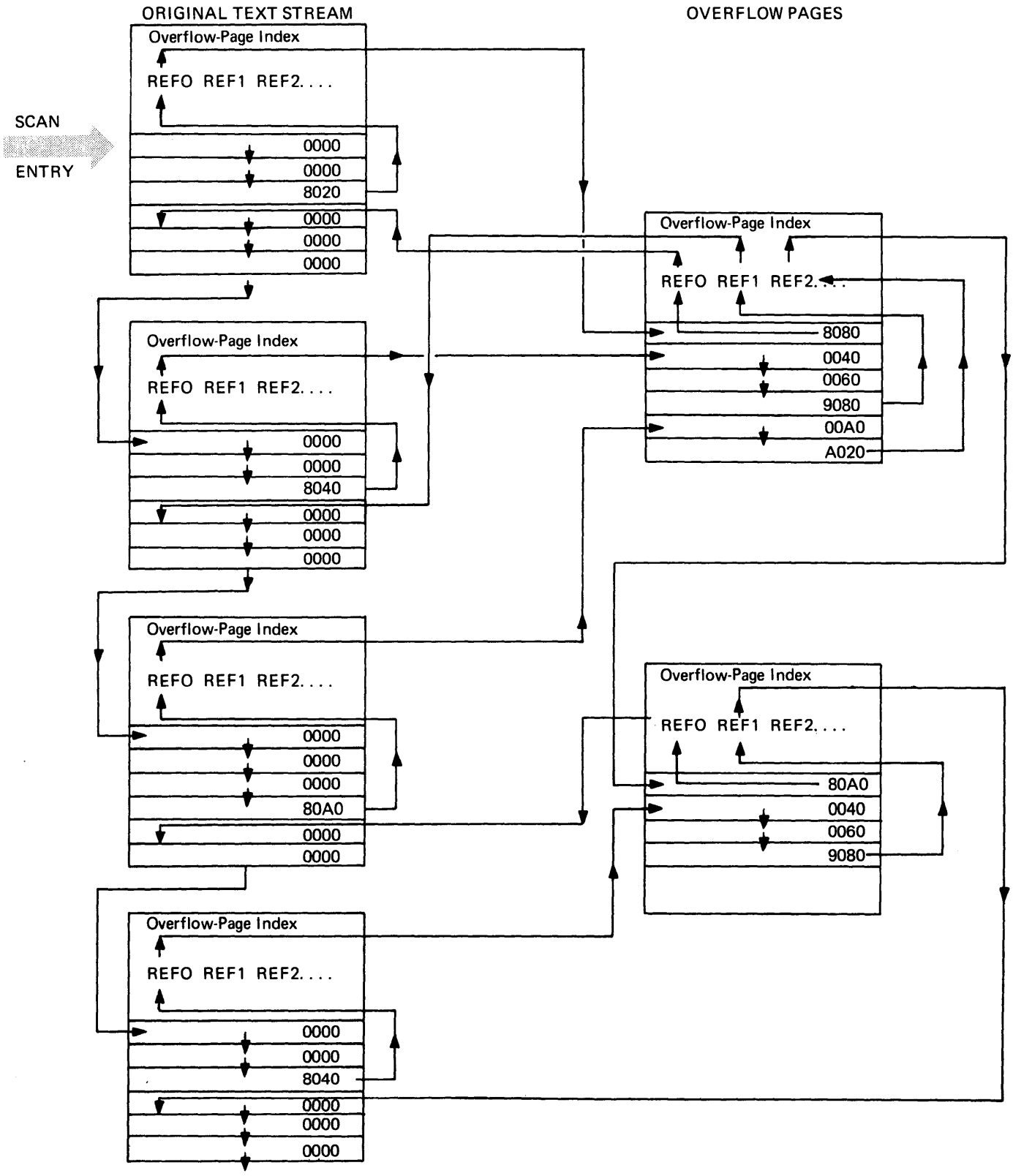


Figure 2.16. Use of overflow pages and chaining of text

ATTRIBUTES AND CROSS-REFERENCES LISTING (PHASE IK)

This phase builds lists of the attributes and/or cross references of all identifiers in the source program, and passes them to the SYSLST data set for printing. The phase is only loaded and executed if either or both of the ATTRIBUTES (A) and XREF (X) compiler options are specified. The listings are organized in EBCDIC collating sequence of identifiers.

PHASE INPUT

Input to the phase consists of the main text stream, in Type-2 text format, and the names dictionary. The variables and general dictionaries are accessed as required.

PHASE OUTPUT

The main output from the phase is a printed listing, consisting of the attributes of each identifier in the program, and/or the number of the source statement in which an identifier is defined and the numbers of the source statements in which the identifier is referred to.

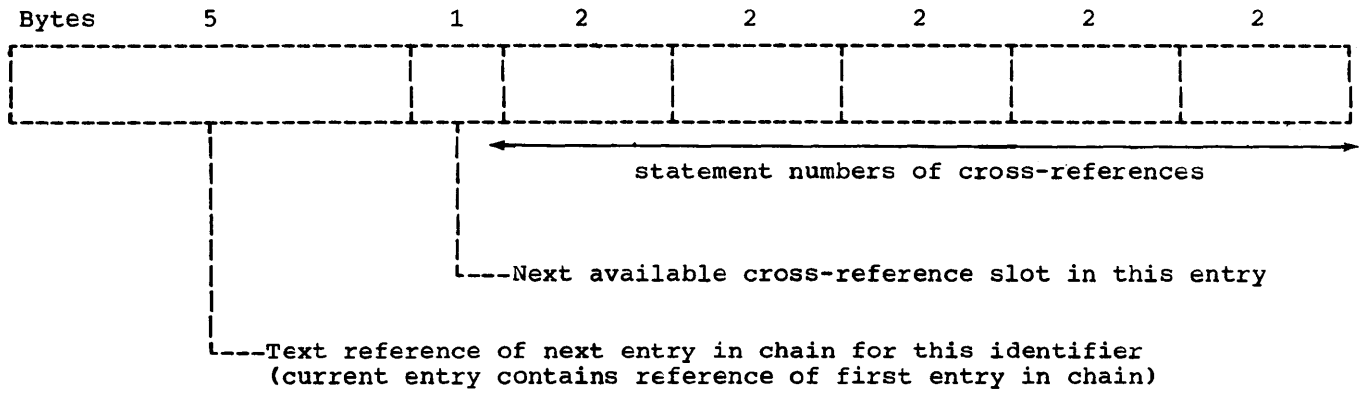
The main text stream is not changed by this phase. If a cross-reference listing is generated, each entry in the names dictionary has an additional 5-byte field inserted, but this additional field is not accessed by any later phases. The variables and general dictionaries are not changed by this phase.

PHASE OPERATION

Collection of Identifier Cross-references

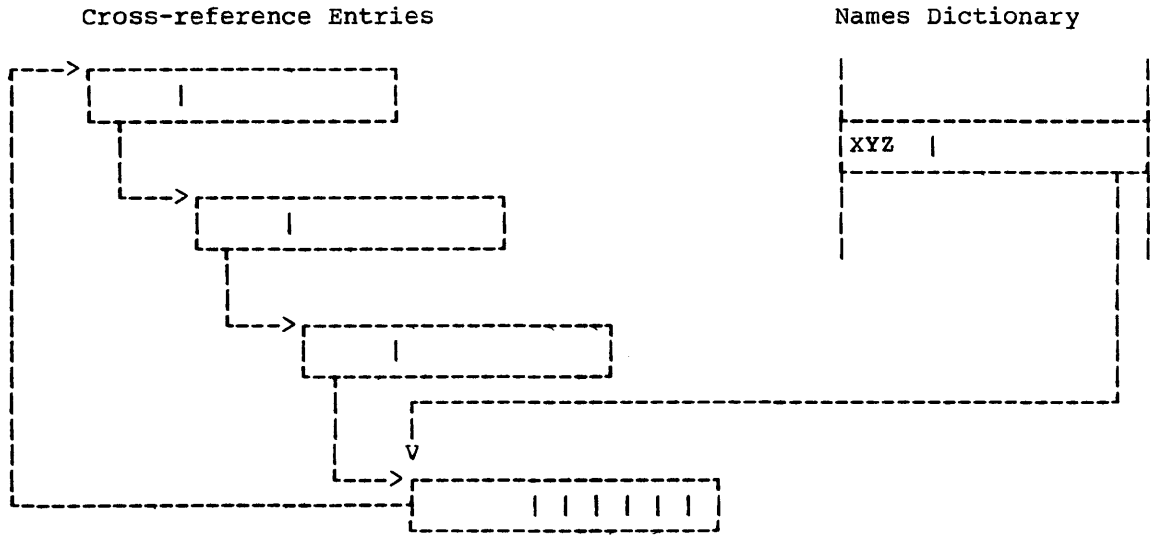
If the XREF compiler option has been specified, a text page, or sequence of text pages, is acquired to hold information collected during processing. Such pages are referred to as cross-reference pages, and they are used only as temporary working storage.

A sequential scan is made of the main text stream. As each reference to an identifier is found, its entry in the names dictionary is accessed. If it is the first reference to that identifier, a new entry is made in a cross-reference page. Each cross-reference entry is 16 bytes long, and has the following format:



An additional item (in the form of a 5-byte text reference) is inserted at the end of each names dictionary entry, to point at the current cross-reference entry for the identifier. When each subsequent reference to an identifier is found, its names dictionary entry is accessed to obtain the reference of the current cross-reference entry, and the statement number of the new reference is inserted.

When a cross-reference entry is full, a new entry is made and the names dictionary pointer is updated. The cross-reference entries for each identifier are chained together, as illustrated below:



The cross-reference pages are saved until the identifers have been sorted into the order required in the output.

Sorting of Identifiers

The routine at IK40 acquires one or more text pages, which are used to hold information required when the identifiers are sorted into the required output sequence (names dictionary entries cannot be sorted). These pages are referred to as sort pages, and are discarded at the end of processing by the phase.

The names dictionary is scanned sequentially. As each entry is scanned, a modified entry is copied into a sort page. Each sort-page entry is 40-bytes long, and contains the identifier name translated from compiler internal code into EBCDIC.

A sorting operation is performed on the sort-page entries. Identifier names are compared, and entries are sorted so that they are eventually organized according to the EBCDIC collating sequence of the names. Where comparison of identifiers shows equality, the shorter name is placed first in the sorted sequence, e.g., ABC is placed before ABCD.

Output of Attributes and Cross-reference Listings

The sorted sort-page entries are scanned sequentially. As each entry is accessed, the identifier is copied to the XPRBF1 or XPRBF2 buffer for output to the SYSLST data set. If the ATTRIBUTES option is specified, the relevant entry in the variables dictionary is accessed, and a list of attributes is copied to output. In the case of data aggregates,

attribute information is also extracted from relevant entries in the general dictionary.

If the XREF option is specified, the entry in the names dictionary is used to access the entries for the identifier in the cross-reference pages. The cross-reference information is output as a series of statement numbers, following the identifier name or the list of attributes. Where there is no continuity in the arithmetic sequence of cross-reference statement numbers, it indicates a change of block level, and is caused by the de-nesting of blocks in the syntax analysis stage.

IF-STATEMENT PROCESSING (PHASE KA)

The main function of Phase KA is the processing of IF and WHEN statements and WHILE and UNTIL clauses. Text tables representing these statements are replaced with conditional-branch tables (BC or BCB). Logical clauses within such statements or clauses are analyzed, and any comparison operations they contain are replaced by conditional branches with appropriate condition codes. This processing is performed during a sequential scan of the text. Other functions performed during this scan include:

- Allocation of overflow pages, at the rate of one to every four existing pages, to allow for later expansion of the text.
- Chaining of certain types of statement, to enable later phases tc find these statements without scanning the whole text stream. The phase also detects language features that affect the loading of some optional phases, and sets bits in XCOMM to indicate the requirement for such phases.
- Resolution of items declared with the UNALIGNED attribute, and modification of DEDs if required.
- Resolution of string temporary operands and string chameleon-temporary operands.
- Detection of situations where on-unit code can be optimized.

PHASE INPUT

This phase examines each item in the main text stream, which is in the Type-2 text format created by Phase II. Entries in the variables and general dictionaries are accessed.

PHASE OUTPUT

On output from the phase, the main text stream is modified as described in following paragraphs. Where additional text is generated it is created on overflow text pages, and linked into the logical sequence by use of chains.

Some statement header tables are modified by the insertion of values in their chain fields, and associated chain fields in XCOMM also have values inserted.

The data element descriptors of some operands in text are completed, and associated entries in the variables dictionary for chameleon-temporary operands are also completed.

PHASE OPERATION

The main text stream is scanned sequentially and every item of text is examined to determine whether it requires processing by this phase. As pages of text are accessed, overflow text pages are allocated, at the rate of one overflow page to every four existing pages, as described in the introduction to the statement processing stage. All required modifications of text are then either performed in situ, by nullification or modification of text tables, or by creation of new text tables on overflow pages.

Creation of Statement-type Chains

As statement headers are examined during the scan of text, certain types are linked to other statement headers of similar type for ease of access by later phases. The chaining addresses are inserted in various fields in the statement header, and the heads of the backwards-pointing chains thus formed are inserted in appropriate fields in XCOMM. The contents of these fields affect the sequence of phase loading as described in section 3. The statement-type chains, and the chaining fields used, are shown in figure 2.17. One feature concerned with the chaining of DO statements is the creation (by Phase II) of dummy statement headers for DO specifications within stream I/O statements. These specifications are thus linked in the DO statement chain.

Statement Type	Chain Field in Statement Header	Chain-head Field in XCOMM
Stream I/O statements	CHAIN1	XSIOCH
Record I/O statements	CHAIN1	XRIOCH
Locator and label variable assignments (plus CALL statements after Phase KV)	CHAIN1	XBELCH
System statements (PROC, END, BEGIN, DISPLAY, DELAY, WAIT, etc.)	CHAIN1	XSYSCH
All statements containing subscripts	CHAIN2	XSUBCH
DO statements (including compiler-generated DO statements)	CHAIN3	XDOCH

Figure 2.17. Statement-type chains created by Phase KA

When the statements are scanned, tests are made for the presence of language features that are processed by some optional phases of the compiler. When a particular language feature is detected, an appropriate subroutine is called to set a bit in the XOPPHS1 field of XCOMM to indicate the requirement for the relevant phase. The indications of bit settings in XOPPHS1 are shown in figure 3.2.

Processing of IF Statements and WHILE Clauses

Note: To improve clarity, the following description refers mainly to IF statements but, unless there is special mention of differences, the description also applies to processing of WHILE clauses.

During the scan of text, the text tables following a statement header for an IF statement are scanned for the IF text table. In some cases, code optimization may be performed during this scan. For example, a source statement of the form:

```
IF X - Y = 0 THEN ....
```

results in the input text containing the following text tables:

```
MINUS     X        Y        temp.
EQ        temp.    0        Bit(1) temp.
```

Detection of the MINUS text table causes the IFMIN routine to be called. This routine changes the text to an effective form of:

```
IF X = Y THEN ....
```

by changing the text tables to the following format:

```
EQ        X         Y         Bit(1) temp.
NULL      -         -             -
```

When the IF table is detected, a register is set to point at it, and the preceding table is examined to determine the type of IF statement as follows:

1. If the preceding table has a comparison operator, i.e., GT, EQ, LT, GE, LE, NE, control is passed to the COMPR routine.

2. If the preceding table has a logical operator, i.e., AND, OR, or NOT, control is passed to the LOGEX routine.

3. If the preceding text table has other than a logical or comparison operator, control is passed to the QUERY routine.

The COMPR Routine: This routine processes IF statements that have a single comparison operator, e.g., IF A>B THEN X = Y; For such a statement, the input to Phase KA would be:

```
SN
GT      A      B      T1
IF      T1     -      CL.1
SN
ASSN    Y      -      X
GSL     CL.1   -      -
```

The COMPR routine processes this type of statement by converting the IF statement to a branch-on-condition statement. The output from this phase would be:

```
SN
BC      A      B      (BNH,CL.1)
NULL    -      -      -
SN
ASSN    Y      -      X
GSL     CL.1   -      -
```

The condition code in the output is the inverse of that for the comparison operator in the input (GT becomes BNH, EQ becomes BNE, etc.). The Bit(1) temporary operands (created by Phase II) are eliminated.

The LOGEX Routine: This routine processes IF statements containing logical expressions, e.g.,

```
IF (A = B) | (C = D) THEN------
```

or IF A|B|C THEN-------

The expression is analyzed in a single backwards scan of the text tables preceding the IF table, using the XTXEN macro. During this scan, a stack is built to determine the significance of any temporary operands. If the expression consists of comparison operations connected by logical operations, the comparison tables are converted to BC tables and the condition codes are set in accordance with the logical operator e.g.,

The source statement:

```
IF (A = B) | (C = D) THEN X = Y;
```

would appear as the following input to Phase KA:

```
SN
EQ      A       B       T1
EQ      C       D       T2
OR      T1      T2      T3
IF      T3      -       CL.1
SN
ASSN    Y       -       X
GSL     CL.1    -       -
```

On output from Phase KA, the statement would appear as:

```
SN
BC      A       B       (BE,CL.2)
BC      C       D       (BNE,CL.1)
GSL     CL.2    -       -
SN
ASSN    Y       -       X
GSL     CL.1    -       -
```

If the expression consists of bit-string variables connected by logical expressions, the subroutine CONVPROC is called to process the expression.

The QUERY Routine: This routine processes IF statements containing a single variable e.g., IF X THEN------, or containing a non-logical expression e.g., IF X+Y THEN------. The subroutine CONVPROC is called to perform the processing.

The CONVPROC Subroutine: This subroutine is called from either the LOGEX routine or the QUERY routine. Its function is to process IF statements that do not contain comparisons or logical operations, or IF statements containing bit-string variables connected by logical expressions. The input to this routine will frequently contain CONV text tables, inserted by Phase II if the variables are not already bit-string variables. The processing by this subroutine depends upon whether variables can be converted in this way. Consider the following examples:

1. IF X THEN------

appears as input to Phase KA in the form:

```
IF      X       -       CL.1
```

where X is the data type specified in the source program.

2. IF A|B THEN------

appears as input to Phase KA in the form:

```
CONV    A       -       T1
CONV    B       -       T2
OR      T1      T2      T3
IF      T3      -       CL.1
```

where A and B are converted if necessary to bit-string variables by Phase II.

In example 1, the variable in the IF text table is examined, and in example 2, the variables in the CONV tables are examined. If the variables can be converted to fixed binary data and thence to bit-string

data without loss of significant digits, a literal constant of zero is generated. The IF table or the CONV table is then replaced by a BC table with the variable as the first operand and the constant as the second operand.

Thus,

```
IF X THEN Y=Z;
```

becomes:

```
BC      X       0       (BE,CL.1)
ASSN    Z       -           Y
GSL     CL.1    -           -
```

and

```
IF A|B THEN P=Q;
```

becomes:

```
SN
BC      A       0       (BNE,CL.2)
BC      B       0       (BE,CL.1)
GSL     CL.2    -           -
SN
ASSN    Q       -           P
GSL     CL.1    -           -
```

If the variables cannot be converted without loss of significant digits, in example 1 a CONV table with a bit-temporary variable as the third operand is generated, followed by a BCB table. This is later expanded by Phase OX to test for bits in the string with a value of 1. Thus,

```
IF X THEN Y=Z;
```

becomes:

```
SN
CONV    X       -       T1
BCB     T1      -       (BZ,CL.1)
SN
ASSN    Z       -       Y
GSL     CL.1    -       -
```

and

```
IF A|B THEN P=Q;
```

becomes:

```
SN
CONV    A       -       T1
BCB     T1      -       (BNZ,CL.2)
CONV    B       -       T2
BCB     T2      -       (BZ,CL.1)
GSL     CL.2    -       -
SN
ASSN    Q       -       P
GSL     CL.1    -       -
```

THEN Clauses: When the processing of an IF statement is completed, the THEN clause is examined. If it contains only a GOTO statement, the GOTOY routine is called to reprocess the code. In general, this reprocessing consists of reversing the condition code of the conditional branch which replaced the IF table, and replacing the compiler generated branch around the THEN clause with a branch to the label specified in the GOTO statement. For example, the statement: IF A>B THEN GOTO L; after processing by the COMPR routine would be:

```
        SN
        BC      A       B       (BNH,CL.1)
        GOTO    -       -       L
        GSL     CL.1    -       -
```

The GOTOY routine replaces this with:

```
        SN
        BC      A       B       (BH,L)
```

If a logical expression in the IF statement causes the generation of more than one BC table, only the last one will have its condition reversed.

e.g., IF A|B & C|D THEN GOTO L;

after processing by the LOGEX routine would become:

```
        BC      A       B       (BNH,CL.1)
        BC      C       D       (BNH,CL.1)
        GOTO    -       -       L
        GSL     CL.1    -       -
```

After processing by the GOTOY routine this would become:

```
        BC      A       B       (BNH,CL.1)
        BC      C       D       (BH,L)
        GSL     CL.1
```

ELSE Clauses: No special processing of ELSE clauses is performed by this phase.

Processing Identifiers Declared with the UNALIGNED Attribute

An identifier that is declared with the UNALIGNED attribute has the third bit of the ZTYP byte of its data element descriptor set to indicate this attribute (see figure 5.59). This attribute is also reflected in the DEDs of Q-temps. and chameleon-temporary operands associated with these identifiers. This attribute is taken into account in processing functions such as argument matching, generic selection, defined matching, etc. For later processing it is necessary to know whether the attribute is really applicable, i.e., whether an operand is really unaligned. If an operand is really unaligned it means that:

1. If it is a bit string it may occupy part of a byte that is also used by another operand, and may not start at the beginning of the byte.

2. The operand may not start on the boundary required for the processing instructions; a situation that could result in a specification interrupt.

Phase KA examines operands which are declared as unaligned, and changes the bit setting in the DED in text if they are not really unaligned. It does not change the DED in the dictionary. Examples of the relevant operands are shown in the following lists:

Variables declared to be unaligned which are NOT really unaligned

1. All arrays.
2. All decimal or character element variables.
3. All bit element variables which are not members of structures or arrays, and which are not defined or parameters.
4. All varying bit strings (note that the length field will not be on a half word boundary if the string is a member of an aggregate, however).
5. All based element bit strings.
6. All temporary strings not derived from built-in functions or pseudovariables.

Variables declared to be unaligned which are REALLY unaligned

1. All bit strings, fixed binary, and float data, of structures or arrays which are not also varying.
2. Element variables which are parameters and not character or fixed decimal.
3. Element bit variables which are string overlay defined.
4. Element variables which are simple defined, or a member of an array or structure, and are not character or fixed decimal.
5. Bit temporary operands derived from SUBSTR built-in function or pseudovariable, where the second argument is not a multiple of 8 or where the first argument is really unaligned.
6. Temporary operands derived from UNSPEC, where the argument is really-unaligned bit string.
7. All based variables that are fixed binary or float.

Resolution of String Temporary Operands

All temporary operands and chameleon-temporary operands that represent string items are examined, and information is determined from related text so that their DEDs can be completed. In the case of chameleon temporary operands, the DEDs in the relevant dictionary entries are also completed.

Detection of Optimizable On-units

ON statements are examined for situations where on-unit code can be optimized. Such situations can occur in statements where the on-unit consists only of a branch to a label-variable or label-constant, e.g., ON condition GOTO label-variable; . If Phase KA detects such a statement, a bit is set (bit 5 in YH2FL, overlayed on YHPOPT) in the general dictionary block-header entry for the ON block. The setting of this bit is tested by Phase KV.

INTERLANGUAGE COMMUNICATION (PHASE IM)

Phase IM performs functions required to enable communication between a PL/I program and FORTRAN, COBOL, or RPGII programs. This facility is required in the following circumstances:

- When a FORTRAN, COBOL, or RPGII program calls a subroutine that has been compiled on the DOS PL/I Optimizing Compiler.
- When a PL/I program calls a subroutine that has been compiled on the IBM System/360 DOS FORTRAN IV F Compiler, or on an IBM System/360 DOS COBCL compiler.
- When a PL/I program uses a data set that has been created by a COBOL program, or when a PL/I program creates a data set that may be used by a COBOL program.

Note: A full list of COBOL, FORTRAN, and RPGII compilers supported can be found in the publications: DOS PL/I Optimizing Compiler: General Information, and DOS PL/I Optimizing Compiler: Installation.

The main problems involved in communication between these different programming languages are:

1. The existence of different data types in the different languages. Phase IM checks the validity of data items involved in interlanguage communication.
2. Differences in the methods by which the different programming languages hold data aggregates in storage. If a data aggregate is involved in interlanguage communication, Phase IM determines whether there is any difference between the methods of mapping used by the two languages involved. If there is, it creates an aggregate temporary operand, assigns the item to it, and indicates the type of mapping required of Phase IQ or a library subroutine.
3. The need for the existence of an operating environment appropriate to a particular programming language, before a program or subroutine written in that language can be executed. Phase IM generates calls to library subroutines that create the required operating environment, or restore the original environment.

PHASE INPUT

Input to the phase consists of the main text stream, which is scanned sequentially, (and also scanned by subsidiary routines), and the general dictionary, in which all entries for external entry points are scanned. Other entries in the general dictionary and in the variables dictionary are accessed as required.

PHASE OUTPUT

This phase modifies the main text stream by deletion or by insertion of new or modified text tables as described in following paragraphs. Entries may be made in the general and variables dictionaries for a block header, entry points, parameters, aggregate temporary operands, and aggregate tables. All relevant newly-generated items are linked to appropriate chains in the text and dictionaries, and chain-header fields in XCOMM are updated if necessary.

PHASE OPERATION

General Considerations

Most of the functions required to enable interlanguage communication are performed by Phase IM, and are only performed if the facility is required. These functions duplicate some of the logical functions performed by other phases, e.g., creation of dictionary entries and text for PROC and ENTRY statements, generation of aggregate temporary operands and aggregate-temporary-operand assignments. Because of the position of Phase IM in the sequence of phase loading, some further duplication of logical function is also required in order to maintain compatibility with the general state of processing, e.g., statement-type chains created by Phase KA must be updated to include text generated by this phase, aggregate operands must be expanded in keeping with the processing performed by Phase IE, etc.

RPG is processed as if it were a COBOL invocation, with the exception of diagnostics which will indicate RPG.

Sequence of Processing

The phase first processes any invocations of PL/I procedures by COBOL or FORTRAN programs. The routine PRCFND examines all general dictionary entries for external entry points, and checks for the presence of the COBOL or FORTRAN option of the OPTIONS option. If one of these options is found on an entry point, the subroutine PRCFFF is called to create a special PL/I procedure block. The subroutine BRG000 is called to generate the required text and dictionary entries, which are later linked into any relevant chains. For each subsequent FORTRAN or COBOL option found, an entry point is created in the interface procedure block.

Control is then passed to the routine TXTSSS to perform any processing required for calls to FORTRAN or COBOL subroutines, or in connection with COBOL files. The entire text stream, with the exception of new text generated by this phase, is scanned. Each statement header is examined for indication of a record-oriented input/output statement, and the text tables of such statements are examined for references to COBOL files. The subroutine REDFND is called to generate any text that is required in such a case. During the main scan of the text, each statement is examined for the presence of text tables indicating a procedure call or function reference. If such an item is found, control is passed to the routine ARG000. This routine determines whether any text is required to provide an interface with a COBOL or FORTRAN subroutine, and generates the necessary text and dictionary entries. On completion, the scan of the statement is continued so that any nested calls or function references can be processed. On completion of the scan of each statement, any relevant chains are updated.

Processing Invocations of PL/I from COBOL or FORTRAN

All entries in the general dictionary for external entry points are examined by the routine PRCFND. If any external entry point is found to have been declared with the FORTRAN or COBOL option of the OPTIONS option, the subroutine PRCFFF is called to create a PL/I procedure block which contains the text required to provide an interface between a PL/I procedure and a COBOL or FORTRAN program. This procedure can be considered as a dummy procedure block which surrounds the actual procedure block required, and is referred to in these descriptions as

the encompassing procedure. It is given the name of the first relevant external entry point found in the dictionary, and is characterized by a block level of one, a block count of zero, and by all its contained text having a statement number of zero. (When Phase GA creates dictionary entries for procedure blocks, the block count of zero is reserved for possible use by Phase IM.)

The subroutine BRG000 is called to generate the text required in the encompassing procedure, and create any dictionary entries that are required. The text that indicates the prologue code for the encompassing procedure includes a call to a subroutine in the library module IBMDIEP. This subroutine sets up an operating environment in which the required procedure can be executed, and restores the FORTRAN or COBOL environment when that execution is complete.

The dictionary references of parameters on the required procedure are stacked in entries in ARGSTX, and NOMAP/NOMAPIN/NOMAPOUT options are analyzed and merged into the stack. For each parameter that is a data aggregate requiring mapping according to PL/I rules, an aggregate temporary operand (and associated dictionary entry) is created, and text corresponding to the following functions is generated:

1. To initialize the locator for the parameter.
2. To reserve storage for a descriptor for the temporary operand.
3. To map the temporary operand according to PL/I rules.
4. To assign the parameter to the temporary operand.

When this processing is complete, text is generated to build an argument list which can be passed to the required PL/I procedure. This argument list contains either the original parameters or the aggregate temporary operands. A call to the required procedure is then generated.

In preparation for the return of control, text is generated to assign each temporary operand back to the original parameter, (unless suppressed by the NOMAPOUT option), to invoke the library routine required to restore the COBOL or FORTRAN environment, and to return control to the calling program.

The scan of entries in the general dictionary is resumed and, for each relevant external entry point, an entry point is created in the encompassing procedure, and processing as described is repeated. Each entry point in the encompassing procedure has the same name as the corresponding entry point in a required procedure. To avoid conflict during linkage editing, entry points in the required procedure are flagged to inhibit the creation of external symbol dictionary entries (ESDs).

If the FORTRAN option is applied to an entry point, and the required procedure returns a data type that is valid in FORTRAN, it is assumed that the entry point can be invoked as a function. In this case, the argument list passed to the required procedure is extended to include a returned value, and modified epilogue code for the encompassing procedure is generated.

Processing Invocations of COBOL or FORTRAN from PL/I

In preparation for invocations of COBOL or FORTRAN subroutines from within a PL/I program, Phase IM generates text to call a library subroutine to set up a COBOL or FORTRAN operating environment, and another call to a library subroutine to restore the PL/I environment, before and after each relevant point of invocation. Text is also

generated to process any arguments as required before the library subroutine is invoked, and after control is returned to the PL/I program. In contrast to the processing performed when a FORTRAN or COBOL program calls a PL/I subroutine, all the text generated to provide the interface facility is contained within the same PL/I block as the procedure call or function reference.

The routine TXTSSS scans the main text stream (except for text previously generated by this phase). If a text table indicating a procedure call or function reference is found, a check is made to see if it corresponds to an external entry point declared with the COBOL or FORTRAN option. If so, preceding text tables within the statement are rescanned to check for the presence of an argument list (indicated by an ALIST text table followed by an ARG text table for each argument). The dictionary references of any arguments are stacked in entries in ARGSTX. Any NOMAP/NOMAPIN/NOMAPOUT options are analyzed and the result is also stacked. The code bytes of the ALIST and ARG text tables are changed to NULL. The arguments are then examined and processed as required to create an argument list in a form acceptable to the COBOL or FORTRAN subroutine.

If an argument is a data aggregate that is mapped differently in PL/I and COBOL or FORTRAN, (and remapping is not suppressed by the NOMAP/NOMAPIN/NOMAPOUT options), an aggregate temporary operand, mapped as required for COBOL or FORTRAN, is generated and the argument is assigned to it. If the argument is already an aggregate temporary operand (created by Phase ID), Phase IM flags it to indicate to Phase IQ that it requires mapping according to either FORTRAN or COBOL rules. To ensure that the temporary operand is allocated storage, mapped, and initialized as necessary, text corresponding to the following functions is generated:

1. To reserve storage for a descriptor for the temporary operand.
2. To initialize adjustable fields in the descriptor.
3. To map the temporary operand according to COBOL or FORTRAN rules. (A special dummy assignment text table, MASSN, is generated to assign each element of the record variable that is a varying-length string to the corresponding element of the record variable.)
4. To assign the PL/I argument to the temporary operand (expanded into elements assignments as required).

When all arguments have been processed as required, an argument list containing the original arguments or the temporary operands (dummy arguments) is generated. The arguments are flagged to indicate that the address of the storage, and not the address of its locator, is to be passed as an argument.

A call is then generated to invoke a library subroutine, in either the IBMDIEC or IBMDIEF library modules, to set up the COBOL or FORTRAN operating environment. This is followed by a call to the required COBOL or FORTRAN entry point. To restore the PL/I operating environment, a call to another library subroutine is generated. The process is completed by assigning the value returned by a FORTRAN function to the appropriate temporary operand, or by assigning the specially created temporary operands back to the appropriate arguments, unless NOMAPIN has been specified.

Processing COBOL Files

If a statement header indicating a record-oriented input/output statement is found during the scan of the main text stream, control is passed to the REDFND subroutine. This subroutine examines the text tables within the statement, and any relevant dictionary entries, to see if they refer to a file declared with the COBOL option of the ENVIRONMENT attribute. If so, the statement is checked to ensure that it is not LOCATE, DELETE, or UNLOCK, and each record variable is examined to determine its validity in PL/I and in COBOL.

Each record variable is also examined for the presence of structures. If structures are not present, no further processing is required. If structures are present, the subroutine PLCBMP is called to determine whether the structure is mapped in a similar manner in both PL/I and COBOL. If the structure is mapped differently in the two languages, text is generated to create an aggregate temporary operand that can be mapped according to COBOL rules, and to assign the PL/I variable to the temporary operand. The text tables generated by this phase indicate whether the record variable and the temporary operand are to be mapped by Phase IQ, or whether they require mapping during execution by a resident library subroutine.

According to the type of statement, and features of the record variable involved, text is generated by this phase corresponding to the following functions:

1. To reserve storage for the temporary operand descriptor.

2. To initialize adjustable fields in the descriptor.

3. To map the temporary operand according to COBOL rules. (A special dummy assignment text table, MASSN, is generated to assign each element of a record variable that is a variable-length string to the corresponding element of the temporary operand.)

4. To insert into the statement the dictionary reference of the COBOL-mapped temporary operand.

5. To assign the PL/I record variable to the COBOL-mapped operand (if the statement is WRITE or REWRITE),

   or

   To assign the COBOL-mapped temporary operand to the PL/I record variable (if the statement is READ).

On completion of processing of the statement, all relevant items in the newly generated text are linked into appropriate chains.

ARRAY AND STRUCTURE MAPPING (PHASE IQ)

The main function of Phase IQ is the mapping of arrays and structures in order to set up the requisite locator-descriptors or, where the mapping cannot be performed by this phase, the generation of text defining inline code for the execution-time mapping of arrays and the generation of calls to library subroutines for the execution-time mapping of structures. The mapping is carried out in accordance with the PL/I algorithm (as described in the data-mapping section of the PL/I Optimizing Compiler Language Reference Manual) or the FORTRAN or COBOL algorithms as applicable.

In addition to aggregate mapping, the phase also performs the following functions:

- Generates text to determine the length of aggregate temporary operands containing strings, where they represent parameters whose length is specified by means of asterisks.
- Generates the necessary library-subroutine calling sequences for the execution of ALLOCATE and FREE statements.
- Generates text to allocate aggregate temporary operands.
- Generates text required to determine the length of a concatenation expression where the final result is a temporary operand.

PHASE INPUT

Initial input to the phase is the general dictionary, in which the chains of aggregate-table entries, partly constructed by Phase GE and linked from the XAGHEDA and XAGHEDS fields of XCOMM, are scanned. When processing of these entries is completed, the main text stream is scanned for MAP, RESDES, ALLOC, FREE, CONCAT, and ASSN text tables. Aggregate table entries in the general dictionary for arrays and structures with adjustable bounds or extents, not linked in the chains mentioned above, are accessed when a reference is seen in the text. Entries in the variables dictionary for aggregate temporary operands are also accessed.

PHASE OUTPUT

On output from the phase, the aggregate table entries in the general dictionary are modified and completed as far as possible. The lengths of aggregate temporary operands are inserted in their variables dictionary entries. The text is modified as described in the following paragraphs.

PHASE OPERATION

During phase initialization, the XAGHEDA field of XCOMM is accessed to find the general-dictionary reference of the first aggregate-table entry in a chain of entries for arrays. The subroutine IQMAP is called to process the information in the entry, and to complete the entry as far as possible (see figure 5.11). The process is repeated for each aggregate table entry in the chain, and for each entry in the chain of entries for structures headed by the XAGHEDS field. These chains, which link aggregate tables entries via their YTCHN fields, include entries

for all aggregates with fixed extents, non-controlled aggregates that are not declared with the CONNECTED attribute, and all based and controlled aggregates (including parameters). Aggregate table entries for aggregates with adjustable extents, for defined aggregates, and for aggregates that are non-controlled parameters declared with the CONNECTED attribute are not linked into these chains; they are processed later when a reference is encountered in the text.

The existing information in each aggregate table entry is used to calculate multipliers and other values required to modify or complete as far as possible the values for size, relative virtual origin, and element offset, and to set flags indicating adjustable values that cannot be calculated during compilation. The entry fields that are completed include YTKEFL, YTHNG (for major structures), YTSZ (for aggregate temporary operands), YTROFF (for arrays not in structures), YTSB, YTBFLG, YTRVO, and YTMULT. (The hang value in YTHNG indicates the number of bytes, at the beginning of storage for an aggregate, that are unused because of alignment requirements.)

The setting of flag bits in the YTSZFL field indicates whether the PL/I, FORTRAN, or COBOL algorithm is to be used for mapping the aggregate. The algorithm used for calculating multipliers according to PL/I requirements is illustrated in the following example. For an array declared as follows:

```
DCL A (LB1:HB1, LB2:HB2, LB3:HB3);
```

the address of an element A(I,J,K) is:

```
Virtual origin of A+I*M1+J*M2+K*M3

    where M3 = element length
          M2=M3*(HB3-LB3+1)
          M1=M2*(HB2-LB2+1)

Virtual origin of A=Start of array-((M1*LB1)+(M2*LB2)+(M3*LB3))
```

The sum of the products of multipliers and lower bounds is called the relative virtual origin of the array. This value is inserted in the YTRVO field of the entry.

In some cases multipliers are later recalculated by Phase KE according to a slightly modified algorithm, in order that more efficient addressing code can be generated.

When the scan of the chained aggregate table entries is complete, control is passed to the routine SC0, which scans the main text stream for text tables of interest to the phase and branches to appropriate processing routines as follows:

```
MAP    text tables - processed by routine SC2
RESDES text tables - processed by routine SC25
ALLOC  text tables - processed by routine SC14
FREE   text tables - processed by routine SC26
CONCAT text tables - processed by routine SC60
ASSN   text tables - processed by routine SC91
```

The main scanning routine also scans for OFFS text tables that are associated with structure or array descriptors. If the descriptor applies to an aggregate temporary operand, a controlled aggregate, or a based adjustable aggregate, the reference to the aggregate table in the second operand of the OFFS text table is replaced by the temporary descriptor from the associated RESDES text table. If this descriptor is a dimensioned member of a structure, the offset value in operand 1 is incremented by four bytes to allow for the offset field in the structure descriptor.

Processing MAP Text Tables

All MAP text tables in the text input to Phase IQ have an IOP2 code of X'00'. When one of these text tables is seen, its aggregate table entry in the general dictionary is accessed. If the entry is included in the chains linked from either the XAGHEDA or XAGHEDS field in XCOMM, the aggregate will have been mapped as far as is possible at compiler time. If an entry shows that the aggregate has been completely mapped, the MAP text table is deleted. For all other aggregates, the aggregate table entry is completed, as far as is possible at compile-time by the subroutine IQMAP. Text is then generated to complete the mapping at execution time, either during execution of prologue code or on reference to the variable during execution of main code. For structures, mapping at execution time is performed by a library subroutine, and this phase generates text defining the required calling sequence. For arrays, this phase generates text-defining code to perform the mapping inline.

If a MAP text table refers to a temporary structure or array with one or more string lengths specified by means of asterisks, a call is made to the subroutine CALLEN. This subroutine scans the text following the MAP table for string-handling operations that affect the length of the string to be assigned to the temporary operand. Information derived from these operations is stacked and used to determine the maximum length of each element. Where the information is insufficient to enable the subroutine to determine this length at compile-time, text is generated to indicate the code required to determine the length during execution.

If a MAP table refers to a structure that cannot be completely mapped at compile time, the subroutine GENSDD is called. This subroutine uses the XCADD macro to construct a structure-descriptor-descriptor for the variable (see figure 5.122) and to insert it in an entry in the general dictionary. The subroutine GENLIB is then called to generate text tables defining a calling sequence to the structure-mapping library subroutine; the reference of the structure-descriptor-descriptor is included in the argument list. If the structure is based and has adjustable extents it will require mapping on reference during execution. In such a case, a call to another entry point in the library structure-mapping module is generated.

If the MAP text table refers to an array that cannot be completely mapped at compile-time, text is generated defining inline code required for mapping during execution. Unless the item is a FORTRAN array, (in which case a MAP04 text table is generated), the input MAP00 table is replaced by another MAP00 table, defining code to be generated for mapping during execution.

Processing RESDES Text Tables

A RESDES text table indicates that temporary storage is to be reserved for the descriptor of an aggregate that is a temporary operand, is controlled, or is based with adjustable extents. The RESDES text tables are created by Phases IA and ID. Routine SC25 examines the aggregate table for the variable specified in the RESDES table. If the aggregate has fixed extents, (a RESDES table is generated by Phase ID for each aggregate temporary operand that it creates,) the RESDES table is deleted. Otherwise, the required size of the descriptor is determined, and the length is inserted in the third operand of the text table. If the aggregate is a temporay operand, text defining the code required to move the address of the temporary descriptor into its locator is also generated.

The routine also generates a RESDES text table for any controlled aggregate that has extents specified by asterisks but has no other adjustable extents.

Processing ALLOC Text Tables

When an ALLOC (or LOCATE) text table is seen, routine SC14 examines the allocated variable. If the extents of the variable are not known at compile time, the ALLOC table will be preceded by a MAP text table. In cases where the extent of the variable is indicated by a single asterisk, (e.g., ALLOCATE A(*);), subroutines are called to generate the requisite MAP text table. The length of the variable determined by the MAP table is used as the basis for the length of the required allocation.

If the variable is controlled and requires a locator (i.e., if it is an aggregate, a string, or an area), the length of the locator is added to the length of the required allocation. If the variable also requires a descriptor, the length of the descriptor is added to the allocated length if the variable has adjustable extents.

The subroutine GENLIB is called to generate text tables defining a calling sequence to a library subroutine to perform the allocation. The library subroutine called is one of three, depending on whether the allocated variable is controlled, based and allocated in an area, or based but not allocated in an area.

Processing FREE Text Tables

FREE text tables are processed by routine SC26 in a manner similar to the processing of ALLOC text tables. The conditions affecting selection of the library subroutine to be called are the same.

Processing CONCAT Text Tables

All CONCAT text tables are examined by routine SC60, but processing is only performed if the result (third) operand is a string-address temporary operand.

If the operands are bit strings, processing consists of the generation of a GETVDA text table to indicate the code required to acquire storage for the result. If the operands are character strings, a read-ahead scan is made to see if the string-address temporary operand result is used in another CONCAT text table. If it is, the result operand of the first CONCAT text table is replaced by the result operand of the second table. The process is repeated until a CONCAT text table is found in which the result is not a string-address temporary operand, in which case control is returned to the main scanning routine, or until the result operand is found to be used in a text table other than a CONCAT text table. In this case, the CALLEN subroutine is called to analyze the expression in which the string operands are used, and to generate text defining the code required to calculate the maximum length of the result of the final concatenation operation. Control is returned to the main processing routine and a VDA(00) text table for the calculated length is generated.

Processing Assignment Text Tables

Routine SC91 examines all text tables involving assignment. If the result operand is a string-address temporary operand that is not derived from a pseudovariable, a VDA(00) text table is generated to acquire storage for the result.

Calculation of Expression-result Lengths (The CALLEN Subroutine)

The CALLEN subroutine is called from a number of routines in the phase, to analyze string expressions and to determine the maximum length of the target to which the expression is assigned. Examples of this are when arrays or structure members have extents specified by asterisks, or when a concatenation operation is assigned to a temporary operand result.

The subroutine makes two scans of the string expression. During the first scan a stack is built, in which an entry is made for each operation in the expression that has a temporary operand result, or a result that is a Q-temp. derived from a temporary operand that is being processed. Each entry consists of three 2-byte fields, representing respectively operand 3 (result operand), operand 1, and operand 2. The identifying number of the result temporary operand is inserted in the first field. If operand 1 or operand 2 is an intermediate string temporary operand, its identifying number is inserted in the appropriate field; otherwise, the field is set to X'FFFF'. The stack is then scanned recursively, starting at the entry for the final result operand. For each operand in the stack that is directly involved in the expression, bit zero of its stack entry is set.

The string expression in the text is then scanned again. As each text table representing a string operation is seen, a search of the stack is made to see if the result operand has been flagged as a significant temporary operand. If it has, a length calculation is made according to the type of operation the operand is involved in, as shown below:

Operation type	Basis for length calculation
SUBSTR bif	length of operand 1
REPEAT bif	(length of operand 1+1) * operand 2
NOT, ASSN, CONV	length of operand 1
CONCAT	length of operand 1+length of operand 2
AND, OR	MAX (length of operand 1, length of operand 2)
STRING, bif	a call to a library subroutine is generated to determine the maximum length.

(If any operand has a varying length, the maximum length is used.)

If the lengths of the operands are known at compile time, the calculated length is set as a constant value in the operand 1 position in the stack entry, and the operand 2 position is then used as a switch that is set to indicate the presence of the constant value. If the lengths of the operands are not known at compile time, text tables are generated which define the code required to calculate the lengths at execution time. If the text table contains a temporary operand that is generated to hold the length of an intermediate result, the identifying number of the temporary operand is inserted in the operand 1 field of the appropriate stack entry, and the operand 2 field of that entry is used as a switch, which is set to indicate a variable result.

When the scan of the expression in the text reaches the last text table in the expression, either the maximum constant length of the result is known, or a temporary operand has been generated to hold the required length value. A VDA(00) text table for the required length is then generated.

SUBSCRIPT PROCESSING (PHASE KE)

The main function of this phase is the processing of subscripts. This involves evaluation of constant subscripts, and the provision of information that enables variables subscripts to be evaluated during execution. In addition to this processing, the phase performs the following functions:

1. Generates text indicating optimized aggregate-assignments by use of MVC instructions.
2. Inserts array bounds information in compiler-generated do-loops.
3. Applies base array information to iSUB-defined items.
4. Processes INITIAL assignments to arrays.
5. Checks for occurrences of the SUBSCRIPTRANGE condition or, if this cannot be done during compilation, enables the condition to be tested during execution.

PHASE INPUT

Input to the phase consists of the main text stream in Type 2 text format. Only those statements in the XSUBCH chain created by Phase KA are examined.

Entries in the variables and general dictionaries are accessed to obtain information about arrays.

PHASE OUTPUT

Output from the phase consists of the main text stream, in which text tables have been deleted or modified, or new text tables generated, as described in the following paragraphs. No dictionary entries are made by this phase.

PHASE OPERATION

Sequence of Processing

The main scanning routine uses the XTCH macro to scan statements in the XSUBCH chain. As each statement header is found, the statement-type code byte is examined. If this code byte is SOASSN (X'7C'), indicating an optimizable array or structure assignment, control is passed to the MVC1 routine to process the assignment. When this processing is complete, the scan is moved to the next statement.

If the statement is not SOASSN, the prefix-option bytes in the statement header are examined. If these indicate that the statement contains a compiler-generated do-loop, the text tables of the statements are examined and any do-loops that are found are processed by the DOIT0 routine.

When any do-loop processing that may be required is complete, the prefix-option bytes are further examined for indication of iSUB definition in the statement. If this is indicated, the text tables of

the statement are again scanned, and iSUB defined items are processed by the DEF0 routine.

When any do-loop or iSUB-definition processing that is required is complete, the text tables of the statement are scanned by the LABA routine, which processes any subscripted items indicated by SUBS1 and SUBS text tables. If, during this scan, any IASSN or AID text tables (indicating array INITIAL assignments) are found, then a flag is set to indicate that initial processing is required. During the processing of subscripts, the SUBRGSUB subroutine is called to check for the occurrence of the SUBSCRIPTRANGE condition, or to generate text to enable the condition to be checked during execution. When the end of the statement is reached, the array initial flag is tested and a call made to the INIT0 routine if array initial is present. The main scanning routine is then called to get the next statement.

Optimized Aggregate Assignments

Phase IE determines whether assignments of arrays and structures must be expanded into individual element assignments, or whether the whole aggregate can be assigned without expansion. Where expansion is not necessary, it indicates this fact by creating an SOASSN statement. When an SOASSN statement is found, the routine MVC1 is called to generate text for the optimized assignment.

The processing involves accessing the appropriate aggregate table entries in the general dictionary to determine the virtual origin of each aggregate. This enables the creation of Q-temps., between which assignment can be made. This processing is indicated in the following example. The source statements:

```
DCL A(10),
    B(10) FIXED BIN;
 .
 .
 .
A = B;
```

result in the following text tables being generated by the phase:

```
OFFS     V.O. of A       A        Q-temp1
OFFS     V.O.of B        B        Q-temp2
MOVE     Q-temp2      length      Q-temp1
```

Processing Compiler-generated Do-loops

Examination of the prefix-option bytes in the statement header will indicate whether the statement contains subscript tables that are surrounded by do-loops created by Phase IE. If so, bounds information must be inserted before the do-loop can be processed by Phase KI.

The required processing is performed by the DOIT0 routine. Appropriate aggregate-table entries in the general dictionary are accessed to obtain information about the lower and upper bounds of arrays, which is inserted into the first and second operand fields of the ITDO text table. In a case where an asterisk subscript is used, this is replaced in the subscript text tables by the lower-bound value. The subscript text tables are later processed by the LABA routine.

Processing iSUB-defining Text Tables

In cases of iSUB-defining, the subscript tables for the defined item are always preceded by text tables for the base array. These text tables are DSUBS1, DSUBS, and DSUBSL tables, preceded by a DLST table.

The DEF0 routine converts the subscripts of the defined item to those of the base array, and substitutes them in the base array text tables. The text table operators are changed, and some text tables deleted, as shown in the following example. The source statements:

```
DCL A (5,6),
    B (4) DEFINED A (1SUB,1SUB);
    B (2) = 1;
```

will result in the following text tables being input to Phase KE;

Base Array	DLIST	B	A	-
	DSUBS1	1SUB	A	T1
	DSUBSL	1SUB	A	T1
Defined Item	SUBS1	2	B	T1
	NDX	T1	B	Q-temp1
	ASSN	1	-	Q-temp1

After processing by the DEF0 routine, the modified text tables appear as follows:

SUBS1	2	A	T1
SUBS	2	A	T1
NDX	T1	A	Q-temp1
ASSN	1	-	Q-temp1

During this processing, the SUBRGSUB subroutine is called to check for occurrence of the SUBSCRIPTRANGE condition in the defined item. Occurrence of this condition in the base array is checked later, when the subscripts are being processed by the LABA routine.

Processing Subscripts

The LABA routine processes all subscripts in the text as modified by earlier processing performed by the phase. To do this it scans all text tables in the statement.

For each reference to a subscripted variable (i.e., in a SUBS1 or SUBS text table) Phase KE provides information which enables the offset of that variable, from the actual origin of the array in which it is contained, to be calculated. The information provided by this phase is in the form of subscript multipliers. If the subscripts themselves are variables, then the subscript multipliers are inserted in the second operand fields of the appropriate SUBS1 or SUBS text tables. This enables code to be generated to calculate the required offset during execution. If the subscripts are constant values, then Phase KE calculates the required offset and generates an OFFS text table to replace the appropriate SUBS1, SUBS, and NDX text tables.

Where the elements of an array occupy contiguous areas of storage, and do not have adjustable bounds, the LABA routine calculates the value of each subscript multiplier, using the expression:

```
Multiplier=Hbound-Lbound+1
```

and the length of each array element.

If the subscripts are constant values, the routine calculates the offset of the element, using the expressicn:

```
Element Offset=Array Virtual Origin+([LPAR*N] S1*M1+S2)
              *M2+SI)*MI+SI)*MN)
```

where S = subscript (and SI is the Ith subscript),
M = subscript multiplier (and MI is the Ith subscript-multiplier),
and N = the number of dimensions of the array.

In this expression, the value of each multiplier other than the first is dependent upon the values of preceding multipliers.

Processing performed by the phase is illustrated in the following example.

The statements:

```
DCL A(7,2) FLOAT DECIMAL;
    .
    .
    .
    A(3,2)=1;
```

would result in the following text tables being input to Phase KE:

SUBS1	3	A	T1
SUBS	2	A	T1
NDX	T1	A	Q-temp1
ASSN	1	-	Q-temp1

The subscript multipliers are calculated and, because the subscripts are constant values, the offset of the element is calculated. Output from the phase is:

OFFS	32	A	Q-temp1
ASSN	1	-	Q-temp1

If the subscripts are variables, as in the statement:

```
A(I,J)=1;
```

the input to the phase contains:

SUBS1	I	A	T1
SUBS	J	A	T1
NDX	T1	A	Q-temp1
ASSN	1	-	Q-temp1

Phase KE would not be able to evaluate the offset, but would calculate the subscript multipliers and modify the text so that later phases could generate code to enable the calculation to be made during execution, using the same algorithm. The cutput text would be:

SUBS1	I	2	T1
SUBS	J	4	T1
NDX	T1	A	Q-temp1
ASSN	1	-	Q-temp1

Where the elements of an array are not contiguous in storage, Phase KE does not calculate the subscript multipliers, but uses the multipliers calculated by the aggregate mapping phase (Phase IQ) and inserted in the aggregate table entry in the general dictionary. The expression used to calculate the element offset differs from that used for contiguous arrays, and is

```
Element Offset=Array Virtual Origin+(S1*M1+S2*M2---SI*MI+SN*MN)
```

where the symbols are the same as previously described. In this expression, the value of each multiplier is independent of the values of preceding multipliers. The text tables output are similar to those previously described, but the second operator code byte is set to indicate to phases in the register allocation stage that registers must be allocated according to the modified algorithm.

Processing the SUBSCRIPTRANGE Condition

When the LABA routine is processing subscripts, it calls the SUBRGSUB subroutine to check for situations where the SUBSCRIPTRANGE condition would be raised during execution. This subroutine is also called during processing of iSUB defined items, when the defined array is checked for this condition. The base array is checked during processing of subscripts.

Where subscripts are constants and the bounds are known, situations where the SUBSCRIPTRANGE condition would be raised can be checked by this phase. The checking is performed in all such cases, regardless of whether the condition has been enabled by the source programmer or not. A diagnostic message of the SEVERE ERROR type is generated for each relevant situation.

If a subscript is a variable, or if the bounds of an array are not known during compilation, text indicating the code required to check the condition during execution is generated. This text is only generated for situations where the SUBSCRIPTRANGE condition is enabled. The text consists of conditional branch text tables and compiler-generated labels, as illustrated in the following example.

If the SUBSCRIPTRANGE condition is enabled, the following source statements:

```
DCL A(M:N);
.
.
A(I) = 1;
```

will cause Phase KE to generate the following text tables:

```
BC          I          M      BL,CL.1
BC          I          N      BNH,CL.2
GSL         CL.1       -      -
SIGNAL      SUBRG      -      -
GSL         CL.2       -      -
SUBS1       I          A      T1
```

Processing Array INITIAL Assignments

The input to the phase for array items with the INITIAL attribute will be similar for either contiguous arrays or interleaved arrays. The routine INIT0 determines the environment of the array.

For contiguous arrays with constant repetition factors, initial values will be assigned by the generation of MOVE tables.

For contiguous arrays with variable repetition factors, a flag is set in the XCOMSTR field, in XCOMM, to indicate that a compiler-generated subroutine is required to generate move instructions at execution time.

For interleaved arrays, initial assignment is made by addressing each element in turn and then making the appropriate assignment.

DO-STATEMENT PROCESSING (PHASE KI)

This phase processes all iterative forms of the DO statement, DO specifications in the data lists of stream-oriented input/output statements, and do-loops resulting from array assignments.

Processing consists of replacing or deleting Type-2 text tables in the main text stream, and inserting additional text tables where necessary, so that the output text indicates the machine code that must be generated for inclusion of these items in the object module.

During processing, indications of possible optimization may be inserted for use by phases in the global optimization and register allocation stages.

PHASE INPUT

Input to the phase consists of Type 2 text tables that indicate the PL/I format of the source program. Phase KI examines all statements in a chain built by Phase KA, the head of this chain being in the XDOCH field in XCOMM. Pointers in the chain enable the following items to be accessed:

- Statement headers -- SN or SL tables -- for iterative DO statements.
- Dummy statements headers -- GSN tables -- generated in the case of do-loop specifications preceding array assignments in input/output statement data lists.
- DO3 text tables, generated in connection with the END statements of do-loops.
- DO1 text tables, generated with DO2 text tables during the translation of DO statements by Phase II.

DO1, DO2, and DO3 text tables contain the following operands:

DO1 tables

Operand 1. Null.

Operand 2. The reference of a compiler generated label indicating either the start of the next loop specification in a multiple loop specification, or the DO3 table if the DO1 table is either the only loop specification or the last of a multiple loop specification.

Operand 3. The loop control variable (lcv).

DO2 tables

Operand 1. A constant or a temporary variable indicating the value of the TO clause.

Operand 2. A constant or a temporary variable indicating the value of the BY clause.

Operand 3. A statement label reference, or a compiler-generated label, indicating the first statement in the body of the DO loop.

DO3 tables

Operand 1. A compiler-generated label indicating the first statement after the end of the do-loop (with only one operand).

If a do-loop specification contains a WHILE clause, the output from Phase II contains a WHILE text table or, if there is no TO or BY clause in the specification, a DOWHYL text table. These tables are preceded by a GSN table, which is included in the IF chain, and they are therefore processed by Phase KA. On input to Phase KI, the WHILE and DOWHYL tables appear as a branch table or series of branch tables (preceded by a GSN table), indicating flow through or around the do-loop, according to the evaluation of the expression in the WHILE clause.

If a do-loop specification contains an UNTIL clause, the output from Phase II contains an UNTIL text table preceded by a GSN table which is included in the IF chain and thus processed by Phase KA. On input to Phase KI, the UNTIL tables appear as a series of branch tables. These tables indicate flow through or out of the do-loop according to the evaluation of the expression in the UNTIL clause. The tables are delimited by the GSN table at the front and by the UNTIL table at the end of the clause.

If a do-loop specification contains a REPEAT clause, the output from Phase II contains a REPT table, preceded by a GSN table. On input to Phase KI, the REPEAT clause appears as a series of evaluations, ultimately to the loop control variable, delimited by a GSN table at the front and by a REPT text table at the end of the clause.

Each pair of DO1/DO2 tables is preceded by an ASSN table, in which an initial value is assigned to the loop control variable. A CONV table may appear instead of an ASSN table if the data types of the loop control variable and its initial value differ. If the values in the TO or BY clauses are not constants, ASSN or CONV tables that assign these clause values to temporary variables also appear before the do-loop tables. The dictionary is not accessed by this phase.

PHASE OUTPUT

Output from the phase is in the Type-2 text format. Text tables associated with do-loops are reordered and/or replaced by tables that have a direct correspondence with machine instructions. Types of tables that are commonly used for replacement include:

BC	branch-on-condition table
GOTO	unconditional-branch table
SCI	set-comparand-and-index table
DINC	increment-DO-loop-control-variables table

Typical usage of these tables is shown in the examples used to illustrate phase operation.

PHASE OPERATION

The CHAINDO routine uses the XTCH macro to scan the DO statement chain. When the appropriate statement header tables are found, the SCAN1 routine examines tables within the statement for DO1, GSN (WHILE clause), and CONV tables, and calls appropriate routines and subroutines to analyze and process them.

In general, if the initial and final values of a loop specification are constant values, the conditions for branching are tested at the end of the loop. If the initial and final values of a loop specification are not constant values, a BC table is generated at the head of the loop to test whether the body of the loop should be entered or branched around. If there is more than one specification, the BC test for the first specification is generated at the head of the loop, and tests for the

remaining specifications are generated immediately after the DO3 table at the end of the loop.

WHILE, UNTIL, or REPEAT Clause

WHILE: Having recognised that a WHILE clause is present, Phase KI ensures that the GSN at the front of the clause becomes a GSL01, and that this label is the target of the branch/test at the end of the loop. In the case of an iterative do-loop specification, Phase KI will ensure that the GSL01 label, already inserted in front of the loop body, is deleted.

UNTIL: Having recognised that an UNTIL clause is present, Phase KI moves the clause from the front of the loop to a point immediately after the loop body. It ensures that the target of the branch/test at the end of the clause is the label at the front of the loop preceding a WHILE clause if present.

REPEAT: Phase KI recognises that the specification contains a repeat clause and moves it from the front of the loop to a point immediately after the loop body, or immediately after an UNTIL clause if present. Phase KI then generates an unconditional branch to the front of the loop, or to the WHILE clause if present.

Multiple Loop Specifications

If multiple specifications are provided for a do-loop, a temporary label variable is used to indicate which loop specification is being used at any particular time during execution. The label variable, which appears before the loop, is given an initial value corresponding to a label constant that appears before the loop increment code at the end of the loop. When execution of the first loop specification is completed, the label variable is re-initialized to correspond with the label constant in front of the increment code for the second specification. The process is repeated for each specification.

Thus, a loop that is specified as follows:

```
DC I=1 to N, 20 to M;
.
.
.
(body of loop)
.
.
.
END;
```

would appear in the input to Phase KI as:

```
ASSN      N         -       tN        Text tables
ASSN      1         -       I         for first
DO1       Null    CL.1      I         specification.
DO2       tN        1       CL.2
GSL       CL.1      -       -
ASSN      M         -       tM        Text tables
ASSN      20        -       I         for second
DO1       Null    CL.3      I         specification.
DO2       tM        1       CL.2
GSL       CL.2      -       -         Loop head table.
     (body of loop)
DO3       CL.3      -       -         End of loop.
```

After processing Phase KI, the output from the phase would appear as follows:

```
ASSN    N       -       tN          TO clause value and control
ASSN    1       -       I           variable initialization.

ASSN    CL.4            tLV         label variable set to label
                                    on 1st specification
                                    increment code.

SCI  (see following paragraph)
BC      I       tN      (BH,CL.1)   Test to determine whether
                                    loop to be entered for
                                    1st specification.

GSL     CL.2    -       -           Loop head table.

(body of loop)

GOTO    -       -       tL
                                    Branch to current
                                    specification
                                    increment code.

GSL     CL.4    -       -           Label of 1st
                                    specification
                                    increment code.

DINC    I       1       I           Increment table
BC      I       tN      (ENH,CL.2)Test for branch back
                                    through loop or fall
                                    through to next
                                    specification.

GSL     CL.1    -       -
ASSN    M       -       tM
ASSN    20      -       I           Code to initialize
ASSN    CL.5    -       tLV         2nd loop.
SCI  (see following paragraph)
BC      I       tM      (ENH,CL.2)
GOTO    -       -       CL.3

GSL     CL.5    -       -           Label of 2nd specification
                                    increment code.
DINC    I       1       I           Increment table.
BC      I       tM      (ENH,CL.2)Test for branch back
                                    through loop or
                                    fall through to END
                                    statement.
GSL     CL.3    -       -           End of loop label.
```

Optimization Indication

If the loop control variable does not require conversion before it is incremented and compared with the TO clause value, an SCI (set comparand and index) table is generated in the loop header code. This table indicates to phases in the global optimization and register allocation stages that optimization, by use of BXLE instructions, is possible. If loops are nested, an SCI table is generated for the inner loop only. If the foregoing conditions are satisfied, SCI tables are generated for the initialization of every multiple specification.

Variable TO and BY clauses

It neither the TO nor the BY clauses in a loop specification contain constants, the compiler does not know whether the increments are positive or negative. Consequently the condition code for the end of the loop cannot be determined. The problem is overcome by using a label variable in a similar manner to that used for multiple loop specifications.

Tables are inserted in the loop-head code to test the direction of looping, and to assign a label constant to the label variable accordingly. At the end of the loop, and after the loop control variable has been incremented, a GOTO label-variable table is used to transfer control to the appropriate test for the end of the loop.

For example, the loop specification:

```
DO I = J TO K BY L;
```

would appear as input to Phase KI as follows:

```
ASSN        J        -          tJ
ASSN        K        -          tK
ASSN        L        -          tL
ASSN        tJ       -          I
DO1         Null     CL.1       I
DO2         tK       TL         CL.2
GSL         CL2      -          -
(body of loop)
DO3         CL.1
```

After processing by Phase KI output from the phase would appear as follows:

```
ASSN      J       -        tJ
ASSN      K       -        tK
ASSN      L       -        tL
ASSN      tJ      -        I
BC        tL      0        (BL.CL.3)      Branch if BY clause
                                          value is negative.

ASSN      CL.4    -        tLV            Initialize label variable
                                          for positive value
                                          BY clause.

GOTO      -       -        CL.4
GSL       CL.3    -        -
ASSN      CL.5    -        tLV            Initialize label variable
                                          for negative value
                                          BY clause.

GOTO      -       -        CL.5
GSL       CL.2    -        -              loop head label
   (body of loop)

DINC      I       tL       I              Increment control variable
GOTO      -       -        tLV            Branch to appropriate
                                          end-of-loop test.
GSL       CL.4    -        -              Non-negative-value
BC        I       tK       (BNH,CL.2)     BY clause end-of-
GOTO      -       -        CL.1           loop test.

GSL       CL.5    -        -              Negative value BY
BC        I       tK       (BNL,CL.2)     clause end-of-
GSL       CL.1    -        -              loop test.
```

Do-loops in Array Assignments

Do-loops that result from array assignment appear in the input to Phase KI as follows:

```
ITDO      lowbound      highbound      temporary
                                       control variable

(body of loop)

ENDIT     -             -              temporary
                                       control variable.
```

Phase KI processes such loops as if the input were as follows:

```
ASSN      lowbound      -              temp.
DO1       Null          CL.1           temp.
DO2       highbound     1              CL.2
GSL       CL.2
(body of loop)
DO3       CL.1
```

If the array assignment has more than one dimension, Phase IE expands the statement into nested pairs of ITDO-ENDIT tables. When Phase KI expands the array assignment loops, the subscripts of the assignments within the loops are examined. If the subscripts have multipliers with a highest common factor that is not 1, the looping increment is set to this H.C.F. and the bounds are modified accordingly. This process minimizes the number of multiplication operations necessary within the loop.

SYSTEM-INTERFACE STATEMENT PROCESSING (PHASE KT)

Facilities provided by the system control-program are required to enable the execution of certain types of statements. Examples of these types of statements are input/output statements, which are processed by particular phases of the compiler. Phase KT processes other statements requiring system facilities, such as DISPLAY, SIGNAL, DELAY, and WAIT statements. It generates text indicating the code and/or call to a library subroutine required to provide the interface with the system.

In addition to these statements, Phase KT processes statements which require only prologue or epilogue code, such as PROCEDURE, ENTRY, BEGIN, RETURN, and STOP statements. The phase also processes the CHECK condition prefix, by indicating the library subroutine call that is required if the condition is raised.

PHASE INPUT

Input to the phase consists of the main text stream in Type 2 text format. For all compilations, statements linked in a chain headed by the XSYSCH field in XCOMM are examined and processed. If the CHECK prefix option has been used in the source program, the whole text stream is examined for situations where the condition is enabled.

The general and names dictionaries are accessed during processing.

PHASE OUTPUT

Output from the phase consists of the main text stream, in which text tables in the statements processed by the phase have been modified or replaced by new text tables inserted in the text stream.

No dictionary entries are created by this phase.

PHASE OPERATION

Sequence of Processing

The main scanning routine, KT1, uses the XTCH macro to scan the backwards-pointing XSYSCH chain. As each statement header is found, the statement-type byte is examined and control is passed to the appropriate routine for processing of the statement. When all statements in the chain have been processed, the XSCLNG1 field in XCOMM is examined. If the first bit in this field has been set by Phase EA, it indicates that the CHECK prefix option is used in the program. Control is passed to routine KT2, which scans the entire text stream and carries out the required processing. If the CHECK option is not indicated, control is passed to the next phase immediately after processing of the XSYSCH chain statements.

Processing of Statements Requiring System Interface Facilities

For all statements requiring the facilities of the system control program for their execution, the interface with the system program is

provided by library subroutines. When such a statement is found in the XSYSCH chain, control is passed to an appropriate routine for processing as described in the following paragraphs.

Processing DISPLAY Statements: Display statements are processed by the DISP0 routine. A statement can consist of the keyword-operator and an element-expression only, or this can be followed by the REPLY option and a character-variable, and the EVENT option followed by an event-variable can also be specified. Thus the body of the statement on input to Phase KT can consist of one, two, or three text tables. If both options are present, input to the phase can be represented as follows:

```
SN
SN          null      null      Display expression
REPLY       null      null      Reply expression
EVO         null      null      Event variable
```

Each operand is checked for validity, and the text tables are then replaced by an argument list and a call to the library subroutine required to execute the statement. The text output can be represented as follows:

```
SN
ALIST          3              null            Arg.list-temp no.
ARG       Display var.         1                 "       "    "
ARG       Reply var.           2                 "       "    "
ARG       Event var.           3                 "       "    "
CALL      Arg. list            Library-entry-          null
          temp. no.            point no.
```

The appropriate bit in the XLIBSTR field in XCOMM is set.

Processing DELAY and WAIT Statements: DELAY statements are processed by the WAIT1 routine. In each case, the delay expression or the event variable is checked for validity, and the text tables are replaced by ALIST, ARG, and CALL text tables indicating a call to the appropriate library subroutine. The relevant bit in the XLIBSTR field in XCOMM is set.

If a WAIT statement contains more than one event variable and an expression, the expression is evaluated before text is generated. The first operand of the first ARG table indicates that there is an expression and also indicates the number of event variables. This is followed by an ARG table containing a temporary operand indicating the evaluated result of the expression followed in turn by an ARG table for each event variable.

Processing SIGNAL Statements: SIGNAL statements are processed by the SIGNALA routine. If the specified condition is a program-checkout or computational condition, the prefix options are examined to check whether or not the condition has been disabled. If it has been disabled, the statement is deleted and a diagnostic message is generated. If the condition is valid, it is passed as an argument in a call to the appropriate library subroutine.

Processing PROCEDURE, BEGIN, and ON-BEGIN Statements

Phase KT processes those statements which indicate the block structure of the program, PROCEDURE, BEGIN, and ON-BEGIN statements. These statements require prologue code to provide save areas and to acquire automatic storage (referred to in the following descriptions as GET DSA code), and prologue code to initialize the storage. Phase KT determines the requirement for prologue code, and generates text to define some of that code. In doing so it generates labels which are required to enable branching to various sections of the code.

The processing performed by the phase can best be illustrated by reference to a simple example. The source statements:

```
A:  PROCEDURE(P1,P2,P3);
    .
    .
    .

B:  ENTRY(P2);
    .
    .
    .
    END A;
```

result in the text input to Phase KT containing the following text tables:

```
SL
PROC          dict.ref.       -          -
PEND 02       CL.03           null       null
PEND 00       CL.03           null       null
  .
  .
  .
SN
GSL           null            null       CL.08
  .
  .
  .
SN
END           -               -          -
```

Note: The compiler-generated label identifying numbers are shown for illustration purposes, and do not necessarily relate to numbers appearing in the text.

Information about the entry-point names, any applied options, and parameters is held in dictionary entries.

Phase KT modifies the text, to indicate to later phases that prologue code is required as shown in figure 2.18.

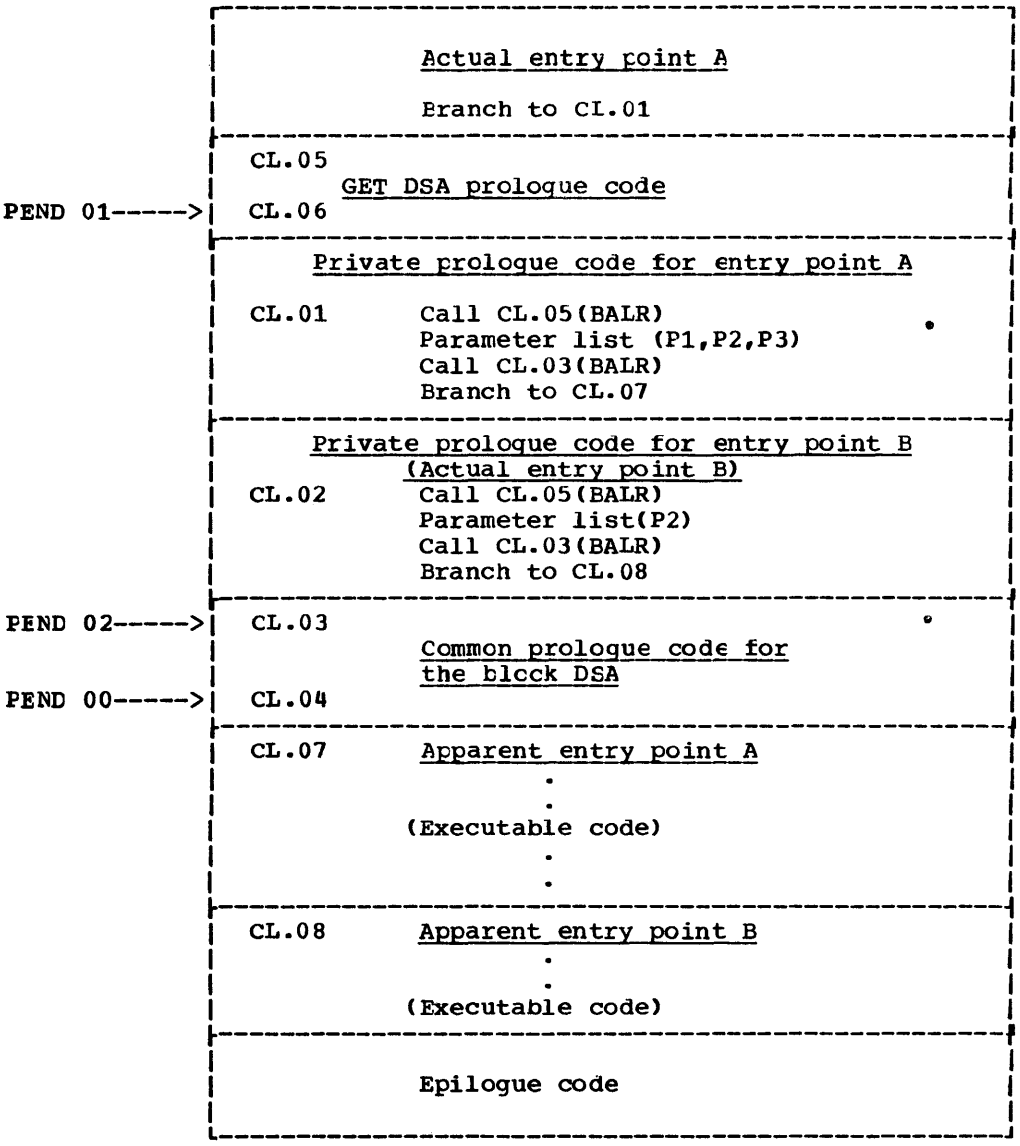


Figure 2.18. Simplified illustration of prologue code for a procedure block with a secondary entry point.

Because the XSYSCH chain is a backwards-pointing chain, the END statement for a block will be seen before other statements in the block. No significant processing of END statements is performed by this phase. ENTRY statement headers, which are followed by a GSL text table to indicate an apparent entry point to the block DSA, are not included in the XSYSCH chain, and therefore processing in connection with the prologue code for entry statements is performed when the block header statement is found.

When a PROCEDURE, BEGIN, or ON-BEGIN statement header is found, the dictionary entry for the entry point is accessed. This entry is linked to the entries for all other entry points in the block, and entry-point entries point to their related parameter-descriptor entries.

Each PROCEDURE, BEGIN, or ON-BEGIN statement header in the input text stream is followed respectively by a PROC, BGIN, or ONB text table. If such an entry point is declared with the COBOL, FORTRAN, or RPG option, the IOP2 byte of the PROC, BGIN, or ONB text table is set to X'02', X'03', or X'04'respectively. For each secondary entry point in the

block, an NTRY text table is generated to indicate the real entry point in the prologue code, and indications of COBOL or FORTRAN options are similarly set in this text table. The IOP2 byte of a PROC text table may also be set to X'01' during processing of RETURN statements, described later.

A PEND 01 text table is inserted after each PROC, BGIN, or ONB text table, containing the number of a compiler-generated label to be used to indicate the end of the "GET DSA section" of the prologue code. The PEND02 and PEND00 text tables following PROC, BGIN, or ONB text tables, or inserted after NTRY text tables, are set to indicate the start and end respectively of the common prologue code for the block. CALL text tables are generated to indicate the code that is required for branching between various sections of the prologue code and the executable code.

If the parameter lists for all entry points on a block are identical, a single parameter list can be moved into the common prologue code, and a MOVE text table to indicate this is generated. If there is more than one entry point, and the parameter lists are not identical, then MOVE text tables are generated for each entry point, to move the individual parameter lists into the appropriate sections of private prologue code.

Processing RETURN Statements

If a RETURN statement is used without a parameter list, no processing is performed by this phase. If a RETURN statement with a parameter list is found, its text reference is saved and the scan is continued until the related PROCEDURE statement is found. All entry points on the block are then processed as previously described, with the addition of processing of the RETURNS option. The dictionary entry for each entry point indicates the explicitly or implicitly declared attributes for the entry point. If the attributes for all entry points in the block are the same, no special action is taken. If entry points in the block have different attributes, then a MOVE text table is generated to insert return switches in the prologue code for each entry point. Text is then inserted in the RETURN statement to indicate the code required to test the return switches and determine the required entry point. CONV text tables are generated to indicate data type conversion of the RETURN parameter to satisfy the requirements of the selected entry point.

An operand in the RTRN text table is set to indicate whether the return is made to an entry point in the same block or in another block. The IOP2 field of the PROC text table is set to X'01' to indicate that the PROCEDURE statement has been processed out of the normal chain-scanning sequence.

Processing STOP and EXIT Statements

If a STOP or EXIT statement is found, control is passed to the STOPA routine. This routine generates a call to the appropriate library subroutine, and sets the appropriate bit in the XLIBSTR field in XCOMM.

Processing the CHECK Option

When the scan of statements in the XSYSCH chain is complete, the XSCLNG1 field in XCOMM is examined. If the first bit in this field is set, it indicates that the CHECK prefix option is used in the program. If the bit is not set, control is passed to the next phase. If the bit is set, control is passed to the routine KT2 which checks for situations where

the CHECK condition is raised. For each occurrence of the condition, a call is generated to a library subroutine, passing the identifiers for which the condition is raised as arguments.

Testing for the condition is performed during a sequential scan of the whole text stream. If the CHECK or NOCHECK prefix option is used on a statement in the source program, it will have been converted into a separate CHECK or NOCHECK statement by Phase EA, and translated accordingly by Phase II. Thus, the source statement:

```
(CHECK(A,B,)):P:PROC;
```

results in the text input to Phase KT containing the following text tables:

```
SN (CHECK)
CHCK          -    -    A
CHCK          -    -    B
SL(PROC)
PROC    .  .  .  .  .  .
```

This phase modifies the text by overwriting the CHECK or NOCHECK statement header with the true statement header (SL(PROC) in the example). It then examines the CHCK or NCHK text tables in the statement, and creates an entry in a stack for each variable in the value list. Each entry contains the 6-byte reference of the variable, followed by a code byte (X'A0' for CHECK, X'B0' for NOCHECK) and the block level of the statement. If the option is used without a value list, a CHCK or NCHK text table with null operands appears in the input, and an appropriate entry is made in the stack.

When the stack has been built during examination of the block-header statement (PROCEDURE or BEGIN), statements within the block are scanned. If the CHECK option applies, then each text table in the statement is examined. If a variable or Q-temp. appears in the third operand, it indicates that its value is being set. The appropriate stack entry is accessed to see if the CHECK condition is enabled for the variable (an intermediate stack is used for Q-temps.). If so, a call is generated to the library subroutine that executes the CHECK condition, passing the variable as an argument.

When an END statement is found, the block level of the next statement is examined to determine whether the next block inherits the prefix options. If not, the stack is cleared and new entries created as required for the new block.

The first time the CHECK option is applied to a variable in a block, the IOP2 field of the relevant CHCK text table is set to X'01' to indicate that storage must be allocated for an ON control block (ONCB) for the variable, unless an ONCB has been inherited from a preceding block. Where the CHECK option is applied other than on a PROCEDURE or BEGIN statement, it may also be necessary to create a NOCHECK ONCB. Where CHECK is specified without a checklist, one ONCB is created for the block.

Identification of Returned VARYING CHAR Strings

To ensure that the library interface is aware when a returned string is VARYING, the compiler generates code in the prologue of any entry point returning VARYING. This code sets the bit in the returns string descriptor (passed by the calling program to indicate VARYING. The library SORT interface routine may then interpret this bit.

This sequence of processing is initiated by Phase KT. When the MOVE text table which copies the return address in the DSA is generated, IST2

bit 6 in the table is set on if the returned string type is VARYING CHAR. During later processing, IST2 bit 6 is examined by Phases QA and SQ, and if the bit is set on, the identification sequence is completed.

PROCESSING FILE DECLARATIONS (PHASE KL)

This phase checks the validity of each file declaration and, for each file, builds a dictionary entry containing a File Control Block (FCB), an Environment Block (ENVB), and Define-the-File Block (DTF). (Note that for a file declared ENV(VSAM) a DTF is not built.)

To conserve space, the following CSECTs are overlaid: XINIT, CHKENV0, DUN01I, DUN01O, DUN02I, DUN02O, DUN13I, DUN13O, DUN26I, DUN26O. All the CSECTs except XINIT and CHKENV0 contain models DTFs for the 3540 diskette unit.

PHASE INPUT

Entries for file constants and for ENVIRONMENT options in the general dictionary are scanned and accessed as required. Entries for file names in the names dictionary are also accessed.

PHASE OUTPUT

The phase builds FCB, ENVB, and DTF entries in the general dictionary. LIOCS names are generated and entries made for them in the names dictionary.

PHASE OPERATION

Checking of File Declarations

The routine DCLSET scans the chain of file constants entries in the general dictionary and checks whether any attributes have been declared. If attributes are declared, the CHKATS routine is called to check that the attributes do not conflict. As the check is carried out, the routine generates two words, ATTWRD and ILGWRD, which are stored in the phase for checking purposes. ATTWRD has a bit set for each valid declared attribute, and is used when selecting default attributes to be applied to the declaration. ILGWRD sets a mask to show which attributes conflict with the declared attributes, and is used in checking the attributes on OPEN statements.

The declaration is checked for the presence of the MEDIUM option and any other ENVIRONMENT options. The MEDIUM option is mandatory (except for ENV(VSAM) where it is not specified); if it is incorrectly specified or is missing, default values (SYS001,2311) are assumed, a diagnostic message is generated, and an error flag is set which will cause the UNDEFINEDFILE condition to be raised when the file is opened. If any other ENVIRONMENT options are declared, the CHKENV routine is called. This routine checks for any conflict in the ENVIRONMENT options and builds a 3-word field, later used when the DTF is built, in which bits are set to indicate each valid ENVIRONMENT option. This is stored in the phase together with a temporary environment block (ENVB) which is not output to the dictionary until the DTF is built. The temporary ENVB consists of twelve 4-byte entries; the first is set to zero and each of the others contains a code byte and either a 3-byte constant or a variable dictionary reference specified in the options. The format of the second part is:

NBLK	Blocksize
NREC	Record length
NRKP	KEYLOC value
NKYL	Keylength
NNDX	Index area size
NADD	Additional-buffer size
NOFF	Overflow tracks
NPASS	Password
NBND	BUFND
NBNI	BUFNI
NBSP	BUFSP

The main declaration-checking routine, CHKDCL, then examines the declared and default attributes, and the options on the ENVIRONMENT attribute, and determines the type of DTF required, and the length of the DTF. The length of the DTF is saved. The attributes and options are used to set a code byte, BRCHWD, which is used to index a branch table to select the appropriate subroutine for the building of the DTF. For VSAM files, a DTF is not required and so this part of the phase is bypassed.

The phase contains a number of skeleton DTFs of different types. According to the type of DTF required, a subroutine is selected to modify the skeleton DTF in accordance with information declared (and in accordance with the detailed specifications in the IBM System/360 DOS LIOCS Program Logic Manual, Order No. GY24-5020). The selection of the DTF type and the appropriate subroutine is shown in the flowchart for this phase in section 3.

During the building of the DTF, the LIOCS module name is generated and an entry made for it in the names dictionary. All LIOCS module name entries are chained together.

When building of the DTF is completed, a DTF header table is constructed to assist in the allocation of static storage for each individual field of the DTF. This table is placed in a general dictionary entry, which contains the header table followed by the DTF. The format of the dictionary entry is shown in figure 5.19.

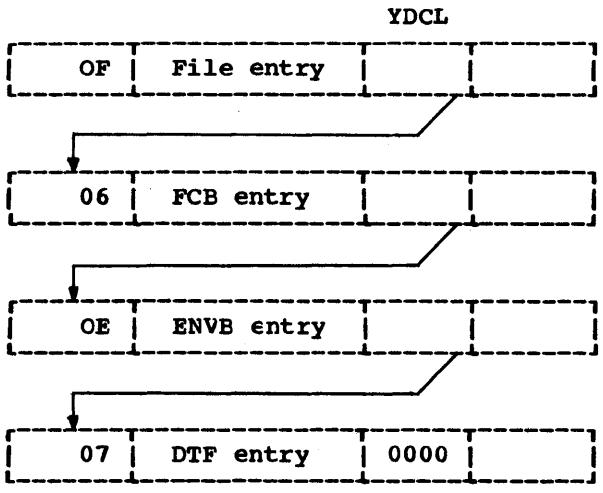
When the DTF dictionary entry has been completed, control is passed to the ENDDTF routine. This routine creates an ENVB dictionary entry (if any environment options other than the MEDIUM option have been specified), using the temporary environment block created earlier (see figure 5.18).

A file control block dictionary entry is then built. This entry contains the file control block (FCB), completed as far as possible at this stage (see figure 5.15), followed by a 28-byte block if the file is declared STREAM, or a 48-byte block if the file is declared RECORD,. No entries are made in a stream I/O block during compilation but, for a record I/O block, the last three fields are completed as follows:

Field name	Length (bytes)	Field content
FHSV	2	0 for scalarvarying. 2 otherwise
FECL	1	2 for backwards. 12 otherwise
FEMT	1	6th character or error module name

Both stream I/O and record I/O blocks in the FCB dictionary entry are followed by a 2-byte field indicating the length of the file name, which is followed by the file name.

At the end of processing by this routine, the dictionary entries for each file declaration are chained as shown below:


YDCL
OF
File entry
06
FCB entry
OE
ENVB entry
07
DTF entry
0000

PROCESSING OPEN, CLOSE, AND RECORD I/O STATEMENTS (PHASE KM)

The main function of Phase KM is processing of record-oriented input/output statements. It replaces some of the text tables that represent these statements with text tables which indicate either the code to be generated for inline execution of the statement or indicate the code to be generated to call a library subroutine for execution of the statements. If a call to a library subroutine is required, this phase determines the information required to be passed in the argument list, creates request control block, key descriptor, and record descriptor entries in the general dictionary, and builds the argument list.

It also checks that the attributes specified for each OPEN and CLOSE statement do not conflict with the attributes declared for the file. For each file referred to in an OPEN statement, an OPEN Control Block (OCB) is constructed. Text tables are created to generate calls to the appropriate library routines, together with the required arguments list.

Another function of the phase, which is performed before the processing of record I/O statements, is optimization of OPEN and CLOSE statements. This optimization consists of the insertion of information in the FCB and DTF of the associated file, thus reducing the number of calls to library subroutines that would otherwise insert such information at execution time.

BIF text tables which refer to the STORAGE and CURRENTSTORAGE built-in functions are replaced by text to evaluate the function.

PHASE INPUT

Initial input to the phase consists of the file-declaration entries in the general dictionary, which are chained together and linked from the XFILCH field of XCOMM. Chaining from each of these entries enables the associated FCB, ENVB, and DTF entries created by Phase KL to be accessed as required.

OPEN, CLOSE, and record I/O statements in the main text stream are accessed by scanning the chain of statements linked from the XRIOCH field of XCOMM. Variables-dictionary entries for the record and key variables are accessed to determine the lengths of these variables (if known). Aggregate-table entries in the general dictionary may also be accessed. FCB dictionary entries (created by Phase KL and possibly modified during earlier processing by this phase) are accessed if it is necessary to build an argument list for a library-subroutine call. If STORAGE or CURRENTSTORAGE are present, then all the text is scanned.

PHASE OUTPUT

If optimization of OPEN and CLOSE statements is found to be feasible, this phase modifies the appropriate FCB and DTF entries in the general dictionary by completing them as far as is possible at compile time. This includes the insertion of record size and block size in the FCB, and insertion of buffer sizes, buffer-address information, and CCW information in the DTF. Relocation information is also inserted in the DTF and in an overflow entry chained from the YCDED field of the FCB entry.

The phase builds OCB and ENVB entries in the general dictionary as required for OPEN and CLOSE statements. ALIST, ARG, and CALL text tables are generated for calls to library OPEN and CLOSE routines.

Output resulting from the processing of record I/O statements consists of the main text stream, in which some of the text tables contained in record I/O statements have been replaced by text tables specifically indicating object code to be generated. Depending on the features of each statement processed, these text tables either indicate code to be generated for inline execution, or indicate a call to a subroutine in library module IBMDRIO. If a library call is generated, entries are made in the general dictionary to hold arguments such as a request control block, a record descriptor, and a key descriptor.

PHASE OPERATION

Sequence of Processing

Optimization of OPEN and CLOSE statements is the first main function performed by the phase; it is performed by routines and subroutines starting at FILSC. All file-declaration entries in the general dictionary are accessed in turn by scanning the chain of entries linked from the XFILCH field of XCOMM. Each file entry contains a pointer which enables access to associated FCB, ENVB, and DTF entries. These entries are examined to determine whether optimization is possible; if so, the FCB and DTF entries are modified by completing them as far as is possible at compile time.

When all entries in the XFILCH chain have been examined and processed as necessary, control passes to the STMSCN routine. This routine uses the XTCH macro routine to scan the chain of statements in the main text stream that are linked from the XRIOCH field of XCOMM, and control is passed to the routines OPENPR, CLOSER, and RECIO, which process OPEN, CLOSE, and record I/O statements respectively.

Processing of any STORAGE and CURRENTSTORAGE built-in function references takes place after all record I/O statement processing.

Optimization of File Opening and Closing

The function of routines starting at FILSC is to check compatibility of associated files, and to determine, and insert into the relevant FCB and DTF, information which would otherwise be determined and inserted by library subroutines at the time of file opening or closing during execution.

The chain of file-declaration entries in the general dictionary is scanned. Each entry contains a pointer (in the YDCL field) which enables the associated FCB, ENVB, and DTF entries created by Phase KL to be accessed. The file, FCB, and ENVB entries are examined to ascertain whether or not optimization can be attempted. The criteria that must be satisfied at this time are:

- The file must have the EXTERNAL attribute.
- There must be no variables in the options list of the ENVIRONMENT attribute.
- The file declaration must not have been flagged by Phase KL as being erroneous (YCFER field of FCB set to X'47' indicates error).

If these conditions are satisfied, the DTF entry and part of the FCB entry are copied into phase working storage (XSTG). Subroutines starting at PA 032, which are similar to subroutines contained in the library modules IBMDOPA and IBMDOPB, are called to ascertain or

calculate values to be inserted into various fields of the FCB and DTF. These values include record size, block size, key length, key location, number and sizes of buffers, index-area sizes, and CCWs. When these values have been inserted into the copies of the FCB and DTF, the subroutines RFCBRT and RDTFRT are called to calculate the offsets of the various fields from the origins of the FCB and the DTF, and the relocation factors to be applied to enable items in the DTF to be addressed relative to the origin of the FCB. The dictionary entries created by Phase KL contain a number of fields which are initially set to zero, and which are overwritten by these offset and relocation values. An overflow dictionary entry is built to hold additional entries to the FCB, and a pointer to this overflow entry is inserted in the YCDED field of the existing FCB entry.

When all the required values have been determined and inserted in the working copies of the dictionary entries, the size of the file control section as modified by the allocation of buffers, index areas, etc., is calculated. If the size is greater than or equal to 32K bytes, optimization performed by this phase cannot be applied, existing entries in the dictionary are left unmodified, and all the calculations performed by this phase in the attempt at optimization are left for recalculation by library subroutines at execution time. If the file control section is less than 32K bytes long, the dictionary entries for the file are modified by use of the copies in the phase working storage. The scan of the file chain is then stepped to the next entry.

Processing OPEN and CLOSE Statements

All OPEN and CLOSE statements are included in the XRIOCH chain. Phase KM scans this chain, using the routine STMSCN.

When an OPEN text table is found, the OPENPR routine is invoked. This routine examines the attribute declared in the OPEN statement (only INPUT or OUTPUT are allowed) and checks that it does not conflict with the file declaration. An open control block (OCB) (see figure 5.23) is then constructed for use as an argument in the library call. The OCB is a single word which indicates which attribute was specified. ALIST, ARG, and CALL text tables are then created to generate a call to the library OPEN routine.

When a CLOSE statement is found, the CLOSER routine is invoked. This routine examines the ENVIRONMENT option on the statement; only the LEAVE option is permitted. It then creates an ENVB entry in the general dictionary, (see figure 5.18) and creates ALIST, ARG, and CALL text tables to generate a call to the library CLOSE routine.

Processing Record I/O Statements

The main scanning routine, STMSCN, scans the statements chained from the XRIOCH field of XCOMM. When the statement header of a record-oriented input/output statement is found, the routine RECIO is called to process the statement.

Within each statement, the main features are contained within two text tables. Whilst scanning for these text tables, certain items of information that may be contained in other text tables are copied into a save area for use in later processing. These items include:

- References to Q-temps., which are saved so that variables being qualified can be identified.

- Information about keys, contained in some CONV and CONCAT text tables, which may be used for optimization of key processing for some REGIONAL(3) files.

The contents of the operator and operand fields of the two text tables that indicate the main features of the statement are as follows:

1.

Operator	IOP1	Statement type (e.g., READ, WRITE, LOCATE, etc..)
	IOP2	Statement index number (uniquely identifies the statement type and the options used)
Operand 1		File reference
Operand 2		KEY/KEYTO/KEYFROM option
Operand 3		INTO/FROM/SET/IGNORE variable reference or LOCATE variable if it is a LOCATE statement.

2.

Operator	IOP1	EVO
	IOP2	Zero
Operand 1		Length of COBOL variable (if a COBOL file) or Abnormal-Locate-Return label (if a LOCATE statement).
Operand 2		Null
Operand 3		EVENT variable or SET pointer if LOCATE or READ-SET statement.

When these text tables are found, their operands are copied into two save areas, OPNDSV and OPNDSV2 within the phase.

The statement index number is used to control searches of two tables in the phase, RCBTAB and TMTAB. RCBTAB contains a request-control-block word (RCB word) for each basic type of record I/O statement. Each RCB word consists of a bit pattern indicating the statement type and options used. When an appropriate RCB word is found, it is copied into RCBSAV for later use. Each entry in TMTAB contains a test-under-mask instruction which is used, together with the RCB word, to form the request-control-block that is used as a library argument. The appropriate test-under-mask instruction is saved in TMINST.

If the file referred to in the statement is not a variable or a parameter, the validity of the statement is checked by comparison of the statement type with the declared file attributes as shown in the file-control-block in the FCB entry, built in the general dictionary by Phase KL. Unless the statement is found to be invalid, flag bits are set in the first six bytes of an 8-byte field named INLFLG. These flags indicate the declared file attributes and any environment options applicable to the statement. INLFLG is later completed and used by a subroutine, INMON, to test for the feasibility of generating inline code.

Appropriate routines are then called to process any of the INTO, FROM, SET, or IGNORE options that are specified on the statement. Similarly, other routines are called to process any KEY, KEYTO, or KEYFROM options, and finally the EVENT option is processed. During processing of some of these options, the INMON subroutine is called and, if the generation of

inline code is feasible, appropriate routines are called to generate the required text tables. If INMON finds that inline code is not feasible, control is returned to the option processing routine, and arguments are constructed for use in a library call.

GENERATION OF INLINE CODE: Before it can be determined whether inline code can be generated instead of a library call, information about various features of the statement must be collected in a usable form. This is done by setting flag bits in an 8-byte field, INLFLG. The main processing routine sets flags in the first four bytes of INLFLG to indicate the declared file attributes (as shown in the FCB dictionary entry) and in the fifth and sixth bytes to indicate any options of the environment attribute that are used in the statement. These flags are set when information required for building the request control block has been collected, and the statement has been checked for compatibility with the declared file attributes.

During processing of the INTO, FROM, or SET options, (which is described in later paragraphs,) the routine INFRRT builds a record descriptor for the record variable or the locate variable as applicable. The record descriptor is passed as an argument if a library call is generated.

Before the record descriptor is built, but when some of the information required for it has been collected, the subroutine INMON is called.

The function of the INMON subroutine is to monitor the feasibility of generating inline code. If it is found to be feasible, control is passed to routines which generate the required text tables. Otherwise, INMON returns control to the options processing routine from which it was called.

One of the first actions of INMON is to complete the collection of information about the statement in INLFLG. Flag bits are set in the seventh byte (the eighth byte is not used,) to indicate:

- Whether the record variable is a varying-length string.
- Whether the SCALARVARYING option of the ENVIRONMENT attribute is specified.
- Whether the record variable is an area.
- Whether the record (or block) size is known.
- Whether the record variable length is known.
- Whether the statement is an input statement.
- Whether the IGNORE option is specified.

When these flags are set, the bit pattern in INLFLG is compared with preset bit patterns in a table named INLRTS. This table contains a number of entries, each consisting of a pair of 8-byte fields. The preset bit patterns in these fields indicate a number of conditions which must be satisifed before inline code can be generated. INLFLG is compared with the first field of each entry in turn to determine whether certain conditions are satisfied. When a satisfactory comparison is found, INLFLG is then compared with the second field of that particular entry, to ensure that certain features do not apply to the statement. These two comparisons are required because certain features are not always critical, and therefore an exact match of bit patterns would not always be a valid test.

If a satisfactory comparison is not found in INLRTS, it indicates that inline code cannot be generated, and control is returned to the

option-processing routine so that arguments required in a library call can be constructed. If a satisfactory comparison is found, the sequential number of the matching entry in INLRTS is used to index a directory, RTCADS. This directory contains the address of an entry in a table named INCODS. The entry referred to is applicable to the type of statement being processed.

Entries in INCODS are of varying length, and each byte of each entry indicates a 2-digit hexadecimal number. Each hexadecimal number indicates a particular routine that is called to generate text tables indicating a particular code sequence. The selected entry is scanned, one byte at a time, and the text-table-generating routines are called in the sequence indicated.

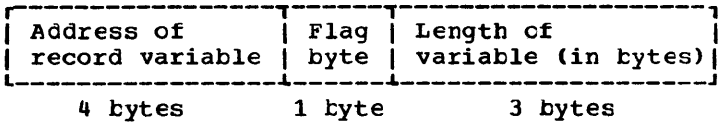
BUILDING THE LIBRARY-CALL ARGUMENT LIST: If a record I/O statement is to be executed by means of a library call, an AIIST text table, a series of ARG text tables, and a CALL text table are generated. The ARG text tables indicate the arguments to be passed to the library subroutine, which are:

1. File control block (FCB)
2. Request control block (RCB)
3. Record descriptor or ignore factor
4. Key descriptor or zero
5. Event variable or zero
6. Abnormal locate return or zero

The dictionary entry for the file control block is built by Phase KL, and is accessed during processing by this phase. Information required for building the request control block is collected in early processing by this phase, but the associated dictionary entry is not created until after it has been determined that inline code cannot be generated. The tests for the feasibility of inline code are made during the processing of some of the options in the statement, when much of the information required for building the record descriptor has been determined. If inline code is not to be generated, processing of the options is completed and the arguments are constructed. When all the arguments have been constructed, the required ALIST, ARG, and CALL text tables are generated.

Building the Record Descriptor: If the statement contains the INTO or FROM option, the routine INFRRT is called to examine the record variable and create a record descriptor (RD). The routine is also used to create a record descriptor if the statement is a LOCATE statement.

A record descriptor is a 2-word control block, with the following format:



Bits are set in the flag byte if the variable is a varying-length character-string, a varying-length bit string, or an area.

When examination of a record variable indicates certain conditions, a skeleton RD is created. The conditions are:

1. The address of the record variable will be known during execution of the prologue code.

2. The length of the record variable can be found during compilation

   or

   The length of the record variable cannot be greater than 32K bytes, and prologue code to insert the length in the RD can be generated.

If the above conditions can be satisfied, a skeleton RD is built and placed in the general dictionary (see figure 5.21), where it can be accessed by a phase in the storage allocation stage for allocation of the requisite static storage. Flags are set in the dictionary entry to indicate whether prologue code will be required for the purpose of inserting the variable length in the RD, and whether the RD needs to be moved from static storage into automatic storage during execution. Where possible, RD dictionary entries are commoned, to save static storage space and to avoid duplication of prologue code.

If the conditions for creation of a skeleton RD are not satisfied, text tables are generated to enable the RD to be built inline, in temporary storage, during execution. The information required in the text tables is found by use of various subroutines, and a number of temporary storage fields and flags are used by the phase to pass information between these subroutines.

The type of the statement (input or output), and the data type of the record variable are examined. Subroutine LTHFND is invoked, and, if the length of the record variable is known it stores this value in the SIZVAL field and sets a flag in LTHFLG to indicate that the length is known. If the record variable length is not known, LTHFND sets other flags in LTHFLG to indicate certain other characteristics of the variable. If the variable is an unaligned bit string, this subroutine also attempts to find the hang (the number of unused bits in the first and last bytes). If the hang is found, another flag in LTHFLG is set. If LTHFLG indicates that both length and hang are known, the value in SIZVAL is reset to allow for the hang.

If the record variable is a structure or array, the subroutine LTHFND examines the aggregate table entries in the general dictionary. The information used varies according to the type of the array or structure. If the variable is a structure with adjustable extents, the data type of the last base element has a considerable effect on the calculation of the variable length. For example, if the last base element is an AREA variable, the length required for an output statement is the current extent of the area plus the length of the other elements of the structure. If the last base element is an interleaved array of unaligned bit strings, a library call must be made to find the length of the variable. On return of control from LTHFND, INFRRT examines the record variable and sets various flags, including flags in DSCFLG to indicate any flag settings required in the RD.

The subroutine INMON is then called to test whether inline code can be generated. If so, no further processing is performed by INFRRT, and control is passed to the text-table-generating routines. If inline code cannot be generated, control returns to INFRRT.

The subroutine RLTHGN is then invoked. If the length of the record variable is known, RLTHGN generates an ASSN text table to assign the length value in SIZVAL to a temporary operand. If the length is not known, text tables are generated to enable the length to be calculated during execution. The processing required varies according to whether the record variable is a base element, a structure, an array, or a cross-section of an array. In each case, the generated text tables include an assignment of the length to a temporary operand.

On return from the subroutine RLTHGN, another subroutine, TDSCIN is

invoked TDSCIN creates an 8-byte temporary RD, and places a 6-byte reference to it into the buffer TMPDSC. Subroutine ASLDSC is then used to assign the temporary operand that holds the record variable length to the second word of TMPDSC, and MVFLAG is called to assign the flag byte to TMPDSC. Finally the MVADDR subroutine is called to generate a LADDR text table, to move the address of the record variable (or the SET pointer, if the statement is a LOCATE statement) into the first word of TMPDSC.

Text tables are then generated, in which the contents of TMPDSC are assigned to a temporary operand, and which indicate code to be generated to enable the record descriptor to be created at execution time.

The sequence of operations performed by the INFRRT routine is shown in figure 2.19.

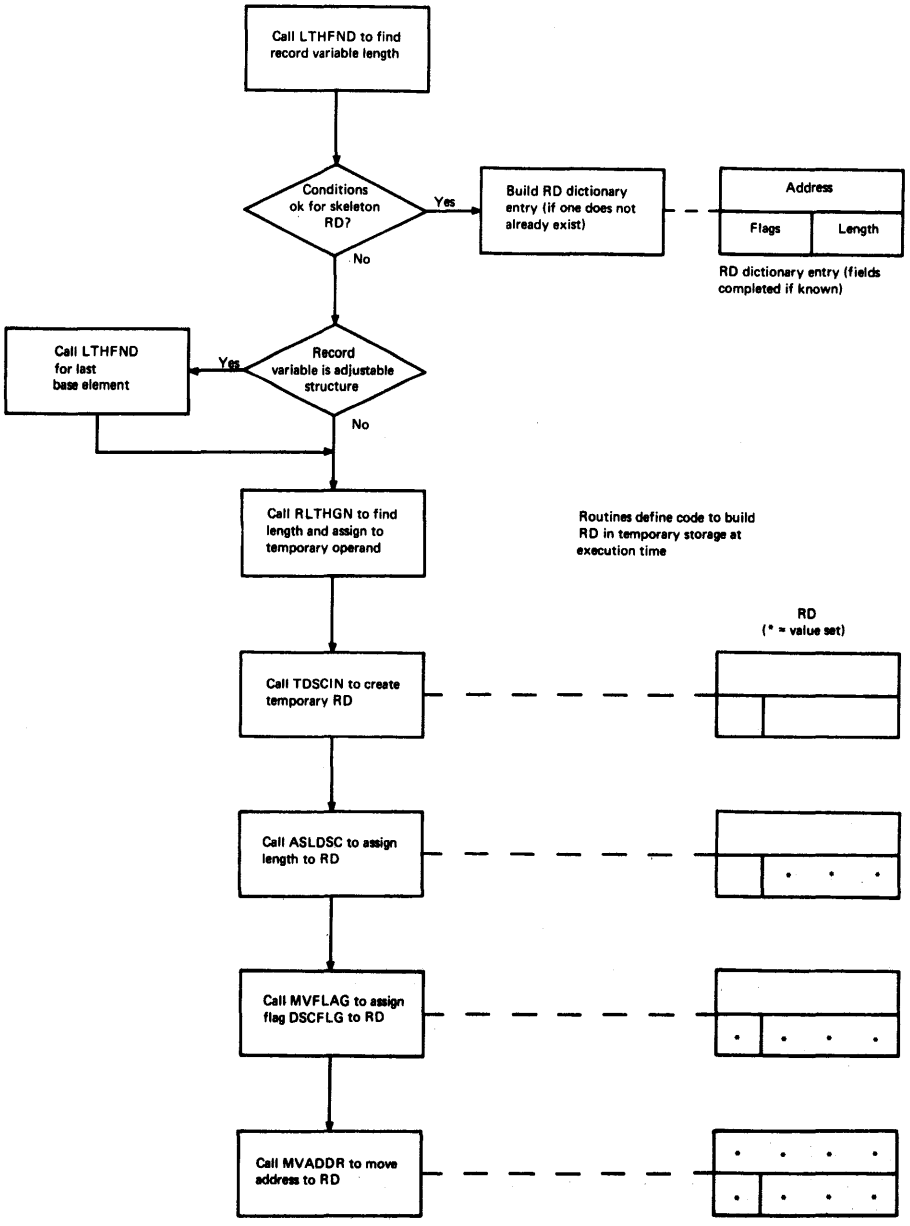


Figure 2.19. Creation of record-descriptors by Phase KM

Processing the IGNORE Option: The routine IGNRRT processes the IGNORE option. It examines the IGNORE factor from the input text, which has been saved in OPNDSV, and if necessary has text generated to convert the factor to fullword binary integer format. The factor is then saved for used in building the argument list.

Processing the SET Option: The SET option is processed by the SETRT routine. If the statement is of the form READ---SET---, the routine simply saves a reference to the SET pointer, for use in building the argument list. If the statement is a LOCATE statement, a reference to the abnormal-locate return address is also saved. For a LOCATE statement, the pointer reference is moved into PTRREF. If the statement does not contain an explicit pointer, the implicit pointer is found from the PTSAT text table containing the record variable. The INFRRT routine is then entered at IF0100 to create an RD for the LOCATE variable.

Building the Key Descriptor (KD): If the KEY, KEYTO, or KEYFROM option is used in the statement, the KEYRT routine is used to build a key descriptor (KD). The KD is a 2-word control block with the format:

The flag byte is used to indicate whether the key is a variable length string in a KEYTO option, and whether a binary region number is supplied. If the region number is supplied a word containing the number is added to the end of the KD. This enables some optimization when processing REGIONAL(1) and REGIONAL(3) files.

The KD is built in a similar manner to the RD. If the key is a character-string variable for which the address will be known during execution of the prologue code, a skeleton KD is created and placed in the general dictionary, for use by a storage allocation phase.

If a skeleton KD is not used, a temporary KD must be constructed inline. The subroutines LTHFND, RLTHGN, and ASLDSC are used to construct the code in similar way to the code for the RD.

Processing the EVENT Option: If the EVENT option is used in the statement, the routine EVNTRT is invoked. This routine saves the event variable for use in building the argument list.

Processing STORAGE and CURRENTSTORAGE Built-in Function References: Since a reference to either of these functions can appear in most statement types, including record I/O and stream I/O, Phase KM scans the whole of the text when XSCLNG2 Bit 5 indicates that such a function is referenced by the program. Processing involves the use of routine RECIO, with calls being made (as appropriate) to LTHFND and then RLTHGN.

Initially, a QT stack is not maintained for STORAGE/CURRENTSTORAGE processing. However, if operand 1 of the BIF text table is a QT, then the statement is re-processed with a QT stack being maintained.

PROCESSING STREAM I/O STATEMENTS (PHASE KQ)

Phase KQ processes all stream-oriented input/output statements, and replaces the text tables that represent these statements with text tables that indicate the code required for execution of the statements.

Execution of stream-oriented input/output statements is always performed in some part by one or more library subroutines. Phase KQ determines which library subroutine is required, and generates text representing the argument list to be passed in the call to the appropriate library subroutine. The phase also determines what data-type conversions (i.e., to and from character-type data) are required for transmission to and from I/O devices, and generates calls to library routines that perform these conversions during execution. If global optimization has been specified, the phase will define the code to be generated to perform some of these conversions in line.

Specification of a global optimization option affects the processing performed on edit-directed I/O statements. If optimization is specified, the phase performs processing to match and merge items in the data list and format list of a statement, so that repeated branching during execution is minimized.

A feature of edit-directed stream I/O statements is that certain sequences of generated code are repeated a number of times. In order to reduce object-module size whilst avoiding the degradation of execution-time performance resulting from repeated calls to a library subroutine, common sections of code are collected in subroutines. These subroutines are generated by the phase, and can be called from any point in the compiled object code during execution.

PHASE INPUT

Input to the phase consist of the main text stream in Type 2 text format. Only those statements linked by Phase KA into the chain headed by the XSIOCH field in XCOMM are examined and processed. These statements contain text tables peculiar to stream oriented input/output statements, as shown in figure 2.20, and these are the text tables that are replaced by this phase. Each DATAE text table identifies an element of a data list, which may be an element variable, an array, or a structure. Each FORME text table identifies an element of a format list, the type of the element being identified in the IOP2 field of the operator. A FIT text table indicates the start of a format-list iteration, with the iteration factor being contained in the first operand field. The end of a format list iteration is indicated by a FITE text table. When Phase II translates the text into Type-2 text format, it generates five NULL text tables after each GET, PUT, and DATAE text table to allow for expansion of text during processing by this phase.

The variables and general dictionaries are accessed for information about files and pictured format items appearing in the processed statements.

PHASE OUTPUT

Output from the phase consists of the main text stream in Type 2 text format, in which text tables peculiar to stream-oriented I/O statements have been replaced by general-type text tabels indicating object code to be generated. The text that is generated is described in the paragraphs describing phase operation.

Some entries may be made in the general dictionary for format-element descriptors (FEDs).

PHASE OPERATION

Sequence of Processing

The routine KQSS00 uses the XTCH macro to scan statement headers in the XSIOCH chain. As each statement header is found it is examined, and certain items of information are saved for use in later processing. Control is then passed to a routine appropriate to the statement type. If the statement is a GET or PUT statement, control is passed to the routine KQGP00. This routine scans the text tables of the phase until the GET or PUT text table is found. Using information in this table and in relevant dictionary entries, it builds an argument list and generates a call to the appropriate library subroutine.

When this preliminary processing, which is performed for all GET or PUT statements, is completed, control is passed to appropriate routines for processing other text tables in the statement. Routine KLSC00 processes LIST directed statements, routine KDSC00 processes DATA-directed statements, and routine KESC00 processes EDIT directed statements. In general, these routines process data and format lists by replacing DATAE text tables in DATA and LIST directed statements, and replacing DATAE, FIT, FITE, and FORME text tables in EDIT-directed statements. On completion of this expansion, any compiler-generated subroutines required for EDIT-directed I/O statements are output, and calls to library subroutines required to perform data-type conversions are also generated. The scan is then stepped to the next statement in the XSIOCH chain. If it is a FORMAT statement, control is passed to the KQFM00 routine, which performs processing similar to that performed on the format list of GET EDIT and PUT EDIT statements, and uses the same subroutines to perform this processing.

A feature of phase operation is that before each input text table is processed, it is copied into a field named QVBFTX in the phase working storage. Fields within QVBFTX are identified by symbolic names similar to those used to identify fields in text tables, e.g., BOPND2 holds the operand equivalent to the IOPND2 field of a text table.

OPERATOR	OPERAND 1	OPERAND 2	OPERAND 3
GET {LIST / EDIT / DATA} [PAGE]	FILE(filename) or STRING(stringname)	[SKIP(exprn)]	[COPY(filename)]
PUT {LIST / EDIT / DATA} [PAGE]	FILE(filename) or STRING(stringname)	[SKIP(exprn)]	[LINE(exprn)]
DATAE	data-list element	null	null
FORME PAGE	null	null	null
FORME LINE	w	null	null
FORME COLUMN	w	null	null
FORME SKIP	w	null	null
FORME REMOTE	label const./vble	null	null
FORME X	w	null	null
FORME F	w	[d]	[p]
FORME E	w	d	[s]
FORME S1	picture spec.	null	null
FORME A	w	null	null
FORME B	w	null	null
FIT	n	null	null
FITE	null	null	null

Figure 2.20. Text tables used in stream I/O statements prior to Phase KQ

Processing GET and PUT Text Tables

The routine KQGP00 scans for the GET or PUT text table following a GET or PUT statement header. When the text table is found, the second operator code byte (IOP2) is examined to determine whether the statement relates to LIST, DATA, or EDIT directed input/output, and whether the PAGE or COPY options apply to the statement. If the statement is EDIT directed, a flag is set in the ISF field of the statement header to indicate this fact to phases in the global optimization stage; the complicated branching involved in EDIT-directed input/output makes such statements unsuitable for optimization. The routine KQSF00 is used to check the validity of SKIP, LINE, and PAGE control format items on GET FILE and PUT FILE statements. This routine accesses the FCB entry, created in the general dictionary by Phase KL, to obtain information about the relevant file attributes, which is then used for validity checking.

The text tables that are generated to replace a GET or PUT text table include an ALIST text table, indicating the number of arguments in a following argument list, a number of ARG text tables, each indicating an argument, and a CALL text table, indicating the library subroutine that is to be called at execution time. These text tables, which are created by various subroutines, indicate the initial code required for execution of the statement. One of the items in the argument list is the stream input/output control block (SIOCB) in which a number of fields must be initialized at compile time. A temporary operand is generated to indicate that temporary storage must be allocated for the SIOCB, and OFFS and MOVE text tables are generated to initialize various fields. The format of the text that is generated is illustrated in the following example:

The source statement:

```
GET DATA SKIP(5);
```

results in text input to Phase KQ containing a text table which can be represented as follows:

```
GET(DATA)       SYSIN      5      null
                DTF
```

After processing by Phase KQ, the sequence of text tables that are generated to replace the input text table can be represented as follows:

ALIST	3 (no.of args.)	null	argument- list no.	
ARG	SYSIPT DTF	1	"	
ARG	SIOCB temp.	2	"	
OFFS	offset of field	SIOCB temp.	Q-temp.1	SIOCB initialization
MOVE/05	null	code for 'GET DATA'	Q-temp.1	
OFFS	offset of field	SIOCB temp.	Q-temp.2	
MOVE/05	null	DSA LEVEL NO.	Q-temp.2	
OFFS	offset of FIELD	SIOCB temp.	Q-temp.3	
LA/02	EOS GSL	NULL	Q-temp.3	
ARG	5	3	argument list no.	SKIP option
CALL/01	argument- list no.	library entry point no.	null	

The library subroutine that is called depends upon whether any PAGE, SKIP, LINE, or COPY optional control format items are specified. The subroutine SLMEP0 sets the appropriate bit in the XLIBSTR field of XCOMM, to indicate to Phase SI the library subroutine that is required.

When this initial text for the statement has been generated, control is passed to the KLSC00 routine if the statement refers to list-directed I/O the KDSC00 routine if the statement refers to data-directed I/O, or KESC00 if the statement refers to edit-directed I/O.

Processing DATAE Text Tables in List-directed I/O Statements

Routine KLSC00 scans the input text tables of a GET LIST or PUT LIST statement, searching for DATAE text tables. These indicate items in the data list of such a statement. As each DATAE text table is found, its operand is checked for validity within a data list. If the operand is valid, control is then passed to either KLAE00, if the operand is an array, or to KLSE00 if the operand is an element. Different library subroutines are used to control the transmission of these two types of data, and each of the two routines mentioned generates an argument list appropriate to the relevant library subroutine.

If the operand is an element, the argument list consists only of the SIOCB. Routine KLSE00 generates text to point R1 at the SIOCB, and then to store the address of the data item and its DED in SIOCB. A CALL text table is then generated to indicate the required library subroutine, and the apppropriate bit in the XLIBSTR field of XCOMM is set.

If the operand is an array, the required argument list consists of the SIOCB, the array locator, the DED of the array, and the number of dimensions. Routine KLAE00 generates an ALIST text table, four ARG text tables containing the arguments, and a CALL text table to indicate a call to the appropriate library subroutine. It also ensures that the appropriate bit in XLIBSTR is set.

On completion of processing of a DATAE text table, control is returned to KLSC00 which then scans for the next DATAE text table.

Processing DATAE Text Tables in Data-directed I/O Statements

The routine KDSC00 searches the text tables of a GET DATA or PUT DATA statement, searching for DATAE text tables. If all the operand fields of the first DATAE text table found are null, it indicates that the statement does not have a data list. The text expansion for this type of statement is performed by routine KDNL00. If any DATAE text tables contain a valid operand, either routine KDIL00 or KDOL00 is branched-to, to control the processing of the data list of a GET DATA or PUT DATA statement.

For a statement without a data list, routine KDNL00 generates text tables to create an argument list with two arguments, and to indicate a call to the appropriate library subroutine. The arguments are the SIOCB, and the first element in a chain linking symbol tables for variables active at the time the statement is executed. Because this chain cannot be constructed at this stage of compilation, a marker is set in the ARG text table to indicate that Phase PC is required to create the chain of symbol tables and to complete the ARG text table.

For a GET DATA statement with an argument list, one call is made to a library subroutine to control the transmission of the entire data list. When the first DATAE text table is foumd, routine KDIL00 generates an ALIST text table, and an ARG text table to indicate the SIOCB. It then generates an ARG text table containing the first item in the data list, and this text table is flagged to indicate to Phase PC that a symbol table is required to replace this argument. Control is then returned to routine KDSC00 which finds the next text table. KDIL00 generates one ARG text table, similarly marked for action by Phase PC, and the sequence is repeated until the last DATAE text table in the statement is replaced. A CALL text table, indicating a call to the required library subroutine is generated, and the appropriate bit in XLIBSTR is set.

For a PUT DATA statement with an argument list, it may be necessary to generate calls to more than one library subroutine to control the transmission of the data list. Library subroutine IBMBSDOA deals with elements and complete arrays; library subroutine IBMBSDOB deals with array elements. The argument list passed with each invocation of one of these library subroutines must contain only the appropriate data type. Therefore the processing of a data list may contain a number of calls to these library subroutines, each preceded by an argument list.

When the first DATAE text table is foumd, an ALIST text table is generated, followed by the appropriate ARG text table. The next DATAE text table is examined and, if it contains an operand in the same category as the preceding one, an ARG text table is generated accordingly. When a DATAE text table is found to contain an operand in a different data category from those preceding it, it is not processed until a library call for the preceding category has been generated, followed by an ALIST text table to indicate a new argument list. The current DATAE text table is then replaced, and the process is repeated until all DATAE text tables have been replaced. Before the last call to a library subroutine is generated, an OFFS text table and an OR text table are generated to indicate code required to set a flag in the SIOCB. This flag indicates that the library subroutine is required to generated a terminating semicolon in the output stream.

Processing Edit-directed I/O Statements

After replacing the GET or PUT text table of an edit-directed I/O statement, routine KQGP00 passes control to routine KESC00 for initial processing of the datalist and format list. This routine generates a LADDR text table to indicate code required to point R1 at the SIOCB, and an ASSN text table which indicates to Phase QA (register allocation

phase) that R1 should always point at the SIOCB if possible. A check is then made of the XNSYGBT field in XCOMM to determine whether OPT=TIME has been specified. If global optimization has been specified, control is passed to routine KEMT00; if not, control is passed to routine KEDT00.

Routine KEMT00 examines the data lists and format lists to see if the items they contain can be organized into matching pairs of items. Because a statement can have more than one pair of data and format lists, KEMT00 may be called more than once during the processing of a statement, and text is generated after each pair of lists has been examined. If this routine finds that the items contained in a pair of lists can be individually matched, control is passed to KEPT00 for generation of the optimized text. If a pair of lists cannot be matched, control passes to routine KEDT00 for generation of non-optimized text. Although the sequence of text tables generated by routine KEPT00 and KEDT00 differs, much of the text is common, and both call subroutines SELDD0, SCDLE0, SECSR0, and SEFMT0 to create text tables.

The text generated for list-directed and data-directed data transmission consists, to a great extent, of calls to library subroutines, preceded by appropriate argument lists. In the case of edit-directed input/output, most of the data transmission is controlled by compiler-generated subroutines, which are generated at the end of processing by this phase. In processing the statement, calls to these subroutines are generated. The arguments to these subroutines are passed in the SIOCB for the statement, at which R1 is set to point. The text generated for an item in a data list indicates the code required to insert the address of the source data item and the address of its DED in appropriate fields of the SIOCB. Similarly, for a format item, the text indicates the code required to insert the address of the target field and the address of its format element descriptor (FED) in other fields of the SIOCB. Text indicating a call to the appropriate compiler-generated subroutine is then generated.

If data-type conversion is required for edit-directed I/O, it is usually controlled by a library format-director subroutine, and text is generated to indicate the required calls. In some cases, e.g., conversion of fixed decimal or fixed binary data with F-type format to character form for output, text is generated to indicate code required to perform the conversion inline.

When the processing of a data list/format list is complete, routines KEPT00 and KEDT00 return control to routine KESC00 for processing of the next pair of lists or, if all lists in the statement have been processed, pass control to routine KQSE00 for end-of-statement processing.

End-of-statement Processing

When the data list or the data and format lists of a GET or PUT statement have been processed, control is passed to routine KQSE00. This routine generates the text required at the end of such a statement. The text consists of a GSL text table, containing a compiler-generated label allocated during initial processing of the statement, to mark the logical end of the statement so that global optimization phases can branch around the statement. This is followed by a KONST text table, which indicates to the register and storage allocation phases that the SIOCB for the statement is no longer required, and that R1 can be reallocated.

Processing FORMAT Statements

When the main scanning routine finds a FORMAT statement, control is passed to routine KQFM00 for processing of the statement. Because edit-directed I/O statements are not suitable for processing by the global optimization phases, a flag is set in the ISF field of the statement header, and text is set up to enable branching around the statement. This is done by generating a copy of the statement header table, and replacing the original statement header with a GOTO text table. This indicates a compiler-generated label that is generated in a GSL text table at the end of the statement.

The text tables of the statement are then scanned, and the text references of the first FORME or FIT text tables and the last FORME or FITE text tables are saved. These text references are saved to enable all the text tables between the two references to be duplicated. One copy of the text tables can then be replaced with text indicating the code for an output statement, and the second copy replaced with text indicating code for an input statement.

The scan is then reset to the start of the statement and text is generated to:

1. Save the return address of the format list of the calling GET or PUT statement.

2. Test whether the calling statement is a GET or PUT statement, and to indicate a branch to the second part of the FORMAT statement if it is a GET statement.

The text tables of the statement are then rescanned, and subroutines SEFMT0, SEFIT0, and SEFIE0 are called as required to replace FORME, FIT, and FITE text tables with the required output text. During this processing, and when the output-statement format list has been replaced, text is generated to load the calling statement return address into R9 and to return control to the calling statement. The text tables of the input statement format list are then replaced, and the end-of-statement text is generated as for an edit-directed GET or PUT statement.

Generation of Data-transmission-control Subroutines

If edit-directed input/output statements have been processed by the phase, control is passed to routine KQSR00 when all statements have been processed. This phase generates subroutines that are called from various places in edit-directed input/output statements.

The phase coding includes a table of text-table skeletons. This table is divided into five main sections, each containing text to be generated to define a subroutine. Of the five subroutines that can be generated by the compiler to control edit-directed transmission, two are used for data input, two are used for data output, and one is used for output of X-format items. When the requirement for one of these subroutines is recognized during statement processing, an appropriate bit is set in the XCOMSTR field in XCOMM. This bit indicates processing required by the final assembly phase (Phase SI) and also indicates to routine KQSR00 which section of the text-table-skeleton table is to be copied into the text stream.

The text for each of these subroutines consists of a CGSR text table, a number of GEN and LLAD text tables, and a CGSR01 text table. Each GEN text table contains a sequence of object code that is to be included in the final assembly.

Identification of Library Subroutines Required for Conversions

Stream input/output data transmission involves the conversion of data to and from character-type data. The conversions are performed by library subroutines, and the need for these subroutines is indicated when the conversion requirement is known, even though the subroutines are not called from compiled code. The requirement for the library subroutines is indicated to the final assembly phase by the setting of appropriate bits in the XLIBSTR field in XCOMM.

Each time a data item in a data-directed or list-directed input/output statement is checked for validity, the subroutine SILCRO is called. This subroutine examines the data type, and checks whether the item is in a GET or PUT statement, to determine which library subroutine is required to perform data conversion. It then sets the appropriate bit in XLIBSTR.

In edit-directed I/O statements, the selection of conversion subroutines depends on the format item. A subroutine that processes format lists, SFED00, contains a list of the conversion subroutines required for each type of format item for either input or output transmission. This table is accessed each time a format item is seen, and the appropriate bit is set in XLIBSTR.

SPECIAL-CASE PROCESSING (PHASE KV)

This phase scans the main text stream, searching for situations where the text can be modified in such a way that more efficient object code will be generated by the compiler. This form of optimization is performed for every compilation, regardless of whether or not the compiler options specify that phases in the global optimization stage shall be executed, (although one of the functions performed by the phase is mainly preparation for global optimization stage processing).

The functions of the phase are:

1. To mark the beginning of flow units, for use by phases in the optimization stage.
2. To optimize the use of compiler-generated labels and branching instructions.
3. To de-nest argument lists for programmer-defined functions and procedure calls.
4. To optimize operations involving constant value exponents and multipliers, by replacing them by repeated multiplication operations or addition operations respectively.
5. To optimize comparison operations by replacing the bit-string result of an operation by the variable to which the bit-string is assigned after conversion.
6. To optimize arithmetic operations (ADD, SUBTRACT, and MULTIPLY) involving fixed-decimal operands.
7. To process references to the SUBSTR, UNSPEC, CHAR, and BIT built-in functions, the SUBSTR pseudovariable, and concatenation operations involving bit strings.

PHASE INPUT

Input to this phase consists of the main text stream in Type-2 text format, in which the logical sequence of text tables is indicated by chain fields. The dictionary sections are not accessed by this phase.

PHASE OUTPUT

Output from the phase consists of the main text stream, modified by the insertion or deletion from use of various text tables. The modifications are mentioned in the description of phase operation.

PHASE OPERATION

The text stream is scanned sequentially by use of the XNEXT macro. Where situations requiring processing are encountered, text tables are added to the text in logical sequence by use of the XNSRT macro, or are deleted from use by employing the XLINK macro to modify the forward chain fields.

Marking of Flow Units

A flow unit is a sequence of text that can only be logically entered at the beginning and left at the end. Division of the text stream into flow units is required by phases in the global optimization stage of the compiler. This phase marks the beginning of each flow unit by setting the ISF field of the appropriate statement-header text table to a value of X'80'. Flow units are detected by scanning for labels, or for CALL, FNCT, or BC text tables. The first text table (other than a NULL text table) following these indicaticns is the table in which the flag is set. In the case of labels, the flag is set in the table containing the label.

Optimization of Compiler-Generated Branching Instructions and Labels

The text is examined for situations where compiler-generated branching instructions cause sequences of code to be unreachable, or where compiler-generated GOTO instructions are redundant. In such cases, redundant text is deleted.

The text is also examined for situations where compiler-generated labels appear in juxtaposition, allowing excess labels to be removed. To detect such situations, a text page is acquired for the building of a label directory. An entry is made in the label directory for each compiler-generated label detected, either in a GSL text table or as an operand in a BC, LA, or GOTO text table.

When a GSL text table is found, a scan ahead is made to determine whether a group of GSL or SL text tables appear together. If they do, the situation is examined to determine whether one label can be selected to serve the function of the group of labels. Whenever possible, a source-statement label (SL) is selected in preference to a compiler-generated label (GSL). The label directory is used to direct a scan of all references to the label group; the references are changed to references to the selected label.

During the search through juxtaposed labels, the previously described optimization of GOTO statements is carried out.

De-nesting of Arguments to Programmer-defined Functions and Procedures

Argument lists passed in calls to procedures and references to programmer-defined functions are de-nested as required by this phase. Although procedure calls cannot be nested, their argument list may be nested if any function references are nested within the call. For example, de-nesting of the argument lists required by the source statement:

```
CALL P(F1(F2(F3(X))));
```

(where P is a procedure and F1, F2, and F3 are references to programmer-defined functions)

will effectively result in text to indicate the following sequence:

```
CALL F3(X),F2(T1),F1(T2),P(T3)
```

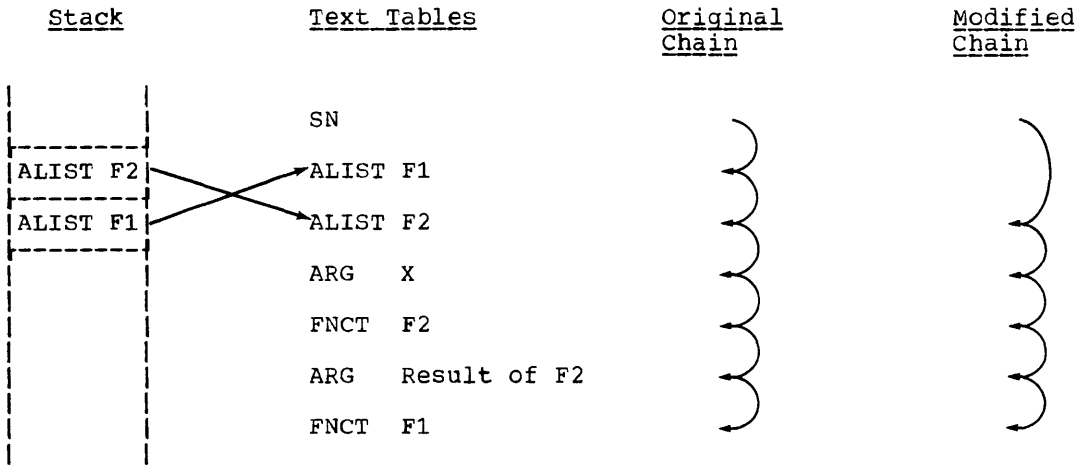
where T1, T2, and T3 are temporary operands representing values returned by functions and passed as arguments to another function or procedure. To assist in de-nesting argument lists, a stack is maintained, with an entry for each level of nesting. Entries are made when a CALL, FNCT, or

ALIST text table is seen. Each entry contains a table-type marker and the text reference of the table.

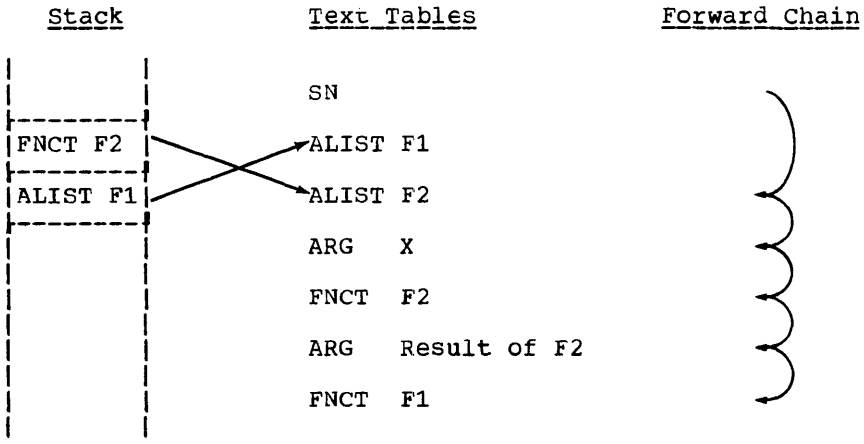
When an ALIST text table is seen, the entry in the stack is examined. If that entry is an ALIST entry, a new entry is made for the new ALIST text table. If the top entry is a CALL or FNCT entry, it is overwritten with the ALIST entry, thus maintaining one entry for each level of nesting. When a second ALIST entry is made in the stack, the forward chains in the text tables are modified so that the text table preceding the one for the first ALIST entry in the stack points at the ALIST text table related to the second ALIST entry in the stack. Thus, the processing for the source statement:

```
A=F1(F2(X));
```

is illustrated below. (Note: The text tables shown are symbolic and do not indicate the operand fields).

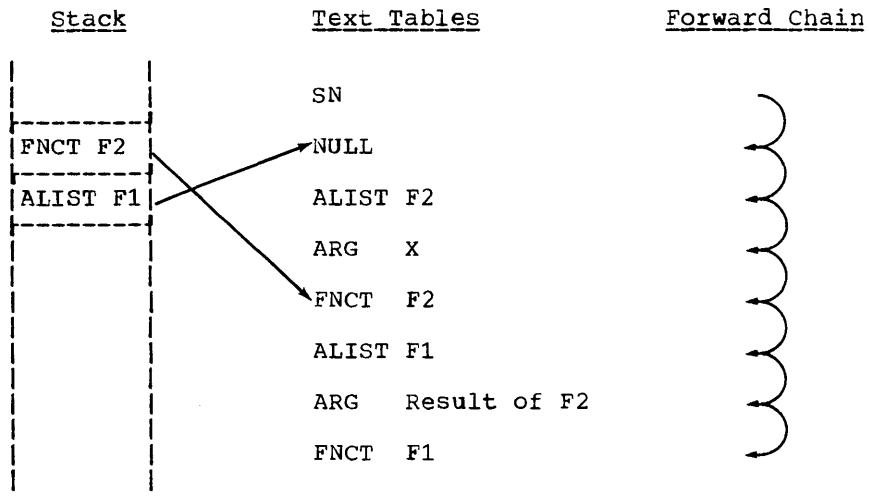


When the text table forward chain has been modified, the top ALIST stack entry is overwritten by an entry for the relevant function reference, so that the situation appears as illustrated below:



The text table scan is resumed. If a FNCT text table is found when the top entry in the stack is a FNCT entry, this entry is effectively deleted and the penultimate entry in the stack is accessed. This entry will be the ALIST entry corresponding to the FNCT table found in the text. This entry is used as a pointer for modifying the text-table sequence. Starting at the ALIST text table pointed at by this stack entry, each text table is repositioned in the text, after the FNCT text table pointed to by the top entry in the stack (the insertions are made by use of overflow pages and chains). As each table is copied, the

original text table is deleted by changing its code byte to NULL. When a nested ALIST table in the original text stream is encountered during this copying process, the copying is suspended until the corresponding FNCT text table is encountered. Copying is resumed after this text table, and continues until the FNCT text table pointed at from the top stack entry is encountered. The situation at this stage of processing is illustrated below:



If the currently-accessed entry in the stack is the only entry, the de-nesting is complete. If there are other entries remaining in the stack, the preceding (ALIST) entry is overwritten with the FNCT entry, and the processing is resumed.

Optimization of Exponentiation and Multiplication Operations (Strength Reduction)

When a MULT (multiplication) or POWER (exponentiation) text table is encountered, the second operand field is examined to determine whether the exponent or multiplier is a constant with a value in the range 0-18, or of 20, 24, or 32. If this is found to be so, optimization is possible. MULT text tables are converted to a series of PLUS text tables, and POWER text tables are converted to a series of MULT text tables.

This optimization is not performed for MULT text tables with fixed decimal operands, as the execution of addition instructions for packed decimal data is not significantly faster than execution of multiplication instructions.

Optimization of Comparison Operations

Text tables indicating comparison operations (GT, EQ, LT, GE, LE, and NE text tables) are examined to see if the first and second operands are of the same data type, and if the third operand is a bit string. When these text tables are generated (by Phase II), a bit string temporary operand of length 1 is generated in the third operand field if the result of the comparison is not used in a bit-string operation.

When Phase KV detects one of these text tables with a bit-string temporary third operand, it carries out a look-ahead search for a reference to the temporary operand in another text table. If the temporary operand is used as the source operand in a CONV (convert) text

table, the target (third operand) of the conversion is used to replace the third operand in the comparison text table, and the CONV text table is changed to NULL.

The modified comparison text table is later processed by Phase OC and replaced by the following sequence of text tables:

```
ASSN        1          -        Result
BC     operand1    operand 2    CL.1
ASSN        0          -        Result
GSL        CL.1
```

Optimization of Decimal Arithmetic Operations

The code generated for evaluation of some expressions containing fixed decimal operands can be made more efficient than the minimum required for implementation of the language. The DECOPT routine examines the text for expressions in which addition, subtraction, and multiplication operations involve fixed decimal operands, and modifies the text generated by Phase II to enable this optimization.

The optimization is performed during compilation because of a difference between the machine implementation of fixed decimal operations, and of float or binary operations. This difference is best shown by reference to the following example. The source statement:

```
A = B + C * D;
```

causes Phase II to generate text containing the following text tables:

```
MULT        C        D   temp.1
PLUS        temp.1   B     A
```

For operations involving fixed binary or float operands, the operations are executed using registers which hold the intermediate temporary results. Registers are not used in the same way in fixed decimal operations, so that if A, B, C, and D in the example are fixed decimal operands, the code generated to represent the expression would contain the following sequence of instructions:

```
ZAP         temp.1, C
 MP         temp.1, D
MVC         A, temp.1
 AP         A, B
```

The DECOPT routine examines all expressions involving fixed decimal operands and, where possible, modifies the text so that it indicates instructions in which the operations are carried out into the target, thus eliminating intermediate temporary operands. Thus, for the example given, the code generated would contain the following optimized sequence:

```
ZAP         A,C
 MP         A,D
 AP         A,B
```

This form of optimization is not possible in the following circumstances:

- If any intermediate temporary operand has a precision which exceeds that of the final result. Substitution of the final result might cause an overflow interrupt.
- If the operand used to hold the final result is also an active operand in the expression, e.g., A=B+C*D+A;

In such cases, a temporary operand with the maximum precision required is used to accumulate the intermediate result until the final operation, when the real result is used as the target of the operation.

Optimization of On-units

For each ON statement that is not associated with SYSTEM or a null on-unit, Phase EA creates an ON-BEGIN block, even though the on-unit may not be a BEGIN block. Text for this block is generated and processed by various phases of compiler. If the on-unit consists of an unconditional branch to a label variable or a label constant, Phase KA sets a flag bit in the block-header entry in the general dictionary. The ONTAB routine in Phase KV checks this flag. If it is set, text is generated (before the ONS text table that represents the ON statement) to load the address of the branched-to label into the ONCB. If the label is a label constant, it is first assigned to a label-variable temporary operand. The text representing the ON-BEGIN block is then deleted.

BUILT-IN FUNCTION AND PSEUDOVARIABLE PROCESSING (PHASE KK)

Phase KK examines all built-in function (BIF), pseudovariable (PSV), and exponentiation (POWER) text tables. It replaces them with sequences of text tables which indicate, to later phases of the compiler, the code to be generated in order to obtain one of the following results at execution time:

1. Inline evaluation of the built-in function or pseudovariable.
2. A call to a library routine for evaluation of the built-in function or pseudovariable.

This phase also reorganizes the text stream so that the logical sequence of text tables coincides with their physical sequence, and no text tables output from this phase appear on overflow text pages. At the same time it allocates overflow text pages at the rate of one to each four existing pages, in a similar manner to the allocation made by Phase KA.

PHASE INPUT

This phase scans the whole of the main text stream, and accesses the general and variables dictionaries as required. On input to the phase, the logical sequence of the text tables on the original text pages and on overflow pages is maintained by use of a chain via the IFCHN fields (see figures 2.16 and 2.24). Because the text tables that are processed by this phase are not linked in any statement-type chain, the whole of the main text stream is scanned by use of the XNEXT macro so that all BIF, PSV, and POWER text tables can be detected. All of these text tables, except those relating to string-handling operations, are processed by this phase. Before the text is scanned by Phase KK, some items have been processed so that they do not require processing by this phase. These items include:

REAL and IMAG pseudovariables, in which the data-type code bytes of the arguments have been changed by Phase IA.

The format of a typical BIF text table on input to Phase KK is as follows:

Operator	(IOP1	Code byte indicating BIF text table.
	(IOP2	Code byte indicating type of built-in function.
Operand 1		First argument, or null if BIF has no arguments.
Operand 2		Second argument, or null if BIF has less than 2 arguments.
Operand 3		The function result or target.

The arguments will have been converted to the required data type by Phase II, but may have varying precisions and modes. Phase II will also insert a temporary function result in the Operand 3 field if the target is not of the correct data type, or if the target is fixed binary data with a scale factor not equal to that of the function result. The BIF table will then be followed by an ASSN or CONV table which assigns the function result to the target.

Typical input to the phase is shown in the following examples.

1. The source statement:

```
X = COMPLEX (A,B);
```

will result in the following text table appearing in the input to Phase KK:

```
BIF(COMPLEX)    A    B    X
```

2. The source statements:

```
DCL X COMPLEX FLOAT, (I BINARY, J DECIMAL) FIXED;

X = COMPLEX(I,J);
```

will result in the following text tables appearing in the input to Phase KK:

CONV	J		t1
BIF(COMPLEX)	I	t1	t2
CONV	t2	-	X

where t1 is a fixed binary temporary operand
and t2 is a fixed binary complex temporary operand.

3. Where a built-in function has more than two arguments, additional BIF tables are used. The source statement:

```
X = MAX (A,B,C,D);
```

will result in the following text tables appearing in the input to Phase KK:

BIF(MAX)	A	B	-
BIF(MAX)	C	D	X

PHASE OUTPUT

The main text stream output from the phase is organized so that the logical sequence of text tables coincides with their physical sequence, and all IFCHN fields contain a value of X'0000'. For each four text pages, an overflow page is allocated to allow for text expansion by later phases. If global optimization has been performed, all hash-chain tables except those immediately following flow-unit-header tables are deleted. BIF, PSV, and POWER text tables that appeared in the input text are replaced by general-type text tables, examples of which are shown in the descriptions of phase operation.

Where constant operands are generated during phase operation, general dictionary entries are made as required.

PHASE OPERATION

The main text stream is scanned in logical sequence by use of the XNEXT macro. As each text table is seen it is copied into the output text stream so that the text tables are physically reorganized in logical sequence. If the text has been processed by phases in the global optimization stage, flow-unit-header tables are copied, but only those hash-chain tables immediately following the flow-unit headers are copied. The input text stream is discarded.

As each text table is copied into the output stream, its IOP1 field is examined to see if it is a BIF, PSV, or POWER text table. If it is, control is passed to an appropriate processing routine which generates

text tables to replace the original one in situ. For a BIF or PSV table, the processing routine is selected by using the IOP2 value to index a branch vector table. The routines that process particular types of text tables can be grouped into two general classes. Those routines that generate text for inline evaluation of the built-in function or pseudovariable at execution time, and those routines that generate text defining a call to a library subroutine. (A library subroutine is always called to evaluate exponentiation.) The general processing performed by these two classes of routines is described in following paragraphs.

Generation of Text Tables for Inline Evaluation

In general, the sequence of text tables to be generated for a particular function or pseudovariable is predetermined. Variations of this sequence are required according to the data type of the argument/s. The first function of the processing routine is to test the data type of the argument/s and select the appropriate text-table sequence.

The operands and flag bytes of the input text tables are copied into two tables within the routine, OPNDTB and FLGTBL respectively. The data type of the argument operands is tested and the required text-table sequence is selected. Temporary and constant operands required in the output text tables are then created, and stored in two holding tables, TEMPTB and CONSTB. Routine TMPSET creates a skeleton 6-byte reference to a temporary operand, and places it in TEMPTB, where the DED is completed as required. Routine CNSSET creates a constant operand and stores it in a 6-byte field in CONSTB. If necessary, a general dictionary entry is made for the constant and the dictionary reference is used in CONSTB.

When all required temporary and constant operators have been created and placed in their respective holding tables, the selected sequence of text tables is generated. For each text table required, the XBFSK macro is used to generate a 9-byte driver table, defining the contents required in the various fields of the output text table. Each operand is defined in the driver table according to its type, (i.e., an operand in the input text, a newly created temporary or constant operand, etc.,), and its location (i.e., in OPNDTB, TEMPTB, CONSTB, etc.,).

The output text generation routine, GENTXT, is then called to create the required sequence of output text tables. This routine either copies the contents of the driver table to build the operator and null operand fields, or copies the contents of OPNDTB, TEMPTB, CONSTB, and FLGTBL as directed by the operand type and locator bytes in the driver table. The operand type byte may also indicate the status of the operand (e.g., a temporary operand being used for the last time) and GENTXT will set the operand code byte to reflect this status.

If a required operand is indicated as a compiler generated label, GENTXT will generate a label with the appropriate identifying number, and with the branch condition indicated in the driver table.

Typical processing by the phase is illustrated in the following example:

The PL/I source statement:

```
I = SIGN(X);
```

would result in the following table being input to Phase KK:

```
BIF (SIGN code)    X    -    I
```

The operands and flag bytes would be copied into the tables OPNDTB and FLGTBL:

OPNDTB

1	X
2	NULL
3	I

FLGTBL

1	Flag(X)
2	X'00'
3	Flag(I)

Examination of the argument, X, would indicate that the following coding sequence must be produced by the compiler:

```
         LA    R,1       R is result register
         LE    S,X
         LTER  S,S       test argument for sign
         BP    CL.1
         BZ    CL.2
         BCTR  R,0
CL.2     BCTR  R,0
CL.1     STH   R,I
```

The text table sequence required to cause the above code to be generated would be selected. This would indicate that two temporary operands, representing register R and S, and one constant operand, with a value of 1, must be generated by the processing routine, and stored in the tables TEMPTB and CONSTB.

TEMPTB

1	Register-temporary R
2	Register-temporary S

CONSTB

1	Constant 1

The GENTXT subroutine would then be invoked; a pointer to the following driver tables would be passed to this subroutine as an argument:

```
XBFSK    LADDR,,(NULL),(CONST,1),(TEMP,1)
XBFSK    ASSN,,(OPND,1,LAST),(NULL),(TEMP,2)
XBFSK    GOTO,LTER,(TEMP,2,LAST),(NULL),(CL,1,BP)
XBFSK    GOTO,GOCND,(NULL),(NULL),(CL,2,BZ)
XBFSK    ASSN,BCTR0,(TEMP,1),(NULL),(TEMP,1,RESET)
XBFSK    GSL,,(CL,2),(NULL),(NULL)
XBFSK    ASSN,BCTR0,(TEMP,1),(NULL),(TEMP,1,RESET)
XBFSK    GSL,,(CL,1),(NULL),(NULL)
XBFSK    ASSN,,(TEMP,1,LAST),(NULL),(OPND,3,LAST)
XBFSK    END
```

Using these tables, the GENTXT subroutine would generate any necessary compiler labels (with the required branch condition codes) and generate the following text tables:

Operator IOP1	Operator IOP2	Operand 1	Operand 2	Operand 3	Flag 1	Flag 2	Flag 3
LADDR	00	-	1	R temp	-	-	-
ASSN	00	X	-	S temp	Flag(X)	-	-
GOTO	08	S temp(last)	-	CL1 (BP)	-	-	-
GOTO	03	-	-	CL2 (BZ)	-	-	-
ASSN	0C	R temp	-	R temp(reset)	-	-	-
GSL	00	CL2	-	-	-	-	-
ASSN	0C	R temp	-	R temp(reset)	-	-	-
GSL	00	CL1	-	-	-	-	-
ASSN	00	R temp(last)	-	I	-	-	Flag(1)

Generation of Text Tables for Library Calls

The appropriate processing routine constructs a 6-byte reference for each argument, and places each reference, in its appropriate argument list position, in a table, OPNDTB. Any flag bytes (in the input text table) which refer to these arguments, are copied into corresponding positions in another table, FLGTBL.

The processing routine then calls the PLISTG routine. This routine generates an ALIST text table, and an ARG text table for each argument in OPNDTB. The corresponding flags are copied from FLGTBL into the appropriate flag fields of the output text tables. Because most library routines require all floating-point arguments to be of the same mode and length, the routines FLCONV and FLCTRG are called to convert arguments and create temporary operands as required.

When PLISTG returns control to the processing routine, a CALL text table is generated. The second operand field of this text table is used to indicate the entry point of the library routine to be called. The selection of the library routine is based on the type of the built-in function or pseudovariable. The selection of a particular routine (or entry point within a routine) may depend on the precision and mode of the arguments. The entry points of all library routines used by the compiler are allocated a unique number, and the phase contains a list of these entry-point numbers. The routine SETEPT is used to convert the argument type into a number, which is then used to index the entry-point list. The entry-point number found is inserted in the CALL text table. It is also used to set the appropriate bit in the bit vector in the XLIBSTR field in XCOMM.

Text Deleted by Phase KK

If global optimization has been performed, the text will contain flow unit header tables. Routine RDNEXT examines bit 6 the IFOFI field of each flow unit header to (see figure 5.99) to see if the flow unit is to be deleted If so, all the text tables of the flow unit are deleted, and a diagnostic message indicating this fact is generated. If such a flow unit consists of an END statement, the text is not deleted, but bit 1 of its ISF field is set to indicate that no epilogue code is required.

STRING-HANDLING OPERATIONS - PART ONE (PHASE OC)

Phase OC and OX both process text tables that involve string-handling operations. The operations processed by Phase OC include:

- String assignments.
- The string-handling built-in functions BOOL and TRANSLATE.
- String operations involving the operators AND(&), OR(|), NOT(¬), GT(>), GE(>=), EQ(=), LE(<=), and LT(<).
- Concatenation of character strings, and of some types of bit strings.

PHASE INPUT

The phase scans the whole of the main text stream for text tables of the types processed by this phase. String assignments are indicated by ASSN or CONV text tables, built-in functions are indicated by BIF text tables; the second byte of the operator (IOP2) indicates whether text tables of these types are of interest to Phase OC. The logical and comparison operations are identified by text tables with the equivalent IOP1 codes. Concatenation operations are identified by CONCAT text tables that have not been processed by Phase KV.

The phase accesses entries in the general dictionary for character-string constants.

PHASE OUTPUT

Output from the phase consists of the main text stream, modified as described in the description of phase operation. Although this modified text stream may include calls to compiler-generated string-handling subroutines, the subroutines themselves are not generated by this phase, but are generated by Phase OX. OC sets appropriate bits in the XCOMSTR field of XCOMM to indicate this requirement.

Some entries for character constants may be created in the general dictionary if the TRANSLATE built-in function is present in the text. An entry may also be created in the general dictionary for masks used in bit-string assignment operations.

PHASE OPERATION

The main scanning routine scans the entire text stream, using the XNEXT macro. When a text table of possible interest to the phase is seen, control is passed to the appropriate processing routine as follows:

SCANASSN routine - examines ASSN and CONV text tables

SCANBIFS routine - examines BIF text tables

SCANLOG routine - examines AND, OR, and NOT text tables

SCANCAT routine - examines CONCAT text tables

SCANCOMM routine - examines BC text tables (comparison operators)

These routines examine the pertinent features of the text tables to determine whether they are to be processed by this phase. If so, control is passed to the appropriate processing routine.

In addition to the types of text tables previously mentioned, NDX, OFFS, and PTSAT text tables are also examined for te presence of Q-temps. representing string data items. A stack of active Q-temps. is built and maintained to enable the qualifications of a Q-temp. to be accessed whenever it is used as an operand in a text table processed by this phase.

Processing String Assignments

ASSN and CONV text tables involving assignments of character strings, or of bit strings not processed by Phase KV (i.e., variable-length bit strings and bit strings that are unaligned) are processed by a series of routines starting at SCANASS0. These routines are also used by other routines requiring movement of string data.

The source and target operands are examined to determine whether movement of a string can best be executed by inline code, by a compiler-generated subroutine, or by a library subroutine. A call to library subroutine is generated if an operation involves movement of an unaligned bit string (i.e., really unaligned as against declared UNALIGNED), movement of an aligned bit string to a target field that is really unaligned, or movement of a bit string to a target field that either has an adjustable extent or is more than 2048 bits long. The CALL text table generated in such cases is preceded by the appropriate argument list. A call to a compiler-generated subroutine, (the CGSMV subroutine generated by the Phase OX) is generated if an operation involves movement of a character string longer than 1536 bytes. However, before such a call is generated, the situation is examined to determine whether the code involved in calling the subroutines satisfies the conditions required if OPT(TIME) has been specified. If not, text is generated to define inline code in the form of MVC or MVI instructions which may be contained in a loop controlled by a BCT instruction. For all other operations involving movement of bit or character string, text defining inline code is generated.

Where inline code is required, the main features of the movement operation are defined in one or more MOVE text tables. According to the attributes of the source and target operands, the type of operation to be performed is identified by an appropriate code in the IOP2 field of the MOVE text table (see figure 5.96). Some types of MOVE text tables are always preceded by a KONST text table, containing operands which can effectively be considered as additional operands of the MOVE text table. Where the source and target fields overlap, a temporary operand is created, into which the source field is initially moved, and a second MOVE text table is generated to indicate movement of this temporary operand into the target field. If the temporary operand is adjustable, a VDA text table is generated to indicate that storage is required for the temporary operand; the length used in the VDA text table is derived from the length of the target operand. Where a string movement operation requires padding of the target field, MOVE text tables are generated to move the required padding characters. Where movement of a bit string results in the last bit of the source operand being positioned other than on a byte boundary, padding to the next byte boundary cannot be defined in a MOVE text table, particularly if the source operand is a varying string. An entry is made in the general dictionary, consisting of seven mask bytes defined in BITSMASK. The dictionary reference of the mask entry is inserted in a MOVE 0A text table, which indicates code to be generated to enable selection of the appropriate padding mask during execution.

When control is passed to the SCANASS0 routine for movement of strings in connection with other functions performed by the phase, a code is set in a field, MOVE1, to indicate the type of movement code to be defined by SCANASS0.

Processing Concatenation Operations

Bit-string concatenations not processed by Phase KV are performed by a library subroutine. Phase OC builds the necessary argument lists and generates calls to the requisite library subroutine.

CONCAT text tables with operands that are non-varying character strings with fixed extents are replaced by a series of MOVE and OFFS tables. This is illustrated in the following example:

The following source statements:

```
DCL TAR CHAR(5), C1 CHAR(3), C2 CHAR(2);

    TAR = C1||C2;
```

result in the text input to Phase OC containing the following text table:

```
CONCAT      C1      C2      TAR.
```

This text table is replaced by:

```
MOVE      C1       3       TAR
OFFS       3      TAR      Q.temp.1.
MOVE      C2       2       Q.temp.1.
```

If a concatenation operation involves varying or adjustable character strings, field lengths and offsets cannot be calculated at compilation time. Text tables indicating the code required to perform the calculations at execution time are generated, and the routines starting at SCANASS0 are used to generate inline code or a call to a compiler-generated subroutine to perform the string-movement operations.

If it is possible for an operand (other than the first operand) in a concatenation operation to overlap the target field, a temporary operand is generated to acquire an area of temporary storage in which the concatenated string can be built up. The string is then moved to the target field.

Processing BOOL and TRANSLATE Built-in Functions, and AND, OR, and NOT Operators

When a BIF text table is detected in the input text, the SCANBIF routine examines the IOP2 byte to determine whether it refers to either the BOOL or TRANSLATE built-in functions. If so, control is passed to either the BOOLF routine or the TRANSLF routine.

The routines starting at BOOLF examine the third argument to BOOL, which is in the second of two BIF text tables. If this argument does not represent an AND, OR, NOT, or EXCLUSIVE OR logical operation, a call to a library subroutine is generated. If the third argument does represent one of these logical operations, the arguments are further examined. If the target operand is adjustable, a call to a library subroutine is generated. If any of the operands are unaligned, a call to a different library subroutine is generated. If all operands are aligned, the target is not adjustable, and the third argument specifies one of the

logical operations, inline code is defined. If the operation involves a temporary result that is adjustable, a GETVDA text table is generated to acquire storage for the temporary operand that is generated. Control is then passed to the SCANASS0 routine to generate text defining the requisite move operations.

AND, OR, and NOT text tables are processed in a similar manner by routines starting at NDRNT. (Some of these routines are used in processing BOOL operations.) If any operand in one of these text tables is unaligned, a call is generated to the same library subroutines as is used in similar circumstances in BOOL. If the first or second operand is adjustable, or if the length of the result is greater than 256 bytes, a call to a library subroutine is generated. In other cases, inline code is defined. An appropriate code is set in the MOVE1 field to indicate to the SCANASS0 routine the type of movement code required.

The processing performed by the TRANSLF routine in connection with the TRANSLATE built-in function depends upon whether a position string is specified, and upon the attributes of the argument strings. In general, a call to a library subroutine is generated if the replacement string or the position string is adjustable or varying or longer than 256 bytes. Inline code is generated in other cases; the MVCL compiler-generated subroutine is not called. If the replacement string is non-varying and is less than 256 bytes long, it is assigned to a temporary operand which is 256 bytes long; a dictionary entry is created to hold the string constant used to pad the replacement temporary operand to 256 bytes. A dictionary entry may also be made to hold the translation table, which is moved by the SCANASS0 routine if OPT (TIME) is specified. If OPT (TIME) is not specified, the translation table is built by generating text defining movement operations within a loop controlled by a BCT instruction.

Processing Comparison Operations

Text tables indicating comparison operations (GT, GE, EQ, LE, and LT text tables) are processed by routines starting at SCANCOMP. Prior to input to this phase, the text tables are generated by Phase II with the two operands to be compared in the first two operand fields, and with a bit-string temporary operand in the third operand (result) field. Thus the following source statements:

```
DCL (A,B,C) FIXED BINARY;
 .
 .
 .
A = B > C;
```

cause the following text tables to appear in the output from Phase II:

```
GT        -         C       temp.
CONV      temp.     -        A
```

During processing by Phase KV, the bit-string temporary operand is changed to the data type to which the bit result is ultimately converted. Thus, on input to Phase OC, the text contains the following text table:

```
GT      B       C       A
```

The SCANCOMP routine determines the type of the result of the operation, and generates text to assign a value of one or zero in the appropriate representation to the result, with a conditional branch between the assignments. The text output from the processing can be represented as follows:

```
ASSN      1         -        A
BC        B         C      (BGT,CL.1)
ASSN      0         -        A
GSL       CL.1
```

If the data type of the result is a string, the SCANASS0 routine is used to define the move code required by the assignments. Otherwise, the assignments are left for processing by Phase KX.

STRING-HANDLING OPERATIONS - PART 2 AND COMPLEX-EXPRESSION EXPANSION (PHASE OX)

Phase OX processes text tables that involve string-handling operations, and also text tables involving complex expressions (but not the COMPLEX built-in function).

The string-handling items processed by the phase include the HIGH, INDEX, LENGTH, LOW, REPEAT, STRING, and VERIFY built-in functions, the STRING pseudovariable, and string comparisons defined in BC and BCB text tables. Text is generated to define inline code, a call to a compiler-generated subroutine, or a call to a library subroutine required for execution of these operations. This phase generates the text defining any of the five compiler-generated subroutines concerned with string handling that are called from text generated by this phase or Phase OC.

If an expression contains a variable declared with the COMPLEX attribute, Phase OX expands the text so that both the real and imaginary parts of the variable are included.

PHASE INPUT

Input to the phase consists of the main text stream, which is scanned sequentially. Entries in the variables and general dictionaries may be accessed if it is necessary to create a descriptor-descriptor for a complex variable.

PHASE OUTPUT

Output from the phase consists of the main text stream, modified as described in the description of phase operation. If compiler-generated string-handling subroutines are required, they are inserted in the text immediately following the first labeled statement header (SL). The subroutines that may be generated are:

- CGSCB - Compare long bit strings.
- CGSB0 - Test bits for zero, branch on not ones.
- CGSBB - Test bits for zero, branch on not zeros.
- CGSCL - Compare long character strings.
- CGSMV - Move long character strings.

Some of the text tables generated for these subroutines contain actual code sequences to be copied into the object module, rather than indicating the code sequences to be generated by the code generation phases.

Entries for descriptor-descriptors, and also for translate-and-test tables used in processing the INDEX and VERIFY built-in functions, may be made in the general dictionary.

The main text stream is scanned by use of the XNEXT macro. When the first SL statement header is seen, its text reference is saved to indicate the position at which any compiler-generated subroutines that are required should be inserted at the end of phase operation. When other text tables of possible interest to the phase are seen, control is passed to an appropriate routine for further examination as follows:

SCANBIFS routine - examines BIF text tables
STRNGF0 routine - examines PSV text tables
SCANBCB routine - examines BCB text tables
SCANBC routine - examines BC text tables
CP001 routine - examines PLUS, PPLUS, MINUS, PMINUS, MULT, DIVIDE, ASSN, CONV, BC (with non-string operands), sl, sn, sinit, and IASSN text tables.

If examination reveals that these text tables are of interest to the phase, they are processed accordingly. Otherwise, the scan is stepped to the next text table.

In addition to the types of text tables previously mentioned, NDX, OFFS, and PTSAT text tables are also examined for the presence of Q-temps. representing data types processed by the phase. A stack of active Q-temps. is built and maintained to enable the qualifications of a Q-temp. to be accessed whenever it is used as an operand in a text table processed by this phase.

Processing String Built-in Functions

If the IOP2 code byte of a BIF text table indicates that it is a reference to a string-handling built-in function processed by this phase, control is passed to the appropriate processing routine as follows:

VERIFY bif - VRFNCT routine
INDEX bif - INDEX00 routine
REPEAT bif - REPEATF routine
LENGTH bif - LENBIF routine
HIGH or LOW bif - HIGHLOW routine
STRING bif - STRNGF00 routine (this routine also processes STRING pseudovariables)

None of these routines generate calls to compiler-generated subroutines. The LENBIF and HIGHLOW routines always generate inline code in the form of ASSN text tables. The VRFNCT, INDEX00, REPEAT0, and STRNGF00 routines may generate inline code or calls to library subroutines, according to the attributes of the operands. The subroutine GENLIB is used to generate library calling sequences, using information set up in the GLOPND1 and GLNUM fields of XSTG, and in RE.

Processing BC and BCB Text Tables

BCB text tables, and BC tables with string operands are processed by the routine SCANCOMP. Where possible, they are replaced by text defining inline code. If the lengths of the fist two operands are known to be equal, the BC or BCB text table is replaced with a COMPARE text table followed by a GOTO text table with the appropriate branch condition. If the lengths of the first two operands are not equal, a COMPARE text table with the length of the shorter operand is generated, followed by a GOTO text table with a condition code that is the inverse of the

original branch condition. This is followed by another COMPARE text table, with the remainder of the longer operand and an equal length string of zeros or blank characters, and followed by a GOTO text table with the original branch condition.

Where the lengths of the operands make the generation of inline code unsuitable, a call is generated to a compiler-generated subroutine. If either of the operands is a varying string, a call to a library subroutine is generated.

Processing Expressions with Complex Operands

Routines starting at OX2 (the second module of the phase) examine text tables which may contain complex data operands. If a statement contains a complex data item, it is expanded, either by the generation of text-defining inline code or by a call to a library subroutine, into one or more statements which handle the real and imaginary parts of the item separately.

The text tables that are examined are PLUS, PPLUS, MINUS, PMINUS, MULT, DIVIDE, ASSN, IASSN, SINIT, CONV, and BC (with non-string operands). Each of the operands in a text table are examined in turn and for each operand a switch, 01SW, 02SW, or 03SW respectively, is set to indicate whether the data type is real, complex, or the real or imaginary part of a complex item. If all the operands are imaginary, or either real or the real part of a complex item, then no expansion is required. If the divisor operand of a DIVIDE text table is complex or does not have the FLOAT attribute, the GENLIB subroutine is called to generate a call to a library subroutine to perform the expansion.

If expansion is to be performed, a value set according to the indications of 01SW and 02SW is added to the value of the IOP1 code byte, and the result, TABNO, is used to index a table, TABAD. Each entry in TABAD is the address of an entry in a second table, CONV1. Entries in CONV1 are of varying length, and the number of significant bytes in an entry indicates the number of new text tables to be generated. Some of the entries have internal code bytes with values set to inhibit further processing if the result operand is real. The value of each other byte in the entry (except the end marker, X'FF') indicates an entry in a third table, INSTAB. Thus, a text table:

```
PLUS      B       C       A
```

in which B, C, and A are all complex data items, results in TABAD indicating an entry in CONV1 with the following format:

```
PLUS1 DC X'1DFDFE1FFF'
```

In this entry, the last byte, X'FF' is the end marker, and the two bytes, X'FDFE' are the internal code bytes mentioned previously. There are two significant bytes, X'1D' and X'1F', which point to two entries in INSTAB. Each entry in INSTAB is six bytes long, and defines a text table to be generated. The first two bytes define the IOP1 and IOP2 code bytes of the text table. The next three bytes, each of which can have a value in the range X"01" to X"0F", describe operands 1, 2, and 3 respectively. The last byte contains an end marker, X'FF'. The entries referred to in the foregoing example are:

```
X'6300010305FF' 1D
X'6300020406FF' 1F
```

Each of these entries defines a PLUS text table to be generated. In the first, the real part of the original operand 1 is added to the real part of the original operand 2, and the result is stored in the real part of the original operand 3. The second text table indicates a similar operation using the imaginary parts of the original operands. The original text table is changed to NULL.

Routine SCANLAB/SCANSN

This routine examines information concerning the last statement and determines:

1. Whether a VDA was used and if any previous VDAs should be freed before the VDA is obtained.

2. Whether any previously obtained VDAs should be freed in statement.

DATA CONVERSION PROCESSING (PHASE KX)

The main function of this phase is to analyze the conversions of data types that are required for various operands in the text, and to modify the text so that it indicates the object code required to enable the conversions to be performed at execution time. The code to be generated enables the conversion to be done either by inline code or by a call to a library routine that performs the conversion. The method is chosen by this phase according to two factors - the type of conversion required, and the type of optimization specified in the compiler options.

The phase also examines the data types of the operands in a number of other types of text tables, and sets code bytes or flags in those text tables to indicate the format of the object code required.

Picture specifications in the general dictionary are examined and converted to a format required by use by library routines at execution time.

PHASE INPUT

The phase scans the main text stream, searching for text tables that are processed by the phase. Because some of the preceding phases are optionally executed, depending upon the contents of the PL/I source program or the compiler options that are specified, text input to the phase may be the text output from Phase KV, OM, KK, OC, or OX. The logical sequence of the text may be indicated only by the contents of the forward chain fields in the text tables or, if the text has been processed by the phases in the global optimization stage, flow-unit header tables, hash tables, statement header tables, and end-of-program tables may also be used to follow the logical sequence. The text tables processed by this phase are: CONV(convert), PLUS(add), PPLUS(prefix plus), MINUS(subtract), PMINUS(prefix minus), DIVIDE, COMP(compare), ASSN(assign), GOTO, SUBS(subscript) and DINC(Do-loop increment).

The general dictionary entries for picture specifications are scanned and processed.

PHASE OUTPUT

Output from the phase consists of the input to the phase, with modifications and insertions made where necessary.

Text tables processed by this phase (except the CONV text tables) are modified but do not cause other text tables to be generated. Each CONV text table has a value set in the IOP2 field, and may have additional text tables created before and/or after it.

During processing, bits are set in the XLIBSTR field in XCOMM to indicate library modules for which calls must be generated.

The picture specification entries in the general dictionary have characters replaced in situ.

Processing Picture Specifications.

The entries for picture specifications in the general dictionary are scanned sequentially. Each picture specification is translated in situ into a format acceptable to library routines at execution time. The phase contains a table that indicates replacement characters and formats equivalent to the picture specifications in the dictionary entry. Items in the table are used as direct replacements for the existing items in the dictionary entries.

Processing Text Tables

Although the physical sequence of the text tables in the main text stream input may vary according to which phases have been executed, this phase scans the text tables in logical sequence by use of the XNEXT macro routine. A single sequential scan is made, during which text tables processed by the phase are detected. As each text table is detected, a routine that processes that type of text table is branched to.

The processing of CONV text tables is described later. For all other text tables, the attributes of the operands are found by examination of the DEDs. Constant operands are treated as variables during this processing and, where necessary, the target operand is examined to determine their object-time attributes. The examination of the attributes indicates the particular sequence of object code required. Bits are set in the second byte of the operator (IOP2) and/or the status flag bytes (IST2) of the text table to indicate the code skeleton to be used by the code generation phases.

Processing Conversions.

When a CONV text table is detected, the CONVERT routine is branched to. This routine contains a number of subroutines, each of which processes a particular combination of source attributes (i.e., the attributes of the first operand in the text table), and target attributes (i.e., the attributes of the third operand in the text table). The following eight attributes or combination of attributes are recognized as being unique and valid in their application for conversion purposes:

1. BIT
2. CHARACTER
3. FIXED DECIMAL
4. FLOAT PICTURE DECIMAL
5. FLOAT
6. FIXED PICTURE DECIMAL
7. PICTURE CHARACTER

8. FIXED BINARY

The main routine examines the source attributes and branches to the appropriate one of eight routines, each of which processes conversion tables with a particular source attribute/s. The selected routine then examines the target attributes. For each combination of source and target attributes, a branch is made to a particular subroutine. There are sixty of these subroutines. A few subroutines process more than one combination of source and target attributes e.g., processing of CHARACTER to BINARY conversion uses the CHARACTER to DECIMAL conversions routine.

Each subroutine determines whether the required conversion is to be performed by the generation of inline code, or whether a library routine is to be called to perform the conversion. In order to determine the processing required, the prevailing statement options (indicated in XCOMM) are checked, and the text table is examined.

When possible, code for inline conversion is generated. However, under certain conditions, for example the possible presence of invalid data types in PICTURE specifications, adjustable or varying character strings, and SIZE or CONVERSION conditions enabled, inline conversion code cannot be generated and a library-call conversion is initiated as described below.

In some cases, a hybrid system of conversion is performed, i.e., code is generated to perform some of the conversion process in line, but this code includes a call to a library routine to perform a particular operation in the processing. When the particular method of performing the conversion has been decided, the processing is performed as described below.

Library-call Conversions: All indications for the code required for library-call conversions are generated in a similar manner by the subroutines that test for the required type of conversion. The IOP2 code byte in the CONV text table is set to a value of X'01' and the first byte of the second operand is set to a value of X'2F'. Both these settings indicate to later phases that code for a library call is required. The index number of the particular library module required is set in other bytes of the text table second operand. The appropriate index bits are also set in XLIBSTR (in XCOMM) to indicate the main entry point, and all possible secondary entry points, of the library module. No other text tables are generated.

Inline Conversions: A further branch is made to a subroutine that generates text tables to indicate the particular code sequence required. This subroutine examines the statement prefix options, which have been saved in a phase save-area during the text scan, to see if the enablement of the NOSIZE and CONVERSION conditions affects the attempt to perform conversion under certain conditions.

Whenever possible, standard text tables (e.g., ASSN, MOVE, etc.) are used in the text sequence, and only the CONV text table will generate code peculiar to the particular conversion. The code skeleton required is indicated by setting a value (X'02',X'03',---X'16') in the IOP2 code byte of the CONV text table. A number of conversions are performed by generating the code for two fundamental conversions in series.

The output code required for each type of inline conversion is illustrated in the published listing, and is also described in the publication: DOS PL/I Optimizing Compiler, Guide to Execution.

Hybrid Conversions: Some of the processing required for the conversion from FLOAT to CHARACTER can be performed by code inline. This conversion requires an interpretive routine to analyze the floating-point data, in order to generate the correct scale factor. This part of the processing is best performed by a library routine. A character string of the correct length is set up before calling the library routine, and the subsequent string assignment is processed in line.

The four phases that comprise this stage are only loaded and executed if the OPTIMIZE (TIME/2) (or OPT(TIME/2)) compiler option is specified. The first three phases in the stage, Phases OA, OE, and OI, extract information about variables and their usage in the program, and about the paths along which control can flow within the program. This information is collected in special tables in the text and in dictionary entries. The last phase in the stage, Phase OM, uses this information to enable it to detect situations where the text can be modified or reordered so that it defines a more efficient sequence of code. The extent to which this form of optimization can be performed varies according to the efficiency of the source program, according to the content of the source program (e.g., the number of blocks, the number of variables, etc.), and also depends upon whether the REORDER option has been specified in the source program.

GLOSSARY OF TERMS USED IN GLOBAL OPTIMIZATION

The descriptions of the processing performed by the phases of the global optimization stage contain a number of terms that are used almost exclusively within those descriptions. For convenience of the reader, a number of those terms are explained and illustrated in this special glossary. The terms are arranged in logical sequence of definition rather than in alphabetic order.

flow unit: a sequence of text defining a sequence of code that has no intermediate branching points, and therefore can only be logically entered at the beginning and left at the end. (Each flow unit is allocated an identifying number by Phase OI.)

entry unit: a flow unit that begins with a block-entry statement (PROCEDURE, BEGIN, ON-BEGIN, or ENTRY) or which has a label that can be branched-to from a dynamically contained block (procedure block, begin block, or on-unit).

forward connector (of a flow unit, F): a flow unit to which F can pass control directly. (See figure 2.21.)

backward connector (of a flow unit, F): a flow unit which can pass control directly to F. (See figure 2.21.)

level number (of a flow unit, F): a number that is one greater than the smallest number of flow units through which control must pass to get from an entry to F. (See figure 2.21.) Entry units have a level number of one.

Flow units (PL/I source) and identifying numbers		Forward connectors	Backward connectors	Level numbers
ENTUNIT: BEGIN; A = 5; B = 0;	1*	2	0	1
LOOPENT: IF B = 0	2	3,4	1,7	2
THEN C = A+B;	3	5	2	3
ELSE C = A/B;	4	5	2	3
IF C = 1	5	6,7	3,4	4
THEN GOTO EXIT;	6	8	5	5
PUT LIST (C); B = B+1; GOTO LOOPENT;	7	2	5	5
EXIT: PUT LIST ('A=B'); END;	8	-	6	6
* indicates entry unit				

Figure 2.21. Example showing flow units, forward and backward connectors, and level numbers

back dominator (of a flow unit, F): the flow unit nearest to F through which control must flow before F receives control for the first time. (See figure 2.22.)

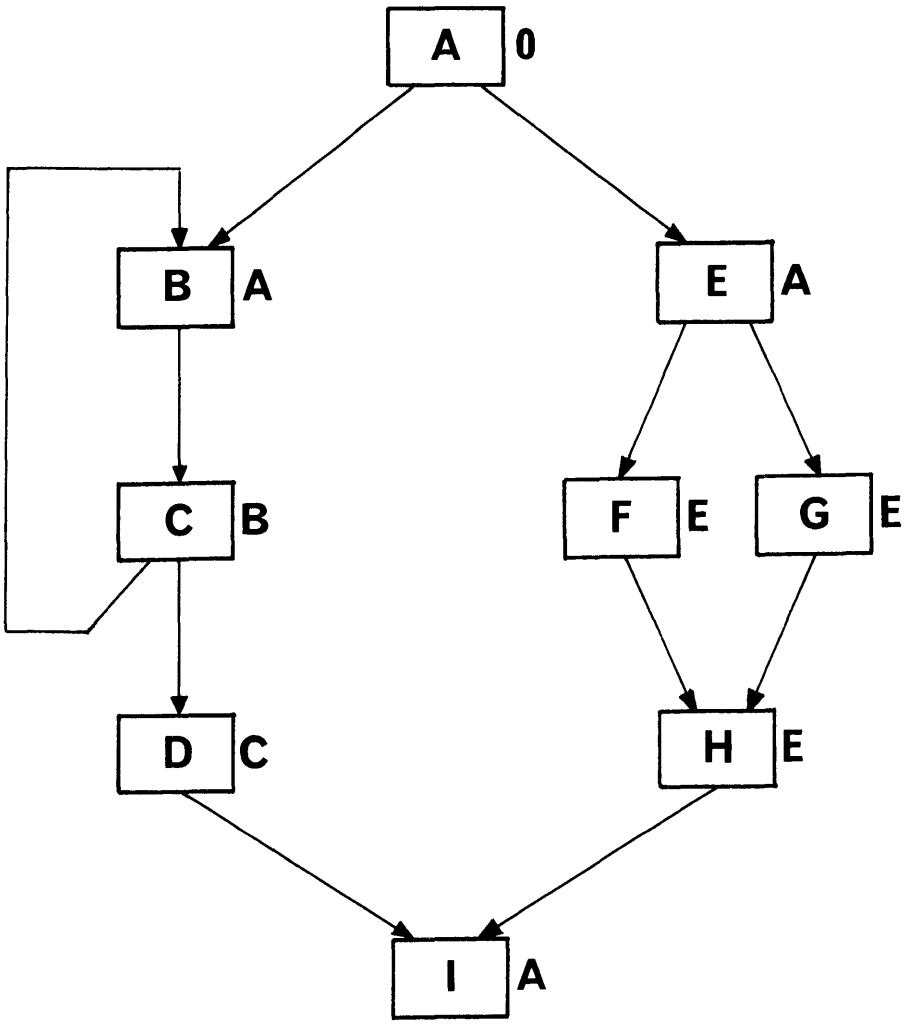


Figure 2.22. Illustration of back dominators

loop entry unit: a flow unit that starts with the entry point of a loop. A flow unit is a loop entry unit either if it has itself as a backward connector or if it has one or more backward connectors for which it is the back dominator. (See figure 2.23.)

back target (of a loop): the flow unit nearest to the loop entry unit through which control must pass before the loop is entered. A loop back target is the back dominator of the loop entry unit. (See figure 2.23.)

back target (of a flow unit, F): the back target of the loop in which F is contained.

forward target (of a loop): the flow unit to which control passes when the execution of a loop is complete. (See figure 2.23.)

loop depth number: a number associated with a loop to indicate its depth of nesting; equal to the number of containing loops plus one. (See figure 2.23.)

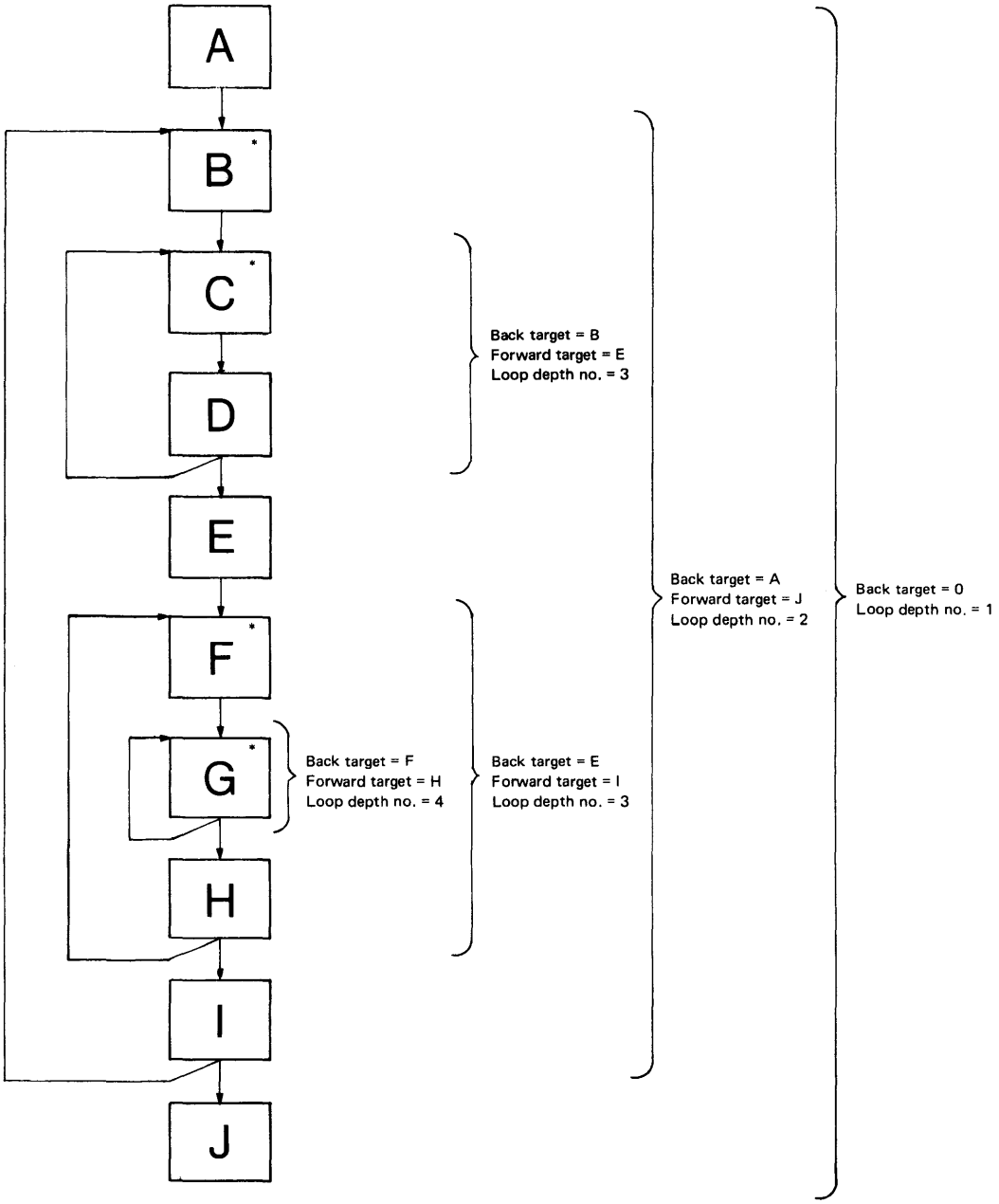


Figure 2.23. Illustration showing back targets, forward targets, and loop depth numbers

EXTRACTION OF ALIAS AND CALL INFORMATION (PHASE OA)

As a preliminary step towards global text optimization, Phase OA determines which variables can be referred to by alias names, and which blocks can be called from other blocks in the program.

The information is collected in value lists which are used to build value list entries in the general dictionary. An entry is made for each parameter, locator, label variable, entry variable, defined variable, and each program block.

Entries containing summaries of the alias information and the block-calling information for the whole compilation are also made in the general dictionary.

PHASE INPUT

The main text stream is scanned sequentially for text tables containing the following features:

Label assignments
Locator assignments
Entry-variable assignments
Argument lists
Function references
Calls to PROCEDURE, BEGIN, and ON-BEGIN blocks

The variables dictionary is scanned for entries for variables with the DEFINED attribute, and the variables and general dictionaries are accessed as required.

PHASE OUTPUT

The main text stream is not altered by this phase. Value list entries are built in the general dictionary (see figure 5.26). Each entry contains a bit vector which indicates either:

1. The variables, label constants, and blocks (as opposed to internal entry points) that can be accessed under one or more alias names, by means of a locator variable, parameter variable, label variable, or entry variable.

or

2. The blocks that can be called, either directly or indirectly, from a particular block in the compilation.

A summary of the alias information for the whole compilation is stored in an entry in the general dictionary, and the reference of this entry is stored in the XALIAS field in XCOMM.

For each block in the compilation, an Optimization Entry is built in the general dictionary (see figure 5.25). This entry is mainly a skeleton entry for use by other phases in the stage, but this phase inserts in it a value list indicating all blocks that can be called from the block to which the entry relates.

For each variable that has a value list entry built in the general dictionary, the reference of that entry is inserted in the variables dictionary entry.

The main scanning routine, TSCAN1, uses the XNEXT macro to scan the main text stream sequentially in order to detect all the items listed in the description of input to the phase. On the first appearance of one of these items, a value list entry is created in the general dictionary. Each entry contains a bit vector, in which bits are set to indicate a particular item. Thus, the setting of the nth bit in a value list bit vector can be used to indicate a reference in the nth entry in the general dictionary, the nth entry in the variables dictionary, or the nth block in the compilation. Bits are set in the appropriate value list entry on each appearance of an item, so that each entry can indicate a number of aliases or a number of blocks that can be called.

Because the information collected in a value list cannot be complete until all locator assignments, label assignments, entry assignments, argument/parameter matches, or CALL statements and function references have been seen, temporary data areas are created in the phase working storage to enable sifting and commoning of collected information. These temporary data areas are referred to as the Primary Value List Transfer Table, the Secondary Value List Transfer Table, and the Call Table. When the text scan is complete, the variables dictionary is scanned for entries for defined variables. Value list entries are made for these variables, and entries are also made in the primary transfer table as required. When the dictionary scan is complete, the information collected in the temporary data areas is reorganized, and the value list entries in the general dictionary are completed. Finally, summaries of the alias information and the block-calling information for the whole compilation are entered in the general dictionary.

Note: In some cases of very large or very complex source programs, either of the transfer tables or the call table may overflow. This prevents further collection of information required for global optimization, and this makes execution of Phases OE, OI and OM impossible. If this happens, a diagnostic message is generated, flags are set to indicate that global optimization has not been performed, and an XPST macro statement is invoked to have Phase KK loaded.

Value Lists for Variables

Value lists are built for label variables, locator variables, entry variables and parameter variables with the following exceptions:

1. Label variables with label lists have value lists built by Phase GE during the dictionary build stage.

2. Base variables serve only to give the attributes of the data being accessed. They have no unique storage allocated to them and the address of the data is given solely by the pointer that always precedes a based variable in text. For example, P -> V does not equal Q -> V unless P = Q. For optimization purposes, only the locator is significant, and based data does not appear in value lists.

For each variable that has a value list built, a reference is inserted in its variables dictionary entry indicating the location of its associated value list entry in the general dictionary.

Label Variables: Each label variable has a value list built, in which bits are set to indicate the label constants it can represent. The coordinate of each bit set indicates the reference of a general dictionary entry for a label constant (all label constant entries are grouped together at the beginning of the general dictionary). The length of label variables value lists, in bytes is (number of

labels+16)/18. The last byte in the list is used to indicate if the associated label variable could transfer control from the current external procedure (or a contained block) to another external procedure.

Entry Variables: Each entry variable has a 32-byte value list built, in which bits are set to indicate the blocks that the variable can represent. The block-count values in the entry-constant entries are used.

Parameters: Each parameter has a value list built in which bits are set to indicate variables in argument lists that the parameter can represent. Bits are not set in the value list if an argument passed is a based variable, a locator, a label variable, a variable with the DEFINED attribute, or a parameter in its own right.

The coordinate of a bit in a value list corresponds to the effective number of an entry in the variables dictionary. Value lists for variables are 32 bytes long, enabling the first 256 variables in the program to be represented. The technique used to refer to variables outside this range is described later.

Locator Variables: Each pointer or offset variable has three value lists built for it. The first is a variables value list (32 bytes long) in which a bit is set whenever the address of a non-based variable which is not a locator, label variable, a variable with the DEFINED attribute, or a parameter, is assigned to the locator.

```
e.g., P = ADDR(V);
```

where P is a pointer and V is a non-based variable. In this case, V is considered to be a value of P so the bit for V is set in the value list for P.

The second value list built for locator variables is one in which bits are set to indicate any block (as opposed to internal entry points) that can be accessed by the locator. The third value list has bits set to indicate any label constants at which the locator can point.

Use of Value List Transfer Tables

The nature of information extracted during the scan of the text tables may make it impossible for every alias to be indicated in a particular value list at that time. To overcome this problem, certain value list information is stored in temporary data areas, known as Value List Transfer Tables, that are built in working storage during the scanning process. When the scanning process is complete, the information stored in the transfer tables is analysed and used to complete the value lists in the dictionary. Two value list transfer tables are used, the Primary Transfer Table and the Secondary Transfer Table.

The Primary Value List Transfer Table: Entries are made in the primary transfer table when:

1. Assignment is made to a non-based label variable of another non-based label variable, or when a non-based entry variable is assigned to another non-based entry variable.

2. An argument that is passed to a parameter is a based or defined variable, or is itself a parameter.

3. Assignment is made to a locator of any value other than the address of a non-based variable which is not a parameter, label variable, locator, or a variable with the DEFINED attribute.

The format of an entry in the primary transfer table is as follows:

FRMREF is the dictionary reference of the value list from which information is to be transferred to the value list at the dictionary reference given at TOREF.

TOFLG and FRMFLG indicate the type of variable, e.g., locator, label-variable, defined variable, etc.

An example of the use of the primary transfer table is:

```
LABA=LABB;
```

where LABA and LABB are both label variables. At the time the assignment statement is scanned, all the label constants that can be assigned to LABB may not be known and therefore not indicated in the value list of LABB. An entry is made in the primary transfer table and, when scanning is completed, any label constant alias indicated in the value list of LABB can also be set in the value list of LABA.

The Secondary Value List Transfer Table: An entry is made in the secondary transfer table whenever an assignment is made to or from a based label variable or entry variable. The format of the entry is similar to that for an entry in the primary transfer table. Because of its use, one of the value lists referred to (in either TOREF or FRMREF) will always be that of a locator and the other will be the value list of a locator, a label variable, or an entry variable, or the value-list coordinate of a label constant or entry constant.

Transference of Variables Value Lists: When the text scan and initial building of value lists is complete, entries in the transfer tables are expanded and value lists transferred as required. The TRNS1 routine expands each entry in the secondary transfer table into several entries which are added to the primary transfer table. Each entry in the secondary transfer table is examined in turn. If the FRMREF field refers to the value list of a locator, it is replaced by a list of value lists of label variables or entry variables that can be addressed by the locator. This list is built by searching the primary transfer table for label or entry variables or other locators whose value lists are to be transferred to that pointer. If the FRMREF field refers to the value list of a label or entry variable, no expansion is necessary. The TOREF field is expanded in a similar way. All possible combinations, of 'To' and 'From' value list references are then added to the primary transfer table. When this has been done for entry in the secondary transfer table, that table is discarded.

The TRNS2 routine scans the primary transfer table and accesses the value lists referred to in its entries. A logical OR operation is performed between the value list to be transferred from and the value list to be transferred to, thus effectively transferring the contents of one value list to the other. A flag is set to indicate which value lists are changed. Successive scans are made in which only the value lists that are flagged are accessed. Processing is complete when a scan is made during which the OR operations do not change any value lists. The primary value list transfer table is then discarded.

Alias Information Summaries

The amount of detailed information that can be contained in value lists is limited by the size of the value lists. Value lists for label

variables are of variable length and can indicate the aliases of all label variables in the program. Value lists for other variables are 32 bytes long and can only indicate aliases for the first 256 variables in the program. For this reason, bit vectors are built during the initial scan of the text by the TSCAN1 routine, summarizing the alias information for a greater number of variables than those for which value lists can be built. Bit vectors are built showing the following information:

1. Those variables with dictionary references less than 2048 that have value lists.

2. Those variables with dictionary references less than 256 that appear in value lists.

3. Those variables with dictionary references between 256 and 2047 that do not appear in any value list but would do so if value lists were larger.

On completion of processing by the phase, these summaries of alias information are stored in the general dictionary, with the reference of the entry being stored in the XALIAS field of the communication area.

Value Lists for Blocks

Calling information for blocks is collected initially in a temporary data area named the Call Table. When the TSCAN1 routine detects a CALL or FNCT text table, an entry is made in the call table showing the block count of the calling block and the block count of the called block. The block count used is that set up during the syntax analysis stage. If an invocation of an entry variable is detected, the entry made in the call table consists of the block count of the calling block and the dictionary reference of the value list entry for the entry variable.

The TRNS3 routine builds in working storage a value list for each block, consisting of a bit vector with all bits initially set to zero. The bit vector is 32 bytes long, enabling the maximum number of blocks (255) to be represented. The call table is scanned and, for each entry, the bit representing the called block is set in the value list of the calling block. When an entry for an entry variable is encountered, the value list entry is accessed and the call table entry is expanded into series of calling block/called block entries before any value list bits are set. When this scan is complete, the Call Table is scanned again and, for each entry, a logical OR operation is performed between the value list of the calling block and the value list of the called block. In this way, bits are set in the value list for the calling block showing which blocks are in turn called by the called block and thus indirectly called by the calling block. Scans of the call table are repeated until no value lists are changed by a logical OR operation.

During the text scan, an Optimization Entry is created in the general dictionary for each block (see figure 5.25). The value list for each block is entered in the optimization entry and the reference of the optimization entry is stored in the block header entry for the block. The call table is discarded.

The following fields are completed at this time:

YALABL(2 bytes)	Dictionary reference of first user-supplied label in the block.
YALABL1(2bytes)	Dictionary reference of first compiler-generated label associated with label-array initialization in the block.
YALABS(2 bytes)	Number of user-supplied labels in the block.
YALABS1(2bytes)	Number of compiler-generated labels associated with label-array initialization in the block.

EXTRACTION OF VARIABLES USAGE AND FLOWPATH INFORMATION (PHASE OE)

The functions of Phase OE are preliminary steps in the global optimization process. They are:

1. Reorganization of text. The text stream is reorganized so that all text tables within a logical flow unit are physically grouped together. Flow Unit Header (FUH) tables and hash tables are created in the text stream.

2. Extraction of variables usage and flow path information. Information indicating which variables are used in each logical flow unit, and how they are used, is collected in each flow unit header table. Information indicating possible flow paths between logical flow units is similarly collected. This information is consolidated for each block in the program and stored in block optimization entries in the general dictionary. A summary of information about variables used or set in on-units is collected in one entry for the whole program (see figure 5.24).

PHASE INPUT

The text stream output from Phase OA is scanned in logical sequence. The physical sequence may not coincide with the logical sequence because of the expansion of text onto overflow pages during the statement processing stage.

The variables dictionary is accessed for the general dictionary references of value lists created by Phase OA.

The general dictionary is accessed for value lists and block optimization entries created by Phase OA.

PHASE OUTPUT

A reorganized text stream is built in which the Type-2 text tables are physically arranged according to their logical sequence within logical flow units. A flow unit header table (see figure 5.98), is built for each flow unit. Information indicating the usage of variables within the flow unit, flow paths out of the flow unit, and the label at the head of the flow unit is stored in the FUH. Flow units are chained together and text pages within a flow unit are chained. Hash tables are inserted after each FUH table and at the beginning of each text page. Forward and backward chains link all text tables to the hash tables. One type of text table may be changed by this phase.

The optimization entry in the general dictionary for each block in the program is completed. Each entry contains information about the usage of variables in the block, possible flow paths from the block, and usage of variables within blocks that can be called. (The format of the optimization entry is shown in section 5, "Data Area Layouts.")

If the program contains on-units, a special entry is made in the general dictionary in which information about the usage of variables within all on-units and within blocks called from on-units is stored.

This phase also sets up the appropriate bits in the IFOF3 field of the FUH table (see figure 5.99) to mark the first flow unit in a block, and the first and last flow units in a DO group.

Reorganization of Text

The SCAN1 routine uses the XNEXT macro to scan the text tables in the input text stream, and calls the appropriate routines or subroutines to perform processing as required. At the beginning of the scan, a new or discarded text page is acquired for the building of a reorganized text stream, and further text pages are acquired as necessary. As each page is acquired, an overflow-page index table is inserted after the page header, and the TA of the page is inserted in the REF0 field of the ISPILB field. Thus each new page is treated as an overflow page, and the forward chain field (IFCHN) of all the text tables later copied or inserted into the page has the initial character '8'.

When the scan encounters a statement-header text table, its ISF field is examined. If this field has been set to X'80' by Phase KV, it indicates that the text table is the beginning of a flow unit, and the FLOW1 routine is branched to. This routine creates skeleton 164-byte tables, referred to as flow-unit header tables (FUHs), and inserts one of these tables in the reorganized text stream at the beginning of each flow unit. The format of an FUH is shown in figure 5.98. Some of the fields in each FUH are completed during processing by this phase; other fields have values inserted by later phases in the stage. All flow-unit headers are linked by a forwards pointing chain in the IFOUNC field.

Immediately following each FUH, a 32-byte hash-chain table is inserted. The format of a hash-chain table is shown in figure 5.100. These tables are so called because Phase OM uses a hashing technique to aid searches for particular types of text tables, and the heads of the hash chains are inserted in the IHASHC fields of these tables (by Phase OM). If a flow unit extends over a page boundary, a hash-chain table is also inserted after the overflow-page index table of the second and any subsequent pages used by the flow unit. Thus a page may contain more than one hash-chain table, even if it contains only one FUH. A typical arrangement of these tables is shown in figure 2.24. The IBPCH, IBCHN, and IFCHN fields of the hash-chain tables are used as described in following paragraphs.

The CHTAB routine copies text tables in logical order in the new text tables, so that the physical sequence of text tables coincides with their logical sequence. The IBCHN and IFCHN fields of each text table are used to create backwards and forwards pointing chains linking all text tables. For the last text table in a flow unit or in a page, the IBCHN field points to the hash-chain table preceding it. The IFCHN field of the last text table on a page points to the nearest preceding hash-chain table.

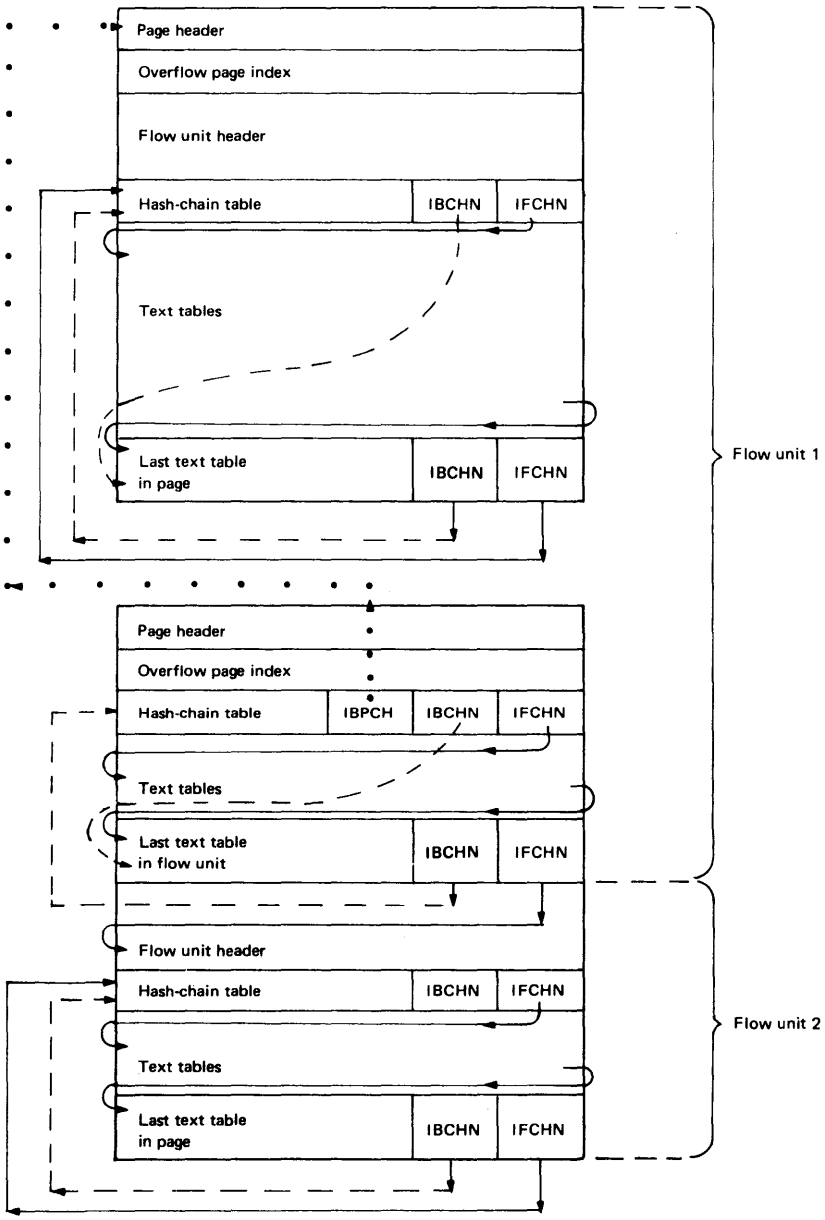


Figure 2.24. Arrangement of tables and chaining in main text stream reorganized by Phase OE

Extraction of Variables Usage Information

While the SCAN1 routine is building the reorganized text stream, it checks on the position of variables in each text table. With a few minor exceptions, the position of a variable within a text table enables its usage within a flow unit to be classified as follows:

- Used - if it appears as operand 1 or operand 2 in a text table
- Set - if it appears as operand 3 in a text table
- Busy on entry -- if it is used in a flow unit without first being set in the same flow unit.

This classification of variables enables information about their usage within each flow unit to be collected in fields in the flow unit header table as follows:

IFOVU - Variables used in the flow unit.

IFOVS - Variables set in the flow unit.

IFOBEN - Variables busy on entry to the flow unit.

IFOVS2 - Variables set more than once in the flow unit.

Each of these fields contains a 32-byte bit vector. The offset of a bit from the start of a field can be used to indicate a particular entry in the variables dictionary, i.e., the nth variable in the variables dictionary can be referred to by setting the nth bit in a bit vector.

Because of the fixed length of the FUH fields, direct usage of the first 256 variables only can be indicated in an FUH. However, for any variable that is active in a flow unit, any relevant alias information collected by Phase OA (in the value list entries and the alias information summaries in the general dictionary) is examined and incorporated. Similarly, examination of alias information can affect the usage classification of a variable. For example, when determining whether a particular variable has been used before being set, the value list for that variable (if it has one) is compared with the IFOVS (variables set) string. Only if no item in the value list has been set in IFOVS can the variable be classified as busy on entry, in which case each item indicated in its value list, as well as the bit for the variable itself, is set in the IFOBEN string.

When the variables in all the flow units within a block have been processed, two bit vectors, each 32 bytes long, are built in the block optimization entry in the general dictionary. The variables usage information collected in each FUH within the block is used to complete the following bit vectors:

YAVUB-Variables used in the block

YAVSB-Variables set in the block

Extraction of Flow Path Information

The CALLR1 and BRCH2 routines extract information from each flow unit about the entry point and about flow paths which can be followed at exit. This information is stored in each FUH and consolidated in the block optimization entry in the general dictionary.

FLOW UNIT ENTRY INFORMATION: If a flow unit starts at an ENTRY statement, this fact is detected by Phase KV and indicated to Phase OE by setting a flag in the third bit of the ISF field of the statement header table. Phase OE passes this information to later optimization phases by setting the first bit of the flag byte IFOFL2 in the FUH.

Except in some cases of flow units following CALL statements and function references, the entry point to a flow unit will be indicated by a label. Phase OE sets a code byte IFOCDE to indicate whether this label is user-supplied or compiler-generated. If it is a user-supplied label, the dictionary reference of the label is inserted in the IFOLAB field in the FUH. If it is a compiler-generated label, the identification number is inserted. If a flow unit is the first flow unit in a block, then the label on the PROCEDURE, BEGIN, or ON statement is external to the block and cannot be used as the flow unit label. In this case, the block-count number is inserted in the IFOCDE code byte and the dictionary reference of the first user-supplied label in the block is inserted in the IFOLAB field. As this label will usually indicate the start of the next flow unit, its dictionary reference will appear in the IFOLAB fields of two flow unit header tables.

FLOW UNIT EXIT INFORMATION: At the end of a flow unit, flow may be direct to the next flow unit in the text stream or it may branch to one or more other flow units. Phase OE examines the text tables and sets a flag byte, IFOFL in the FUH, to indicate either direct or branching flow, or both. When set, the flag bits of IFOFL give the following indications:

Bit no.	Indication when set
0	Flow passes directly to the next flow unit in the text stream. (This bit is always set unless branching to other than the next flow unit is unconditional.)
1	A branch to other than the next flow unit exists.
2	Two branches exist.
3	The flow unit ends at a CALL statement or function reference.
4	The flow unit ends at a RETURN or END statement
5	This is the last flow unit in a block.
6	The flow unit is the beginning of an iterative do-loop.
7	Spare (set to 0 before Phase OI).

Where branches from the flow unit exist, the BRCH1 routine inserts in the FUH information about the labels external to the flow unit that can be branched to. The information is stored in one or two 5-byte label descriptors, according to whether one or two branches exist. The format of each label descriptor varies according to whether the item branched to is a label constant, a compiler-generated temporary label variable, or a label variable.

If the branched-to point is a label variable, the relevant GOTO table will have been changed to a GOOB (go out of block) table by Phase II as the label constant values will not have been resolved at the time of processing by that phase. Phase OE examines the value list entry for the label variable and changes to GOOB text table to a GOTO text table if the branched-to point can only be internal to a non-recursive block.

Consolidation of Block Information

When the SCAN1 routine detects the end of a block, control branches to the BLOCK1 routine. This routine examines the information stored in each FUH with the block and uses it to complete fields in the block optimization entry (built by Phase OA for each block) in the general dictionary.

Much of the information is stored in fields organized as bit vectors. The fields completed when SCAN1 detects the end of a block are:

YAVUB (32 bytes)	Variables used in the block.
YAVSB (32 bytes)	Variables set in the block.
YAXLAB (variable)	Labels in other blocks that are branched-to from within the block.

When these fields have been completed for every block in the program, the BLOCK1 routine accesses the optimization entry for the first block in the program. Using the information inserted by Phase OA in the YABCB field, which indicates the blocks called by a block, it completes other fields in the optimization entry. By accessing the optimization entries for each block called, and performing logical OR operations on the bit vectors in appropriate fields, the following fields are completed:

YAVUBCB (32 bytes)	Variables used in blocks called from the block.
YAVSBCB (32 bytes)	Variables set in blocks called from the block.
YAXLAB (variable)	1. Labels not internal to blocks called from the block but branched-to from within these called blocks. 2. Labels in this block that are branched-to from other blocks.

(Note: YAXLAB contains three variable length bit vectors. They each allow one bit for each label constant in the program and are rounded up to the next complete byte. The total length of YAXLAB is shown at YALEN.)

The process is repeated for each block in the program.

Extraction of On-unit Information

When the SCAN1 routine detects an on-unit (indicated by an ONS text table), it checks whether it is an I/O or AREA on-unit or a computational on-unit. Appropriate bits are set in the YAFLAG field in the block optimization entry.

When the optimization entries for all blocks have been completed, information about the usage of variables in on-units and blocks branched-to from on-units is extracted from the optimization entries. Such information for the whole program is stored in a single entry in the general dictionary, and forms part of the entry for the alias information summary built by Phase OA. The reference of the entry is held in the XALIAS field of the communication area. The fields completed by Phase OE are:

YACONU (32 bytes)	Variables used in computational on-units and in blocks called from them.

YACONS (32 bytes) Variables set in computational on-units and in blocks called from them.

YAIONU (32 bytes) Variables used in input/output or AREA on-units and in blocks called from them.

YAIONS (32 bytes) Variables set in input/output or AREA on-units and in blocks called from them.

YAGOBL (256 bytes) Labels that are branched-to from on-units, or from blocks called from on-units, and which cause a 'go out of block'.

On completion of this process, control is passed to Phase OI.

FLOW ANALYSIS (PHASE OI)

This phase continues the process, started by Phase OA and OE, of extracting information required for efficient global optimization of the program by Phase OM. Phase OI examines the information (extracted by Phase OE) about the possible paths of control flow between flow units, and expands it. It then identifies those flow units which are contained in loops, and chains the text for each block so that it can be scanned by Phase OM from the innermost level of loops to the outermost level. Having performed this detailed flow analysis, the phase is then able to modify the information about the usage of variables (extracted by Phase OE) so that it is more accurate and detailed.

If it is not possible to complete flow analysis on a block basis, it will then be performed on a DO group basis.

PHASE INPUT

Input to the phase consists of the main text stream, in which only the flow unit header tables (FUHs) are examined. The phase also accesses entries for label variables in the variables dictionary, and block optimization entries in the general dictionary.

PHASE OUTPUT

On output from the phase, the main text stream is unaltered except for the flow unit headers, which are changed to the format shown in figure 5.99. The modified format contains the information obtained during the flow analysis performed by the phase.

Existing dictionary entries are not changed by the phase, but a loop-data entry for each loop detected during the flow analysis is inserted in the general dictionary. The format of a loop-data entry is shown in figure 2.25.

PHASE OPERATION

Note: A number of the terms used in this description of phase operation are explained or illustrated under the heading, "Glossary of Terms Used in Gobal Optimization" in the introduction to the description of the global optimization stage.

Use of Tables and Lists

At the beginning of the phase, a number of tables and lists are set up in the phase working storage area (XSTG), or in text pages specially acquired for this purpose, for the collection and storage of information during processing. These tables include a flow unit information table (FUIT), a forward connector table (FCT), a backward connector table (BCT), a label/flow-unit-number table (LFUN), and a number of 32-byte bit strings. A stack is used by a number of routines for various purposes.

The flow unit information table (FUIT) consists of up to 256 18-byte entries. Each entry, which relates to one flow unit, has the following format:

Byte no.	Symbolic name	Field content (Some fields are temporarily used for other purposes)
0-4	FUITR	Text reference of flow unit
5	FUIDT	Flag to indicate 'drop through' flow
6-7	FUIFC	Pointer to FCT entry
8-9	FUIBC	Pointer to BCT entry
10	FUIBD	Identifying number of back dominator
11	FUIBT	Identifying number of back target
12	FUIDN	Depth number of loop
13	FUILC	Loop chain
14	FUICH	Flow-unit chain
15	FUILN	Level number
16	FUILEP	End-of-loop pointer
17	FUIFL	Flags

Bits in the FUIFL field may be set to give the following indications:

Bit no.	Symbolic name	Indication when set
0	INITUN	Unit has a true back target (initialization unit)
1	NONLOP	Unit is not contained in a loop
2	LOOPEN	Unit is a loop entry unit
3	ENDLOP	Unit is the last in a loop
4	DOLOPN	Unit heads a DO-loop
5	ENDLCH	Unit is at the end of a loop chain
6	IMABAT	Unit is a back target
7	-	Not used

The forward-connector table (FCT) and the backward-connector table (BCT) each hold up to 1024 two-byte entries. Each entry has the following format:

Byte no.	Bit no.	Content or indication when set
1	0	First entry in the list of forward/backward connectors for this flow unit
	1	Last entry in the list of forward/backward connectors for this flow unit
	2	Control drops through to the forward connector, or control drops through from the backward connector
	3	Control is passed by a branch out of the block
	4-7	Not used
2	-	Identifying number of forward/backward connector

The forward connector table is used to hold provisional information before the foregoing format is adopted.

The label/flow-unit-number table (LFUN) is used to relate a label constant to the flow unit on which it appears. The table consists of up to 256 3-byte entries. In each entry, the first two bytes contain the dictionary reference of the label, and the third byte is used to hold the identifying number of the flow unit.

Extraction of Forward-connector Information

The routine INSFCT accesses the XTRFLO field of XCOMM to find the text reference of the first flow unit header table (FUH) in the first block in the text stream. It then follows the chain of text references in the IFOUNC field of each FUH, until all FUHs in the block have been seen. Other routines are then called to complete the processing for the flow units in the block before those in the next block are scanned.

The main function of the INSFCT routine is to extract information from each FUH, to use some of it to complete some of the fields in the FUIT entry for the flow unit, and to use some of it to make provisional entries in the FCT.

As each FUH is examined, the following fields in the FUIT are completed as required:

FUITR - text reference of the flow unit.

FUIDT - a flag which is set if control can drop through to the next physically sequential flow unit (Bit 0 of IFOFL in FUH).

FUICH - This field is used initially to construct a chain linking all flow units beginning with block-entry statements (PROC, BEGIN, ON-BEGIN, and ENTRY) or which have a label which can be branched-to from another block (indicated by IFOFL2 in FUH).

DOPLOPN bit in FUIFL - a flag that is set if the flow unit heads a do-loop (Bit 6 of IFOFL in FUH).

The 32-byte bit strings IFOVS and IFOBEN in the FUH, which indicate usage of variables in the flow unit, are copied into a save area. A 32-byte field corresponding to the IFCBEX field of the FUH is set to zero at this stage.

A marker, X'FF', is inserted in the first byte of the next available field of the FCT to indicate that it is the first entry for a flow unit. An identifying number, 1-255, is allocated to the flow unit and inserted in the second byte of this entry. In order to insert provisional branching information in other FCT entries, the optimization entry for the block in the general dictionary is accessed. An entry is made in the FCT for each label constant that can be branched-to from the flow unit. If a label variable is branched to, the alias information extracted by Phase OA is accessed, and an entry is made for each possible value of the label variable. If a label has a higher containing-block-value than the current block, a special out-of-block-entry is made. If a flow unit ends with an END statement, no entry is made. At this stage, each entry contains the dictionary reference of the label. Because each dictionary reference requires to be converted to a flow-unit number, a list of labels is kept in LFUN and appropriate flow-unit numbers are applied to them.

At the end of the scan, entries are made in the tables for a dummy flow unit with an identifying number of zero. This dummy flow unit acts as the backward connector for the first flow unit in the block, and for flow units that end in a RETURN statement, a branch out of a block (GCOB), or a CALL statement which passes control to the containing block. The entries set up for this flow unit are later used to provide backward connector information and to collect variables-busy-on-exit information.

When the initial scan for a block is complete, control passes to the GENFCT routine. This routine uses the information collected in the LFUN table and in the provisional FCT entries, to modify the FCT entries so that each entry indicates a flow unit which is a forward connector of the flow unit associated with the entry. The FUIFC field of the FUIT is

set to point at the first of its list of associated FCT entries. If control can drop through to the next sequential flow unit, the X'FF' entry is replaced by the appropriate forward connector entry. If a flow unit ends at a RETURN statement or a CALL which can pass control to a label outside the block, bit 3 of the flag byte in the FCT is set, and the identifying number (zero) of the dummy flow unit is set in the second byte.

Collection of Backward Connector Information

During the processing of FCT entries by the GENFCT routine, information about the number of BCT entries is collected. Each flow unit that has forward connectors is a backward connector for those flow units. Therefore, each time a flow unit identifying number is seen in the FCT entries, a count temporarily maintained in the FUIBC field of its FUIT entry is incremented.

GENFCT passes control to the FCTOBC routine to generate the BCT entries, which indicate the backward connectors of each flow unit. The counts maintained in the FUIBC slots are used to allocate the required number of BCT entries to each flow unit, and each FUIBC slot is then changed so that it points to the first BCT entry for that flow unit. A sequential scan is made of the FCT, and information in the entries is modified and used as required to make the BCT entries. If a flow unit has no backward connectors, and does not contain a label mentioned in an ON-unit, the code in the flow unit cannot be reached and therefore cannot be executed. The flow unit is flagged to indicate this. Exceptions to this are flow units that are FORMAT statements, which are not flagged as being unreachable.

Calculation of Level Numbers

When the information about forward connectors and backward connectors has been collected, control passes to the CALCLN routine. This routine calculates the smallest number of flow units through which control must pass to get from an entry unit to a particular flow unit. It then adds one to that number to arrive at the level number for the flow unit. Thus each flow unit has a level number one greater than that of its backward connector.

The chain of entry units is scanned. The forward connectors of each entry unit are allocated a level number and added to the end of the chain. When all entry units have been processed, the scan of the chain continues, examining and allocating level numbers to the forward connectors of items added to the chain. In this way, all flow units that have backward connectors are chained in ascending order of level number.

Determination of Back Dominators

During the assignment of level numbers, a check is made of those flow units that have only one backward connector. For control to reach such a flow unit, it must pass through the backward connector, and therefore the backward connector is also the back dominator of that flow unit. The identifying number of a flow unit's back dominator is inserted in the FUIBD field of its FUIT entry. If a flow unit has more than one backward connector, it is chained to other similar flow units; the FUILC field of its FUIT entry is temporarily used for this purpose.

Control then passes to the routine CALCBD, which finds the back dominator of each flow unit for which one has not already been found. The FUIT entries of each flow unit are accessed in ascending order of level number by scanning the chain in the FUILC fields. Each flow unit has a backward connector with a lower level number and, because the flow units are processed in ascending order of level number, the back dominator of that backward connector will have already been determined. A list is made of all the flow units on the path between the backward connector and its back dominator. Then each backward path from the current flow unit is traced until a flow unit which is in the list is met; this flow unit is a possible back dominator. (During the trace of back paths, any flow unit that is met which is not on the list is added to the list, so that a check can be made if a looping path is being traced.) The tracing of a back path is stopped when a flow unit that has a lower level number than the current flow unit is found; the chain of back dominators is followed from that point. When all back paths have been traced, and all possible back dominators have been found, the possible back dominator that has the lowest level number is the actual back dominator.

Identification of Loops and Back Targets

The FLOOEN routine examines the information collected for each flow unit to determine those that are loop entry units. A loop entry unit is a flow unit which either has itself as a backward connector, or has one or more backward connectors for which it is the back dominator. Flow units for which the DOLOPN bit in the FUIFL field of the FUIT entry was set by the INSFCT routine, indicating that the flow unit heads an iterative do-loop, are loop entry units.

When a loop entry unit is found, information about other flow units logically connected with it is examined to determine the nearest flow unit through which control must flow before the loop is entered. That flow unit is referred to as the back target of the loop, and of each flow unit in the loop. Optimization may involve the moving of text from within a loop to its back target, in order to avoid unnecessary repeated executions during execution of the loop. A back target is identified in any of the following ways:

1. If the immediate back dominator of a loop entry unit has only one forward connector, that back dominator is also the loop back target.

2. If control drops through to the entry unit of a do-loop from its back dominator, the back dominator is the do-loop back target.

3. If the foregoing conditions are not satisfied, the loop entry unit is flagged as having a distant back target, i.e., a back target which is not its back dominator. A backwards scan is made of the back dominator chain of the loop entry unit, until a flow unit is found which is a loop entry unit with a non-distant back target. If a backward path exists from the current loop entry unit to this found unit, and does not pass through the back target of the found unit, the back target of the found unit is also the back target of the current loop entry unit.

4. If no flow unit which satisfies the foregoing conditions is found, the dummy flow unit (zero) is nominated as the loop back target.

The back targets of loop entry units are identified in the FUIBT field of their FUIT entries. When all loop entry units and their back targets have been identified, the routine FLOODA makes a further scan of the chain of FUIT entries.

During this scan, the loop in which each flow unit is contained is identified. As each flow unit that is not a loop entry unit is seen, its back dominator chain is examined until a back dominator that is also a loop entry unit is found. If this loop entry unit lies on a forward path from the current flow unit, and the forward path does not pass through the back target of the loop entry unit, then the current flow unit and the loop entry unit have the same back target and belong to the same loop. The current flow unit is linked into the loop chain after its loop entry unit. A stack is used to identify the last flow unit in a loop, and the identifying number of the end-of-loop flow unit is inserted in the FUILEP field of each FUIT entry in the loop.

Back targets are stacked so that any nesting of loops can be recognized. The method of stacking enables each back target to be given a number which indicates its depth of nesting. A depth number one greater than that of its back target is applied to each loop, and the appropriate number is inserted in the FUILEP field of each FUIT entry. Flow units that cannot be identified as belonging to a loop are allocated to a false loop with a depth number of one.

The FUILC field of each FUIT entry is used to construct a chain of flow units according to their grouping within loops. Loops are chained in descending order of depth number, i.e., from the innermost to the outermost loop. Within each loop, flow units are chained in ascending order of level number. The direction of the existing chain of flow units, via the FUICH field of the FUIT entries, is reversed so that it links flow units in descending order of level numbers.

Extraction of "Busy-on-exit" Information

The BUSYEX routine scans the flow unit chain (FUICH), and accesses the IFOVS and IFOBEN bit strings copied from the flow unit headers, to extract information about variables that are busy-on-exit from any flow unit. A variable is busy-on-exit from a flow unit if it is used (i.e., appears as operand 1 or 2 in a text table) before it is set (i.e., appears as operand 3 in a text table) on any forward path from the flow unit. In addition, a variable is busy-on-exit from a block-exit flow unit if it is declared in the containing block, or if it is explicitly declared with either the CONTROLLED attribute or the STATIC attribute in the current block. If a variable is busy on exit from a flow-unit, and is not set in that flow unit, it is also busy-on-entry.

The flow unit chain is scanned so that flow unit tables are seen in descending order of level numbers; this ensures that each flow unit is processed before its back dominator. For each flow unit, the paths of backward connectors are traced as far as its back dominator. As each flow unit on the backward path is seen, its IFOBEX string (set to zero by INSFCT) is modified by performing a logical OR operation between it and the IFOBEN string of the flow unit currently being processed.

The following operations are then performed on the bit strings of each backward connector:

1. A bit string is generated which contains those bits set in its IFOBEX string and not set in its IFOVS string.

2. A logical OR operation is performed between the bit string thus generated and its IFOBEN string.

3. The string thus produced is compared with its IFOBEN string. If the two strings are identical, no further action or tracing of this backward path is performed. If the two strings differ, the IFOBEN string is replaced with the modified string, and the trace of the backward path continues (unless the backward connector is the back dominator of the flow unit currently being processed).

The process is repeated for each flow unit in the flow unit chain, until all the busy-on-entry and busy-on-exit information has been collected or updated.

Insertion of Flow Analysis Information in the Dictionary and Text

The routine LOODAT scans the loop chain in the FUILC field of the FUITs. As each loop entry unit is found, the flow unit header in the text stream is accessed, and a skeleton loop-data entry is created in the general dictionary. The format of a loop data entry is shown in figure 2.25.

Bytes	Symbol	Meaning
0	ILCDE	Code byte (X'12')
1	ILBAT	Back target
2	ILFOT	Forward target
3	ILDEN	Depth number
4	ILOON	Not used
5	ILFLGS	Flags copied from FUIFL field of FUIT for the loop entry unit
6-10	ILBATR	Text reference of back target
11-42	ILSET	Variables set in the loop
43-74	ILUSE	Variable used in the loop
75-106	ILBEX	Variables busy-on-exit from the loop
107-138	ILSET2	Variables set twice in the loop

Figure 2.25. Format of a loop-data entry created in the general dictionary by Phase OI

The ILCDE, ILBAT, ILDEN, and ILOON fields of the loop data entry are completed by copying the FUIT entries. The flow unit header for the loop entry is altered to the format shown in figure 5.99. The FUIT and bit strings of each member of the loop are then accessed in turn, and processing performed to complete the dictionary and FUH entries.

The forward connections of each member of the loop are examined; if this reveals only one forward connector for the loop, this is the forward target to be inserted in the loop data entry. The back target chain is followed until a flow unit with a depth number of zero is found; this is the non-looping back target to be inserted in each FUH. The variables-usage bit strings modified during the phase operation are used to complete the loop-data entry.

If the REORDER option has been specified for a block, a flag bit in the phase is set, and this indicates the setting required for bit 7 of the IFOF2 byte in the FUH.

If a block contains more than 256 flow units, the flow analysis described in preceding paragraphs cannot be performed. This fact is detected by the INSFCT routine, which then passes control to the GUBLOK routine. In such a case, all flow units in the block are treated as if they belong to a non-looping loop (depth-number zero), and bit 7 of the IFOF1 byte of each FUH is not set.

When the flow analysis of a block is complete, and all relevant text and dictionary entries have been made, control returns to the INSFCT routine for analysis of the next block if one exists.

TEXT OPTIMIZATION (PHASE OM)

This phase modifies the text stream, by changing the order of text tables, by modifying or replacing text tables, or by a combination of both methods, in such a way that the object code defined or indicated by the text can be executed more efficiently. The information extracted and collected by Phases OA, OE, and OI is used as an aid to this optimization process.

The principal features of this optimization process are:

- Common Expression Elimination: the elimination of expressions that give a result which has been calculated elsewhere in the same loop. This process is performed for all flow units.
- Backward Movement of Expressions: the removal from a loop to its back target of those expressions which do not need to be recalculated for each execution of the loop. This process is performed for all loops, but may be restricted if the REORDER option is not specified for the block in which a loop is contained.
- Strength Reduction: reduction of the complexity of operations performed within a loop. This can involve the replacement of a multiplication operation by a series of addition operations. It can also involve some backward movement e.g., if a subscript variable within a loop is incremented, the subscript calculation is moved backwards outside the loop, and the variable inside the loop is incremented by addition. A similar form of strength reduction may be performed where conversion operations are involved. In addition, strength reduction is performed in cases where the affected variable is contained in an inert text table. An inert text table is a text table which indicates an incrementation of a variable that is not set anywhere else in the loop.

PHASE INPUT

Input to the phase consists of the main text stream in Type-2 text format. Each flow unit within the text is headed by a flow unit header table, modified and completed by Phase OI (see figure 5.99) and each flow unit contains one or more partially completed hash-chain tables, inserted by Phase OE (see figure 5.100). Entries for locators in the variables dictionary are accessed. Loop data entries, block optimization entries, and the program optimization entry in the general dictionary are accessed.

PHASE OUTPUT

Output from the phase consists of the main text stream, modified as described under the heading "Phase Operation." No dictionary entries are made by this phase.

PHASE OPERATION

The text is scanned by following the loop chain, set up by Phase OI, via the IFOLC field of each flow unit header. The organization of this chain ensures that the innermost of any nested loops are processed first. All the processing required for a loop is performed before the next loop is examined. Within each loop, flow units are scanned in

ascending order of level number, and text tables within each unit are scanned by following the pointers in the IFCHN fields.

As each text table is seen, its IOP1 code byte is examined to see if it is a text table likely to be involved in the processing performed by this phase. Such text tables include: MULT, SUBS, SUBS1, CONV, PLUS, MINUS, ASSN, DIVIDE, SHIFT, AND, OR, NOT, CONCAT, BIF, EQ, GE, LE, NE, GT, LT, and MOVE text tables. All these text tables are initially considered as candidates for common expression elimination, and are processed by routines beginning at CEE100. Text tables that are assessed as movable are also considered as candidates for backward movement out of the loop, and are further processed by the routine BAKMOV. In addition, MULT, SUBS, and SUBS1 text tables, and CONV text tables which have arithmetic targets are linked in a special chain, which is later scanned by the routine STREDU for strength-reduction processing.

Common Expression Elimination

The routines starting at CEE100 attempt to eliminate repeated evaluation within a loop of expressions which represent a value previously calculated. For example, if a loop (as determined by Phase OI) contains the following source statements:

```
X=A*B;
.
.
.
Y=A*B;
```

then, provided that the values of A and B have not been changed (set) in intervening statements, the text can be changed so that it represents the following source statements:

```
X=A*B;
.
.
.
Y=X;
```

To enable a check to be made on the usage of variables (and their aliases), the dictionary entries created by preceding phases in this stage are accessed. A number of fields are set up and maintained in the phase working area to enable relevant information to be accessed. Two of the fields maintained are:

VSSFP - variables set so far in current page.
VSSF - variables set so far in current flow unit.

Each of these fields is 32 bytes long, and consists of a bit vector indicating relevant entries in the variables dictionary.

This information is used to determine whether global temporary operands (i.e., temporary operands that can be used in more than one statement) need to be generated to enable common expression elimination. For example, if a loop contains the following source statements:

```
X=A*B;
.
.
X=X+2;
Y=A*B;
```

the variable X is set between the statements containing the common expression A*B. In such cases, a global temporary operand is generated

and the common expression assigned to it, so that the text represents the following source statements:

```
G.temp.1=A*B;
X=G.temp.1;
.
.
X=X+2;
Y=G.temp.1;
```

Global temporary operands are also used to avoid repeated evaluation of common subscripts. For example, on input to Phase OM, text representing the source statement:

```
A(J)=B(J);
```

contains the following text tables:

```
SUBS1          J          4          t1
NDX            t1         A      Q-temp.1
SUBS1          J          4          t2
NDX            t2         B      Q-temp.2
ASSN       Q-temp.2       -      Q-temp.1
```

The result of the first evaluation of the common subscript is assigned to a global temporary operand, so that on output from Phase OM the text is modified as follows:

```
SUBS1          J          4          t1
ASSN           t1         -      G-temp.1
NDX        G-temp.1       A      Q-temp.1
NDX        G-temp.1       B      Q-temp.2
ASSN       Q-temp.2       -      Q-temp.1
```

Subscripts are also commoned for multidimensional arrays. For example in the PL/I statement:

```
A(J,K,L)=B(J,K,L);
```

the calculations of the subscripts would be commoned in the same way as that described above. A further attempt is made to common partially common subscript calculation. In the statement:

```
A(J,K,L)=B(J,K,M);
```

the calculation of the first two subscript items will be commoned. The commoning of parts of subscript calculations is complicated by the fact that the compiler uses two algorithms for subscript calculation. The more common algorithm works from left to right and relies on the value of previously calculated results. The algorithm is:

```
Element offset=(...(S1*M1+S2)*M2...+SI)*MN

Where SI= the Ith subscript,
       M= subscript multiplier and MI the Ith subscript multiplier,
   and N= the number of dimensions in the array.
```

This is known as the optimizer algorithm, as it was adopted for the optimizing compiler.

The other algorithm calculates the subscript for each dimension separately and then adds the results. The algorithm is:

```
Element offset=(S1*M1+S2*M2....SN*MN)

Where SI= the Ith subscript,
       M= subscript multiplier and MI the Ith subscript multiplier,
   and N= the number of dimensions in the array.
```

If the current text table is a MULT table, a CONV table, or a SUBS1 or SUBS table associated with dependent subscript multipliers, it is a candidate for strength-reduction processing. Regardless of the result of hashing one of these text tables, it is linked into the ninth hash chain. SUBS1 and SUBS text tables associated with independent subscript multipliers are candidates for strength-reduction processing after they have been moved into a back target, and these text tables are linked into the tenth hash chain. When common-expression-elimination and backwards-movement processing for the loop is complete, these chains are scanned by the STREDU routine.

Before an attempt is made to common expressions, tests are made to determine whether a text table is movable or not. A text table is considered to be movable if the following conditions are satisfied:

- Its result (third) operand is a variable.
- The result variable is not set elsewhere in the loop.
- The result variable is not busy-on-exit from the back target of the loop.
- The result is in a flow unit always executed in the loop. Thus, an ASSN text table is not movable unless the REORDER option is specified for the current block.

If a text table is movable, it can be moved to the loop back target or commoned with a matching text table in the loop target. Control is passed to the BAKMOV routine for processing of such a text table. If a text table satisfies all the conditions for being movable except that it is not in a flow unit always executed in the loop, the search for a matching text table with which it can be commoned is made through back dominators up to and including the loop back target. If a text table is not movable, the search for a matching text table with which it can be commoned is made on the backward paths as far as the loop entry unit.

The backwards search for a matching text table is made using the information and chains previously extracted or set up. If a match is found, tests are made to see if either of the first two operands is set between the matching text table and the current text table. If an operand is set, no expression elimination can be performed. If no operands are set between the two text tables, control passes to routine CEI700, which examines the result operands in the two text tables and determines the action to be taken to eliminate the common expression. When the required action is determined, control passes to routine CEE900, which modifies, deletes, or generates text tables as required. The text modifications can be summarized as follows:

- If the result operand of the matching text table is not set between the two text tables, the current text table is changed to an assignment from the result operand of the matching text table.
- If the result operand of the matching text table is set between the two text tables, an ASSN text table assigning the result operand to a global temporary operand, is generated and inserted after the matching text table. The current text table is changed to an assignment from the global temporary operand.
- If the result operand of the current text table is a temporary operand, the current text table may be deleted, and subsequent uses of the temporary operand replaced by the global temporary operand.

Backward-movement Processing

The routine BAKMOV checks for situations where repeated execution within a loop of instructions defined in a text table will always produce the same result, and where movement of the text table to the loop back target will not affect the final result of execution. The conditions

This algorithm is known as the independent algorithm or F algorithm, it was previously used in the PL/I (F) compiler. The optimizer algorithm has the advantage that it requires less space for calculation. However, when it is used, subscript calculation can only be commoned to the point where the first difference in the subscript list occurs. With the F algorithm, commoning can take place regardless of previous entries in the subscript list.

The optimizer algorithm is used provided that all extents of the array are known and the array is not an array within a structure of more than one dimension. (Note: the number of dimensions includes any that may be inherited from a structure.) Take the statement:

```
A(J,K,I,L,M,N)=B(J,K,I2,L,M,N);
```

With the optimizing algorithm, commoning could only take place on the calculation of the first two elements in the list. With the F algorithm, commoning could also take place for the calculation of the last three elements in the list.

When the F algorithm is used, commoning only occurs when the subscript calculation is moved out of a loop. Subscript calculation is moved out of a loop if the subscript value is a loop constant. (A loop constant is a value that cannot be altered during the execution of a loop). Take the loop:

```
DO I=1 TO N;
   B(I)=B(N);
   END;
```

The calculation of the address of B(N) will be done once outside the loop and placed in a temporary. The temporary will then be used for the assignment.

As each text table is seen, its operands are examined to determine whether it can be optimized. If a text table contains an operand that is used or set in an on-unit, or in a block that can be called from an on-unit, the operand is classed as abnormal and no optimization can be performed on the text table.

If a text table is classified as optimizable, special processing is performed to aid any search along its backward paths for a text table which contains an expression common to the current text table. The subroutine HASHER is called to perform this special processing, which is referred to as hashing. The values of the operator code byte and the first operand (also the second operand if applicable) are manipulated in such a way as to produce an arithmetic result, which is known as the hash result. Subroutine HASHER produces two hash results for each text table it processes. The first has result is a value between zero and 2047, and is referred to as the larger hash result. It is used to index a 256-byte bit string, HASTRI. Bits are set in HASTRI to indicate that a text table producing a similar hash result has been processed earlier, and it is therefore worthwhile making a search along the backward paths to find such a table for the purposes of commoning expressions. If the bit in HASTRI corresponding to a larger hash result is not set, a backward search is not performed, but the bit is set for future reference. The second hash result produced by HASHER is a value between zero and seven. This value indicates one of the eight chain fields in the IHASHC field of the hash-chain table for the flow unit or page. Each field contains the reference of the first text table in a chain of text tables with similar hash results. If a search is to be made for the purpose of commoning text tables, following the chain indicated by the smaller hash result speeds the search. If a search is not to be made, the current text table is added to the chain.

affecting the movability of a text table have already been tested before control is passed to the BAKMOV routine. This routine determines whether a text table can be removed in its entirety, or whether the result operand of a moved text table is to be changed to a global temporary operand which can be used in the original position inside the loop. Backward movement is accompanied by common expression elimination. When a text table is moved into the back target, it is also linked into its appropriate hash chain. Any subsequent candidate for backward movement can then be checked for commoning with a previously-moved text table. The subroutine COMBAT is called to search the back target for a matching text table. If no match is found, the text table is moved as required.

When all requirements for backward movement have been determined, control passes to routine CEE900, which makes the required text changes. Some of the text changes that may be made are illustrated in the following examples.

If the source program contains the statements:

```
DO J = 1 TO 10;
   .
   .
   .
   R = X + Y;
   .
   .
   .
END;
```

the text input to Phase OM will contain the following text table within the do-loop:

```
PLUS    X    Y    R
```

If neither X nor Y are set within the loop, then the expression "X + Y" is considered invariant. The backward-movement processing is as follows:

1. If R is a variable then:

   a. If the REORDER option applies, the entire PLUS text table is moved from within the loop to the loop back target.

   b. If the ORDER applies, or if R is set more than once in the loop, a new text table, "PLUS X Y G-temp.1", is generated in the loop back target, and the original text table within the loop is replaced by "ASSN G-temp.1 - R".

2. If R is a local temporary operand, then:

   a. If either the ORDER or REORDER option applies, a text table, "PLUS X Y G-temp.1" is generated in the loop back target, the original text table within the loop is replaced with a NULL text table, and all other references to R within the loop are replaced by the global temporary operand.

   b. The only form of local temporary operands that can validly be set more than once within a loop are string-address temporary operands, which can effectively be set in string-handling operations such as concatenation, etc. If R is a string-address temporary operand, no backward-movement processing is performed.

Strength Reduction Processing

When all the text tables in a loop have been scanned and processed for common expression elimination and backward movement, control passes to the routine STREDU. This routine scans text tables linked by the special hash chain, containing only MULT, SUBS1, and SUBS text tables, and CONV text tables that have arithmetic target operands. The direction of the hash chain is reversed so that text tables are scanned and processed in the order in which they appear in the phase input.

In addition to simplification of operations performed, strength reduction may also involve some common expression elimination and backward movement of text. The process of strength reduction involves analysis of the use of a control variable within a loop, and movement, modification, or replacement of MULT, SUBS1, SUBS, and CONV text tables. This form of optimization is only attempted for a DO-loop with a single specification which contains an inert text table, i.e., a text table in which a variable that is not set elsewhere in the loop is incremented (e.g., control variable = control variable + constant (loop or absolute)). (A loop constant is a variable that is not set inside the loop.)

Strength Reduction of MULT Text Tables: A situation in which a MULT text table can be optimized is represented by the following source statements:

```
DO I = L TO M BY N;
   .
   .
   .
   R = I * K;
   .
   .
   .
END;
```

The action taken is as follows:

1. During the text scan, the text table, "MULT I K R" , is tested to see whether either operand 1 or operand 2 is the loop-control variable and the other is a loop constant. If this is so, the MULT text table is linked into the strength-reduction hash chain.

2. When the MULT text table is seen by routine STREDU during its scan of the strength-reduction hash chain, global temporary operands (G-temp.1 and G-temp.2) are generated, and text tables representing the statements "=G-temp.1 = L * K;" and "G-temp.2 = N * K;" are inserted in the loop back target. (Depending upon the values of K, L, and N, these text tables may themselves be involved in strength-reduction processing.) A text table representing the statement "=G-temp.1 = G-temp.1 + G-temp.2;" is generated at the end of the loop, immediately preceding the TO text table for the loop.

3. The MULT text table within the loop is either replaced by a NULL text table (and all other references to R are replaced by G-temp.1) or, if R is a variable that is busy-on-exit from the loop, it is replaced by a text table assigning R to G-temp.1.

4. If the loop-control variable is not busy-on-exit from the loop, and all references to it within the loop occur in strength-reducible text tables, the loop-control variable is assigned to a global temporary operand (G-temp.3) within the loop back target. The loop control is thus changed so that the loop can be effectively represented by the statements:

```
      G-temp.1 = L * K;
      G-temp.2 = N * K;
      G-temp.3 = M * K;
   DO G-temp.1 = G-temp.1 TO G-temp.3 BY G-temp.2;
      R = G-temp.1
   END;
```

Alternatively, if the control variable is busy-on-exit from the loop, and the loop increment is ±1, and there is a unique exit from the loop, then the loop control mechanism can be changed, provided the following is inserted:

	PLUS	final value	control variable
or	ASSN	final value	control variable

Strength Reduction of CONV Text Table: If a CONV text table involves usage of the loop control variable under conditions similar to those described for a MULT text table, the CONV text table is removed from the loop. This is done by maintaining the conversion control variable in a temporary operand, and incrementing the temporary operand.

STORAGE AND REGISTER ALLOCATION STAGE

This stage consists of seven phases. The first five phases determine the static or automatic storage requirements of all the variables and constants in the text, and define the code required to make these items addressable. Dynamic storage requirements are determined and mapped as far as is possible at compilation time. Phase QA allocates specific registers for use during execution of the code in the object module. The last phase in the stage, Phase QE, is an optional phase. It is only loaded and executed if global optimization is specified; its function is to delete text indicating the storage of register contents in situations where elimination of these instructions enables the object program to be executed more efficiently.

SYMBOL TABLE RESOLUTION (PHASE PC)

This phase partially builds the Pseudo Constants Pool (PCP), which is used by the final assembly phase (Phase SI) to build the constants pool in static storage. Entries are made in the pseudo constants pool by this phase for object-time DEDs and FEDs, symbol tables, and symbol table element lists. Further entries are made by later phases.

PHASE INPUT

Input to the phase consists of the main text stream output from Phase KX. The text consists of Type-2 text tables chained in logical sequence. The physical sequence of the text tables varies according to whether or not phases in the global optimization stage have been executed.

The general, variables, and names dictionaries are accessed. In certain circumstances, the general and variables dictionaries are also scanned, either sequentially or by following chains of entries flagged as requiring symbol tables.

PHASE OUTPUT

Output from the phase consists of a partly-built pseudo constants pool, a scratch page containing information about the pseudo constants pool for use by later phases, and a modified main text stream.

The pseudo constants pool is built on a page (or pages) that is handled separately from the main text stream. Entries in the PCP are of three kinds: object-time DEDs and FEDs, symbol table elements, and symbol tables. The entries in the pseudo constants pool contain the entries that are used in the constants pool in the object-module, plus information which enables the entry to be relocated correctly in the constants pool.

The TA of the scratch page is inserted in the XSCRCH field of XCOMM. The scratch page contains the TA of the page containing the pseudo constants pool, and information indicating the next available space on that page is inserted in the XOUTARG2 field of XCOMM.

In order to make the order of the constants pool as efficient as possible, the PCP output from Phase PC may be in two halves; the first half comprising the DEDs and FEDs, the second half (commencing on a new text page) the symbol table and symbol table elements. The status of the two text streams is passed to Phase PA on the scratch page, so that it can insert entries between them. This ensures that the most commonly used entries in the constants pool are in the most readily addressable part.

Text tables containing items for which object-time DEDs and FEDs have been built have the compile-time DED or FED replaced by the offset in the constants pool of the object-time DED or FED. Variables dictionary entries for items for which symbol tables have been built, have the offset of the symbol table inserted.

The formats of various items for which entries are made in the pseudo constants pool are shown in figures 5.101 to 5.123.

Processing Requirements

The processing required by this phase varies according to the appearance of certain features in the PL/I source program, or in the text output from previous processing phases. The features that require processing by this phase can be classified as follows:

Class 1: PUT DATA, GET DATA, and SIGNAL CHECK statements without data lists.

Each data-directed input/output statement without a data list (or argument list at this stage of compilation) requires a symbol table for each variable declared (explicitly or implicitly) in blocks within the scope of the block in which the source statement appeared. A similar requirement exists for SIGNAL CHECK statements, except that symbol tables are only required for those variables for which the CHECK condition is currently enabled. In all the above cases, a symbol table element list must also be created for each program block involved. Each list consists of a contiguous series of addresses, containing an address for each symbol table associated with the block, and the address of the symbol table element list for the containing block. An object-time DED is required for each item for which a symbol table is required.

Class 2: PUT DATA and GET DATA statements with data lists, and the raising of the CHECK condition for a particular variable.

In these cases, symbol tables are required only for those variables specified in the data list, or for each variable for which the CHECK condition is raised. An object-time DED is required for each item for which a symbol table is required.

Class 3: Text tables indicating that object-time DEDs or FEDs are required to be passed as arguments to a library routine called at execution time. Examples of this requirement are calls to library routines for conversion operations or for input/output operations.

In these cases, object-time DEDs are generated by this phase, using the existing compile-time DEDs as a basis for their construction. Object-time FEDs will have been created by Phase KQ (compile-time FEDs are not created), and will be in text if they are four bytes long, or in the general dictionary if they are more than four bytes long. Picture-item FEDs are created by this phase.

Note: The class numbers shown above are used in the descriptions of processing in the following paragraphs, to indicate the items that are processed and to avoid lengthy repetitions of item descriptions.

In addition to these primary functions, the phase performs several miscellaneous functions, chiefly to provide more efficient input for later phases, particularly Phase PA. The more important of these functions are:

1. Determination of the target attribute for constant conversions.
2. Preliminary mapping of static initial storage.
3. Testing for requirement of static ONCB.
4. Reordering of argument lists.
5. Extinction of string address temporaries.
6. Determination of locator type required in less common cases (UNSPEC, area temporaries, etc.).

Sequence of Processing

Most of the processing is performed during a sequential scan of the text stream, an optional scan of the variables dictionary (made only if Class 1 items are found in the text), and a further optional scan of the variables and general dictionaries (made only if Class 1 or Class 2 items are found in the text). Only the text scan is required for processing Class 3 items.

The text stream is scanned sequentially, using the XNEXT macro routine, and items requiring processing by this phase are detected. Where items described in Class 1 are recognized, the block header entries in the general dictionary are accessed to obtain block-nesting information. This information is used to set flags in a phase save area, to indicate those blocks for which a symbol table element list must be built. Where items described in Class 2 are recognized, a flag is set in the variables dictionary entry for each variable for which a symbol table is required. If the variable is a structure, then the text reference to the structure is replaced by successive references to each of its base elements. If a symbol table is required for a label constant or an entry-point constant, a flag is set in the general dictionary entry for the constant, and the flagged entries are chained together for ease of access during later processing.

As each Class 2 or Class 3 item is seen in the text, a number of subroutines are called in sequence to build a DED or FED entry in the pseudo constants pool. For a Class 3 item, the sequence is as follows:

SRTD00 - obtains information about the required DED or FED from the text.

SRBD00 - sets up the PCP environment for the converted DED or FED.

SROB00 - converts the 3-byte compile-time DED or FED into its object-time form.

SROD00 - creates the required entry in the PCP, checking for duplicates if required.

For a Class 2 item, only the last three of the subroutines (SRBD00, SROB00, and SROD00) are used because information about the required DED is obtained from the variables dictionary. For non-pictured FEDs (which are constructed by Phase KQ) only SROD00 is used to make the entries in the PCP. Picture FEDs cannot be built by Phase KQ because information which is inserted in the general dictionary by Phase KX is required: these items are processed in a manner similar to that used for pictured DEDs, using the subroutines SROB00 and SROD00.

All DED and FED entries occupy contiguous storage at the beginning of the pseudo constants pool. As each new entry is made, an offset counter in the OFFSDED field is updated. Allowance is made for the difference between pseudo constants pool entries and constants pool entries so that the value in OFFSDED indicates the offset of the next DED or FED entry in the constants pool. This value is passed to the SROD00 subroutine each time it is called, and the offset for the new entry is returned and replaces the compile-time DED in text for Class 3 items, or is temporarily placed in the symbol table slot in the variables or general dictionary entry for Class 2 items.

Before a DED or FED entry is made, a test is made for an existing entry of identical form and value. Entries are commoned wherever possible, and the existing entry offset is returned in such cases.

When the text scan is completed, a scan is made of the variables dictionary if Class 1 items have been found. During this scan, DED entries are made in the pseudo constants pool for internal variables with dictionary entries flagged to indicate that they required symbol

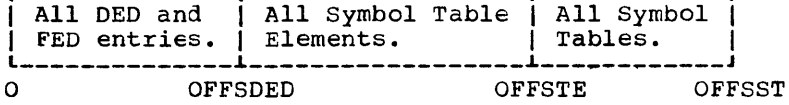
tables because a symbol table element list is required for the block in which they are declared. The number of symbol table elements required for each block is stored in the phase save area in which blocks requiring symbol table element lists were flagged. This number is used to construct the required number of dummy symbol table elements in the pseudo constants pool.

If Class 2 items have been found, a second scan of the variables dictionary is made, during which the SRSE00 subroutine is called to insert the required values in the dummy symbol table elements. This routine calls the SRST00 subroutine to build the symbol tables in the pseudo constants pool. Entries in the variables and names dictionaries are accessed for the required information. As the symbol tables are built, the required addresses are inserted in the symbol table elements, and the offset of the symbol table is inserted in the variables dictionary entry.

If symbol tables are required for any external variables, the DEDs for them are built during this scan, to ensure that the DEDs and symbol tables for a given external variable (which may be a structure) are in the same external control section. If necessary, the symbol table element is completed at this time.

If entries for constants in the general dictionary have been flagged to indicate that a symbol table is required, the chain of flagged entries is scanned and the symbol tables are constructed.

On completion of processing, the pseudo constants pool contains contiguous entries organised as shown below:



Note that the symbol table elements start a new page in the PCP so that Phase PA can insert more commonly required items nearer the start of the constants pool.

The TA of the current pseudo constants pool page, and the relocation factors of the various classes of entry, are passed to Phase PA on a scratch page, to enable that phase to continue construction of the PCP.

CONSTANTS ANALYSIS (PHASE PA)

This phase identifies all items in the text that are constants, or can be considered as constants, and for each item creates an entry in the pseudo constants pool which indicates the storage to be allocated for the item in the constants pool, (which is a part of static storage). Items for which storage is allocated in the constants pool include:

- Source-program constants
- Compiler-generated constants
- Address constants
- Label constants
- Static ONCBs
- Static locators
- Static descriptors
- Descriptor descriptors
- File control blocks
- Record descriptors
- Key descriptors
- Environment blocks
- DTFs
- CONDITION condition names

Source-program constants and compiler-generated constants may require to be converted before being entered in the pseudo constants pool. The library conversion routines are link-edited with this phase. A PL/I environment is created in the phase to enable the library routines to be called, and the conversions executed, during compilation.

The phase also maps completely the storage required for static variables declared with the INITIAL attribute. During processing, this phase rebuilds the text stream so that on output, the physical sequence of the text coincides with the logical sequence.

PHASE INPUT

This phase scans the main text stream output from Phase PC. The text consists of Type-2 text tables, chained to enable the logical sequence to be followed. The variables dictionary is scanned, and the variables and general dictionaries are accessed during the text scan.

Other input consists of the partially-built pseudo constants pool output from Phase PC, which is accessed by use of the information in the scratch page also output from that phase. The scratch page contains the TA of the pseudo constants pool page. The TA of the scratch page is found by accessing the XSCRCH field in XCOMM.

PHASE OUTPUT

Output from the phase consists of a rebuilt text stream in which text tables, flow unit headers, etc., are organized in logical order. Some text tables, e.g., static initial (SINIT) tables, and argument tables referring to static arguments, are deleted because the information they contain is conveyed by other means.

The pseudo constants pool is output on a second stream of one or more text pages. Entries in the pseudo constants pool contain all the information required by the final assembly phase (Phase SI) for building the constants pool in static storage.

The phase also generates information required by later phases in the storage and register allocation stage, and adds this information to the scratch page output by Phase PC. The information includes the sizes of the areas of the pseudo constants pool that are built by this phase.

PHASE OPERATION

Sequence of Processing

A scan of the variables dictionary is first made identifying those variables which require some form of storage in the constants pool regardless of any references in the text. These include:

- anchor slots for controlled variables, with adcons if external.
- adcons for external symbol tables CSECTs.
- locators and descriptors or adcons for static external variables.
- some classes of defined variables.

A scan of the main text stream is then made using the XNEXT macro. During this scan, all forms of entry in the constants pool not already processed are identified and constant descriptors created accordingly.

The phase creates a new, sequential output text stream. It also deletes text tables which are no longer required; for example, NULL and GHOST tables, ARG tables referencing static arguments, static initial and related tables, and unwanted flow-unit header tables.

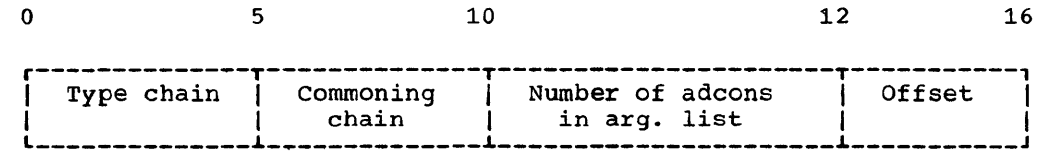
At the end of the phase, the scratch page of information about the PCP, as received from Phase PC, is updated and passed to Phase PE.

Creation of Pseudo Constants Pool Entries

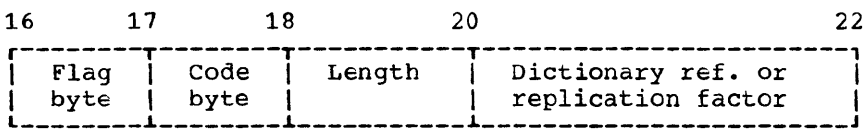
This phase continues the construction of the pseudo constants pool initiated by Phase PC. The PCP is processed by Phase SI to produce the execution-time constants pool.

The PCP entries are identified in a random order from the text and dictionary whilst in the constants pool they are required to be grouped according to type. Consequently, the entries are first created in an intermediate temporary form, existing solely within this phase, known as a constant descriptor. The constant descriptor comprises a 16-byte field on the front of the PCP entry.

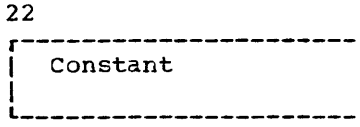
1. Constant descriptor header.



2. PCP header.



3. Constants pool entry.



When created, constant descriptors are divided into eleven classes according to type.

Doubleword aligned	(Long float, extended float, locators, locators/descriptors)
Label constants	
Static ONCBs	
Adcons	(Addressing adcons and argument lists)
Word aligned	(Short float, fullword binary, files, descriptors)
Halfword aligned	(halfword binary)
Byte aligned	(Character, bit, fixed decimal, pictures)
Doubleword word Halfword Byte	Aligned static initial variables

When created, constants within the same type are chained together with a back chain in the type field. In most cases, before the constant descriptor is actually output, a search is made within the constant descriptors of the same type for a duplicate of the present constant. If the duplicate is found, the present constant descriptor is not output; instead, the offset of the duplicate constant, as found in the constant descriptor, is returned. The search is expedited by the presence within the constant descriptor of a second chain, the commoning chain, which chains together those items within a type chain which can possibly be duplicate; thus the word-aligned chain is divided for commoning purposes into short float and fullword binary commoning chains. Items within a type which are excluded from commoning (for example, files and descriptors in the word-aligned chain) are excluded from the commoning chain. Certain whose chains are excluded from commoning attempts; these are the static ONCBs and static initial chains. The adcon chain is handled rather differently, in that its entries comprise groups of argument lists which must be kept intact. If global optimization is requested, an attempt is made to common complete argument lists in which case the commoning chain links together the last member of each argument list; the number-of-adcons slot is used to ensure that commoning is only attempted upon argument lists with an identical number of adcons.

When the scan of the text is complete, the back chain of each of the eleven classes is scanned and converted into a forward chain. From these forward chains the constant descriptors are scanned sequentially,

class by class, to produce the pseudo constants pool entries. The chains are scanned in such an order that the most efficient order of the constants pool ensues; for the same reasons, the classes produced by this phase are interleaved with three classes produced by Phase PC (DEDs/FEDs, symbol table elements, and symbol tables). The final order is:

1. DEDs and FEDs.
2. Halfword aligned constants.
3. Doubleword aligned constants.
4. Word aligned constants.
5. Adcons.
6. Byte aligned constants.
7. Label constants.
8. Static ONCBs.
9. Symbol table elements.
10. Symbol tables.
11-14. 8-, 4-, 2-, 1-byte aligned static initial variables.

Types of Constant Pool Entries

A brief description follows of each of the various pseudo constant pool enttries produced by this phase.

PL/I Source Constants: This phase allocates storage for all types of PL/I source constants used in the program, that is, character, bit, decimal, and binary constants. As far as possible, the phase attempts to convert the constant to the form in which it will be needed at execution time. To achieve this in the more difficult cases (for example, conversions from character to arithmetic and vice-versa, conversions to and from float, and conversions to picture) the phase has link-edited as an integral part of it some of the object-time library conversion package. To make use of this package, the phase has to simulate an execution-time PL/I environment and library call; this involves the setting-up of a DSA TCA, and library workspace as well as the calling sequence involving argument lists, DEDs, and, where appropriate, locators. It is to avoid this expensive overhead in compile-time that the simpler and commoner conversions are performed by inline code.

Files: Reference to a file constant in the text causes the phase to allocate storage for the various components of the file; the FCB, environment block, DTF, and if Phase KM has performed the optimization of the opening of the external file, the buffers. External files also require an adcon to be generated to be used for addressing purposes. File entries are not commoned in the normal way, via constant descriptors. Since each file has its own dictionary entry it is possible to flag the allocation and store the offset of the file (or adcon if external) in the dictionary for use by subsequent references.

If the file is an associated one, Phase PA will leave the offset of the DTF of that file in the field YFATT. For an EXTERNAL file this offset will be that from the EXTERNAL CSECT; for an INTERNAL file, the offset will be that from the start of the 4-byte aligned items in the INTERNAL STATIC CSECT.

Static ONCBs: Phase PC flags tables which require ONCBs. This phase merely takes the ONCB flags and identifiers from the text and stores them in the 8-byte ONCB. Static ONCBs are not checked for duplication.

Locators and Descriptors: The phase must allocate storage for structure and array locators and descriptors, string and area locator-descriptors, and record and key descriptors. Their requirement arises from three sources:

1. For addressing static external and certain classes of defined variables in which case the scan of the variables dictionary indicates that they are needed.

2. For execution-time mapping code produced by Phase IQ in which case operands are encountered in the text which make explicit reference to the locator or descriptor of the variable.

3. For library calls where the argument to be passed is an aggregate, string, or area in which case the variable is found in a LOAD ADDRESS, ARG, or CONVERT(01) text table.

Record/key descriptors are created by Phase KM and are simply copied from the dictionary when referenced in the text.

All the information required in some of these locators is known at the time. String locator-descriptors will have the string length, and, if appropriate, the varying flag and bit offset set, but will only have the address portion filled in if the string is constant (this address will also have been allocated by the phase). Area locator/descriptors contain the area length. Aggregate locators contain the address of the descriptors, since that is also allocated by this phase, but not the address of the data. The missing addresses are supplied by Phase SI when it creates the constants pool.

Labels: Labels (user or compiler) only require static storage in certain circumstances: in label variable assignments, if passed as a library argument, or if the target of a go-to-outer block. Each of these occurrences can be detected by the presence of the label in particular text tables.

CONDITION Conditions: The phase allocates 8 bytes of static storage for user conditions. Phase SI filles these eight bytes with the name of the condition. The requirement is detected from the text scan.

CONTROLLED Variable Anchor Slots: The implementation of controlled adressing requires a 4-byte slot for each controlled variable. The variables dictionary scan performs this allocation for each controlled variable in the program. In addition, if the variable is external, an adcon is generated to address the anchor slot.

Adcons: Adcons are required for two principal reasons:

1. For the addressing of static external CSECTSs. These CSECTs exist for static external variables, external files, anchor slots for external controlled variables, and symbol tables for external variables and external entry names. The requirement for the variable and symbol tables can be detected from the variables dictionary scan; the generation of adcons for external files and entry names is triggered by references to these items in the text.

2. For the construction of argument lists used in execution-time calls to the library and other procedures. In this case an adcon is required for an operand within an ARG text table. A wide range of different adcons can be produced depending upon the type of operand: variable, constant, null operand, DED, temporary, record/key descriptor, file, locator, entry name, symbol table, symbol table element, condition CSECT, controlled anchor slot, and label.

If the adcon references an item in static, other than a variable, then the address of the item is replaced in the adcon by this phase and the ARG table can be deleted (static variables are not mapped until Phase PE and so the adcons must be filled by Phase SI). Adcons for temporaries and non-static variables must be filled with code at execution-time: in these cases, the ARG table is not deleted.

Static Initial: This phase performs the complete allocation of static initial variables. Static initial is detected in the input by the presence of SINIT text tables (for scalars),and IASSN, AID (start of array iteration), and ENDAID (end of array iteration) for arrays. All these tables are deleted by this phase.

Upon encountering a static initial table, the phase first outputs a constant descriptor with sufficient empty storage to hold the whole of the variable, whether it is scalar, array, or structure. In any reference to the same variable it then first determines, using the aggregate table in the general dictionary if required, the offset within this storage at which the particular variable being initialized is located. The source constant must then be converted to the data attributes of the variable and moved into pre-allocated storage. Unlike normal source constants, all replication factors must be applied at this time; AID and ENDAID text tables must be processed to give the correct initialization of arrays.

Text Deleted by Phase PA

This phase will delete flow unit headers (but not associated text) for those flow units for which bit 4 of the IFOF3 field of the flow unit header (see figure 5.99) has been set.

STORAGE ALLOCATION (PHASE PE)

This phase identifies all items that require allocation of storage (except those items that have their storage in the constants pool) and allocates static or automatic storage as required. This can necessitate the mapping of static and automatic storage for up to 255 blocks in any one compilation. This phase does not allocate storage for adjustable items; variable data areas for these items are defined by other phases (KK, IQ, OC, and OX).

In order to minimize the padding required to maintain the correct storage boundary alignments, storage is mapped in decreasing order of alignment requirements for the items. In order to minimize the amount of addressing code required, the size of each item is also taken into account when storage is mapped. Items are allocated to three size classes, according to whether their size is eight bytes or less, between eight bytes and 2048 bytes, or greater than 2048 bytes. Within each storage class, storage is allocated in order of decreasing alignment. The storage dictionary is built by this phase. An entry is made for every item that has an entry in the variables dictionary, and entries in both dictionaries have similar alignment, length, and sequence.

Some entries in the general dictionary are also scanned by this phase, and modified to indicate storage requirements.

Phase PE also scans the main text stream. The primary reason for doing this is to look at all register temporaries whose use spans labels are marked as 'global'.

PHASE INPUT

Input to the phase consists mainly of the variables dictionary and the general dictionary. The scratch page, containing phase-to-phase information output by Phases PC and PA, is accessed for the purpose of adding more information.

PHASE OUTPUT

This phase builds the storage dictionary, in which entries indicate the size and location of storage allocated in static or automatic storage regions for each variable that has an entry in the variables dictionary. Entries in the storage dictionary have the same length and alignment as entries in the variables dictionary, and are made in the same sequence (see figure 5.28).

Entries in the general dictionary for record descriptors and key descriptors have storage allocation information inserted by this phase.

The page containing storage allocation information, which was passed from Phases PC and PA, is output with its original content unaltered, but with the addition of information about the base numbers for each variable for which storage has been allocated. This information is passed in a table, which is referred to as the base table.

The variables dictionary is not altered by this phase, but some entries in the general dictionary have additional information inserted.

The main text stream has been modified to ensure that register tempories are 'global' if required.

Building the Storage Dictionary

The storage dictionary is used to hold information about the storage required at execution time for each variable item. This phase makes a sequential scan of the variables dictionary and, from the information held in the dictionary entries, determines the storage requirements for each item.

At execution time, the output from each compilation requires a dynamic storage area (DSA) for each block, and a static storage area (storage in a DSA is referred to as automatic storage). As each compilation can contain up to 255 blocks, this phase may be required to map up to 256 areas of storage. Each area can contain many variables of different sizes and with different storage boundary alignment requirements.

Each variables dictionary entry is examined to determine whether the item requires static or automatic storage. If automatic storage is required, the block number is used to indicate the DSA in which storage is required. For each storage area, an offset counter table is built in the phase working area to facilitate mapping of storage for the various items.

The storage-mapping requirements for each variable can be classified according to its storage boundary alignment and the size of storage required for the item. Storage boundary alignment requirements are classified according to whether the item must be aligned on a doubleword, word, halfword, byte, or bit storage boundary. Storage size requirements are grouped into three classes: small-sized items requiring up to eight bytes (Class A); medium-sized items requiring nine to 2048 bytes (Class B), and large-sized items requiring more than 2048 bytes (Class C). Combinations of the two requirements result in 15 classes of storage mapping requirements, and a further three classes are used for miscellaneous items used for addressing purposes.

Storage is mapped by calculation of offsets from the origin of storage allocated for variable items within a storage area. The storage mapping classifications (described above) are used to facilitate calculation of the offsets.

The Offset-Counter Table used for mapping storage for each DSA is shown in figure 2.26.

OFF8A	Class A size, doubleword aligned
OFF8B	Class B size, doubleword aligned
OFF8C	Class C size, doubleword aligned
OFF4A	Class A size, word aligned
OFF4B	Class B size, word aligned
OFF4C	Class C size, word aligned
OFF2A	Class A size, halfword aligned
OFF2B	Class B size, halfword aligned
OFF2C	Class C size, halfword aligned
OFF1A	Class A size, byte aligned
OFF1B	Class B size, byte aligned
OFF1C	Class C size, byte aligned
OFFBA	Class A size, bit aligned
OFFBB	Class B size, bit aligned
OFFBC	Class C size, bit aligned
OFFBT	Bit Offset counter
OFF4P	Parameter counter
OFFADC	Addressing Adcon counter

Figure 2.26. Offset counter table for a DSA

Storage is mapped so that items with the highest storage boundary alignment stringency are stored closest to the origin of the variable storage area. Variables are grouped first by size and secondly by alignment.

As storage is mapped for an item, a storage dictionary entry is created for the item. Entries in the storage dictionary have the same alignment, length and sequence as entries in the variables dictionary. The value of the offset counter for the item is inserted in the storage dictionary entry. This value is modified during later processing in the stage. The format of entries in the storage dictionary is shown in figure 5.28.

Base Numbering

To facilitate addressing of items, allocated storage is divided into blocks of 4096 bytes. Each of these blocks is called a region, and each region is uniquely identified by its base number.

Base number 1 is associated with the first region of static storage, base number 2 with the second region of static storage, and so on. When all required static storage has been mapped, the next available base number is associated with the first region of automatic storage. The process is continued until all required DSAs have been mapped. When base numbers have been allocated, each item for which storage is allocated can be addressed by its offset from the origin of a particular region of storage, identified by its base number.

The base number associated with a variable is inserted in the YSBSE field of the storage dictionary entry for the variable. It is also inserted in a table, referred to as the base table, which is built in the scratch page passed from Phase PA, (the constants relocation and base numbers information page) and passed to Phase PI. The storage dictionary reference of a variable is used to index its entry in the base table. No entries are made in the table for indirectly-addressed variables, (e.g., based or controlled variables), and therefore initialization of the table causes these variables to appear with an apparent base number of zero.

Relocation of Offsets

The offsets calculated for the storage for each variable require to be relocated as offsets from the start of each DSA or static storage area. To do this, a relocation factor must be calculated and applied to each offset.

At the beginning of each DSA, storage is required for housekeeping information. The format of this information is standard for all DSAs, but it contains a variable length field which is used to store the ON-cells for the program block. To calculate the length of this field, the block header entry in the general dictionary is accessed to find the number of ON-cells in the block (stored in the entry by Phase PA). This information enables the total amount of storage required for housekeeppng information to be calculated, and the offset from the start of the DSA at which variables can be stored can thus be ascertained.

Using this relocation factor, and the offset counters previously calculated, a scan is made of the storage dictionary for the purpose of relocating the offset of each variable for which static or automatic storage has been allocated. When this process is complete, the offset counter tables are discarded.

Storage for Parameters

In order that parameter lists can be addressed directly, storage for the parameter list for a block is reserved in temporary storage within the appropriate DSA. The temporary storage is not allocated by this phase. During the scan of the variables dictionary, parameters are recognized, and the number of parameters for each block is counted and saved for use by the addressing phase (Phase PI). The requisite amount of temporary storage is determined by a routine in that phase.

Phase PE allocates a unique base number, separate from the base number for the region, to each parameter. This allows each parameter to be addressed directly from its base. The base number is inserted in the storage dictionary entry for the parameter.

Storage for Record and Key Descriptors

When the phase has completed its building of the storage dictionary, the general dictionary is examined for the presence of skeleton record descriptors or key descriptors built by Phase KM. If a chain of such entries exists, flag bytes in each entry are examined to determine the required location of the descriptor at execution time.

Using information already determined for storage allocation, a base number and offset value is inserted in each dictionary entry to allocate storage in the apprcpriate region.

Text Processing

Two scans of text are made. During the first scan information is collected on register temporary usage (e.g., when the temporary is used and for what purpose).

During the second scan of text the information on register temporaries is used to determine if a temporary should be marked as 'global', or if it can be made 'local', or even eliminated.

ADDRESSING OF STORAGE (PHASE PI)

The phase generates information that enables any item for which storage is allocated to be addressed. The information is either inserted in text tables that indicate the addressing code to be generated, or passed in a separate text stream to enable a later phase to generate the required code.

1. Offsets of items in the constants pool are relocated relative to a static storage base number.
2. Text tables are generated to indicate prologue code required for addressing various items (e.g., the temporary storage area for a block), or for initialization of various items (e.g., aggregate locators, record and key descriptors).
3. Text tables are generated to indicate the code required for addressing indirectly addressed variables (e.g., based, controlled).
4. Information is passed to Phase QI to enable variables that are external to the current block to be addressed.
5. Temporary operands are allocated storage offsets within internal classes of temporary storage. Information is passed to Phase QI to enable these offsets to be relocated.

This phase also deletes from the text flow units that are flagged as "unreachable".

Partially optimized blocks are treated in the same manner as wholly unoptimized ones. Bit 7 of the IFOF1 field in the flow unit header (figure 5.97) will be off.

PHASE INPUT

Input to the phase consists of the sequential Type-2 text stream output from Phase PA, the text page containing constants pool and base numbers information collected by Phases PC, PA, and PE, and another text page containing block structure information output from Phase PE. The storage, general and variables dictionaries are accessed as required. The pseudo constants pool text page(s) is not accessed by this phase.

PHASE OUTPUT

This phase generates modified or additional text tables which indicate the addressing code that is required to be generated by later phases. The text tables are output so that a logical sequence is maintained without the use of overflow pages and chains.

A text page containing addressing and temporary storage information is passed to Phase QI separate from the main text stream.

No changes are made to entries in the variables or general dictionaries. The offset values in some entries in the storage dictionary are modified.

PHASE OPERATION

The main text stream is scanned sequentially. Text tables, flow unit headers, etc., that do not require processing by this phase are copied directly to the output text pages. Where a situation that requires processing is recognized, dictionaries are accessed and text tables are modified or generated so that a logically sequential main text stream output is maintained without the need for overflow pages or text table chains.

The text page containing constants pool and base numbers information is copied into the phase working storage for ease of access. The text page containing block structure information is accessed as required.

Relocation of Constants Pool Offsets

If a text table contains a reference to an item for which storage in the constants pool has been allocated, the offset shown in the text will be an offset within a particular alignment/size classification. Using the information supplied by Phase PE, this phase relocates the offset so that it is relative to a particular region of static storage, and inserts the appropriate base numbers in the text.

When Phase PI receives control, allocation of static storage will be similar to that shown in figure 2.27.

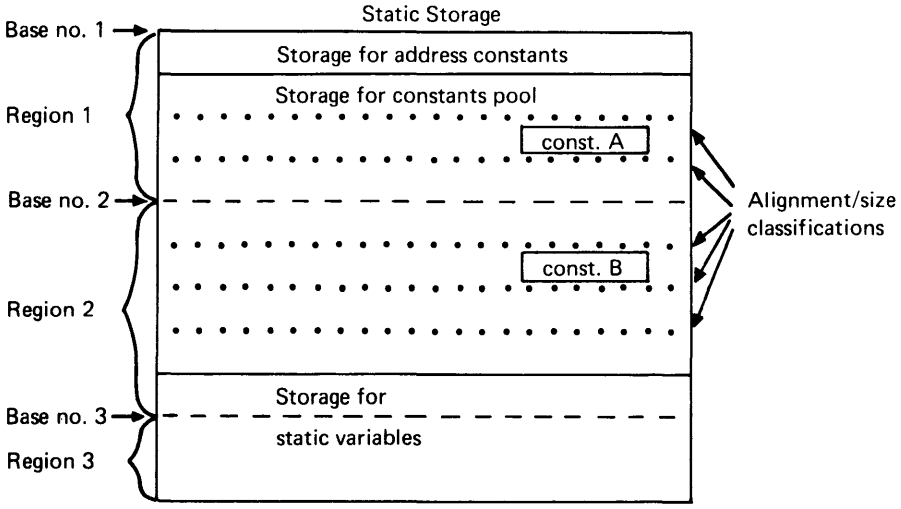


Figure 2.27. Relocation of constants pool offsets

The function of Phase PI is to modify the relocation factors inserted in the text by Phases PC and PA, allowing for the static storage allocated for address constants. In the illustration shown, the modified relocation factor would allow the offset of constant A to be relocated relative to Base Number 1. For constant B in the illustration, the

relocation factor would have to be further modified so that the offset would be relocated relative to Base Number 2. All offsets are calculated relative to the origin of the appropriate region, and the relocated offset and the base number are inserted in the text.

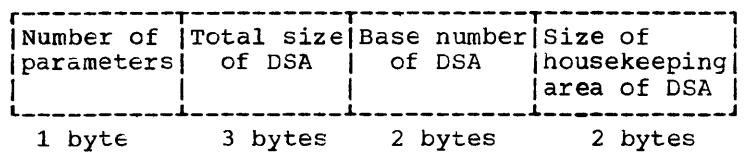
Generation of Prologue Code for Addressing and Initialization

When a prologue-end (PEND 01) text table is found, the storage dictionary is scanned for any items which require addressing or initialization code to be generated in the prologue. Text tables indicatng the required code are generated and output before the PEND table is copied to output.

GENERATION OF ADDRESSING CODE: There are a number of cases where addressing code (i.e., code required to load the base of an item into a register) is required in the prologue. Examples of such cases are the code required to address the first STATIC ONCB in a block, or code required to address the temporary storage area for a block. In such cases, the required offsets and bases are determined by this phase, and generated in text tables that indicate the code required.

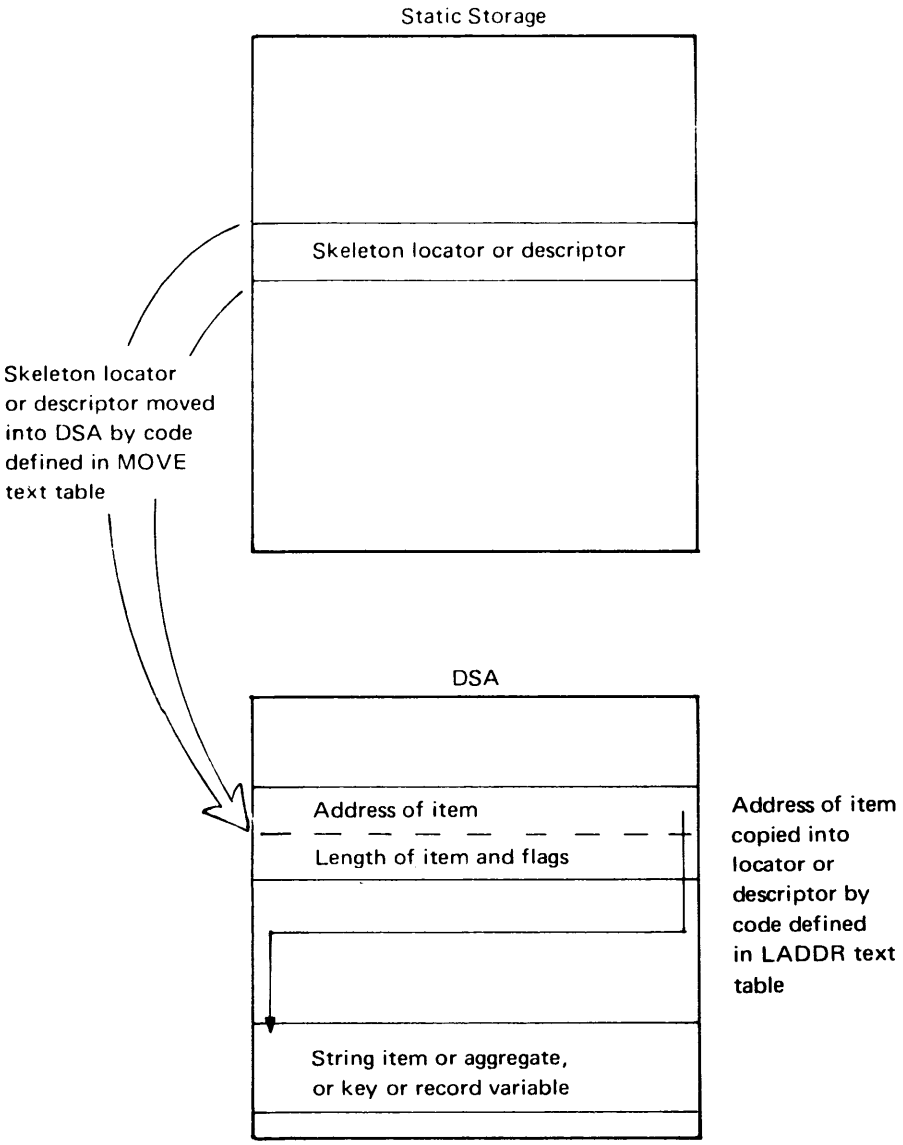
The address of the first STATIC ONCB is required to be inserted at a standard offset (92) from the origin of the housekeeping area of the DSA. The address of the ONCB is determined from the constants pool information passed from Phase PA, and an LA text table is generated to load the address into the predetermined offset.

Information in the block-structure-information page passed by Phase PE is used to calculate the address of the temporary storage area of a DSA. This page contains a number of 8-byte entries, each of which contains information about the DSA for a particular block. The format of each entry is as follows:



Using this information, the offset from the base of a storage region of the start of the temporary storage area is calculated. The temporary storage area is then allocated its own region and base number by obtaining the next available base number from the XNABN field of XCOMM and applying it by generation of a BADDR text table.

GENERATION OF INITIALIZATION CODE: This phase generates text tables indicating the prologue code required to initialize locators for aggregates and string items, and for some key and record descriptors. Static storage for the skeleton descriptors will have been allocated (in the constants pool) by Phase PA. This phase generates text tables to indicate code required to copy the skeleton descriptor or locator into the DSA and insert the address of the aggregate, string item, or record. The process is illustrated in figure 2.28.



Code for the operations illustrated is defined in text tables shown:

MOVE	Offset and base no. in static	Length of skeleton	Offset and base no. in DSA
LADDR	Offset and base no. of item	Length of item	Offset and base no. of locator or descriptor

Figure 2.28. Initialization of locators and descriptors

For aggregate or string item locators, the necessary information is obtained from entries in the storage dictionary. For record and key descriptors, the necessary information is obtained by scanning a chain of entries in the general dictionary.

Generation of Inline Addressing Code

During the scan of text, the storage-dictionary reference of each variable operand is used to index the base table created by Phase PE. The base number of the variable is inserted into the IBA1, IBA2, or IBA3 field of the text table as applicable. This processing can only be performed for static internal and fixed automatic variables. Indirectly addressed variables (e.g., parameter, controlled, based, etc.,) do not have annotated entries in the base table, and therefore are shown with an apparent base number of zero. This indicates to the phase that the INDAD routine must be used to generate text tables that define the required inline addressing code. The storage dictionary fields that indicate the base and offset of the variable, and of the variable's locator and/or descriptor, are accessed as applicable for the required information. A table indicating the information used is contained in the phase listing.

Addressing Variables in Outer Blocks

If a text table contains a reference to a variable declared in an outer block, addressing code is required to enable the variable to be accessed. This phase does not generate the addressing code, but sets up information which enables Phase QI to do so. The information is passed to Phase QI on a separate text page (or pages), referred to as the addressing and temporary storage information page. The TA of this page (or chain of pages) is stored in the XACDRF field in XCOMM.

Allocation of Temporary Storage

If a text table contains a reference to a temporary operand, the code byte of the operand is examined to determine whether storage is required. If storage is required, the amount of storage can be determined from the DED of the temporary operand.

Within the temporary storage area of each DSA storage is divided into four classes:

1. Parameters storage
2. Global temporary register storage
3. Global temporary identifier storage
4. Local temporary identifier storage

Each temporary operand is allocated storage at a particular offset from the origin of the appropriate storage class. The offset value is used to overwrite the identification number of the temporary operand in the text.

If the code byte of a temporary operand indicates that it is being used for the last time, the storage allocated for it is freed and can be re-used. This technique helps to reduce the amount of temporary storage required.

When all the required temporary storage has been allocated, information about the temporary storage area is inserted in the addressing and temporary storage information page, to enable Phase QI to relocate the allocated offsets.

OPTIMIZED ADDRESSING (PHASE QI)

Phase QI completes the allocation of storage and addressing of items not addressed by Phase PI. Where addressing code is generated, note is made of whether or not the code has been optimized by phases in the global optimization stage. Division of the code into flow units, and identification of back dominators by optimizing phases, enables optimization of some addressing code. Functions performed by this phase include:

1. Insertion of addressing code in the optimum places in the code sequence.
2. Relocation of offsets in temporary storage areas.
3. Modification of DSA sizes, to allow for temporary storage and address constants required to address items in outer blocks.
4. A number of operations in preparation for register allocation, but which are independent of register status. These include some loop-processing operations.

The phase will also recognize partically optimized blocks and will perform full register allocation for optimized DO groups.

PHASE INPUT

Input to the phase consists of the main text stream, with Type-2 text tables organized in logical sequence. The text page output from Phase PI, containing addressing and temporary storage information, is accessed and copied into the phase working area.

If the phases of the global optimization stage have been executed, the summaries of variables usage information inserted in the general dictionary by Phase OE are accessed. No other dictionary sections are accessed by this phase.

PHASE OUTPUT

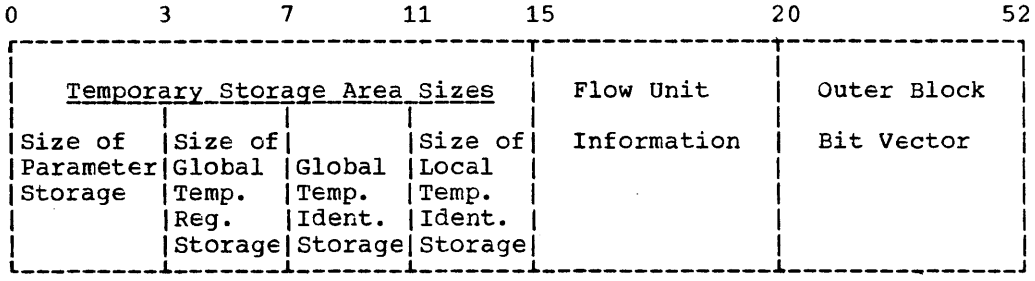
Output from the phase consists of the main text stream, in which text tables required for generation of addressing code have been inserted in logical sequence, and the storage offsets in references to temporary operands have been modified.

PHASE OPERATION

The text is scanned sequentially. Text tables, etc., that do not require processing by this phase are copied directly to the output text stream. Where situations requiring processing are recognized, text tables are modified or additional text tables are generated, and output in the text stream in logical sequence.

Addressing and Temporary Storage Information

Phase PI passes a text page containing information which enables Phase QI to generate addressing code for variables in outer blocks, and to relocate the offsets of items for which storage has been allocated already. For each program block (i.e., each section of code that requires a DSA), Phase PI passes information in the format shown below:



The flow unit information consists of:

1. Identification byte (X'9A'). This serves to distinguish flow unit information from 'endpage'.

2. Flow unit identification number (IFOFUN in FUH)

3. Back target identification number (IFOBTN in FUH)

4. Back dominator identification number (IFOBD in FUH)

5. Flow unit flags (IFOF2 in FUH)

The information varies according to whether the phases in the global optimization stage have been executed or not. If the compilation has not been globally optimized, the flow unit information word will be undefined because the information will not be available. Also, the outer block bit vector will have bits set for all the outer blocks containing variables referred to in the current block.

If the compilation has been globally optimized, the entry for the current block will contain the temporary storage area size information. The flow unit information and an outer block bit vector are repeated for each flow unit in the block. The outer block bit vector will have bits set to indicate only those blocks containing variables addressed from within the particular flow unit.

As each block header is seen in the sequential text scan, the PROCZ routine is called to copy the addressing and temporary storage information for the block into the phase working storage, for ease of access.

Addressing Code for Variables in Outer Blocks

This phase generates text tables which indicate the code required for addressing variables declared in blocks outside the blocks in which the variables are referred to. The place in which the addressing code is inserted depends upon whether the compilation has been globally optimized (i.e., phases in the optimization stage have been executed) or not.

If the compilation has not been globally optimized, the addressing code is generated in the prologue code for the appropriate block. When a

BADDR (02) text table is seen, the INAC routine is called to generate text tables indicating the prologue code required to chain back to any outer blocks requiring addressing.

The blocks which require addressing are indicated in the outer block bit vector passed by Phase PI. The chaining code enables access to a base at a standard offset from the start of the relevant DSAs, so that the variable can then be addressed relative to that base. The bases for outer blocks are saved in the temporary storage area of the current DSA, and this phase allocates the temporary storage.

If the compilation has been globally optimized, then the addressing code can be optimized. Instead of inserting the addressing code in the prologue, where it must be processed for every execution, the addressing code is inserted at a position in the executable code, where it is only executed if required. This optimization is enabled by the flow unit information generated by the phases in the optimization stage, and passed by Phase PI. It is only possible for the first 256 flow units.

In the optimized case, the addressing code is generated when a flow unit header table is seen. The outer block bit vector for the particular flow unit is examined to determine the outer blocks which must be chained back to.

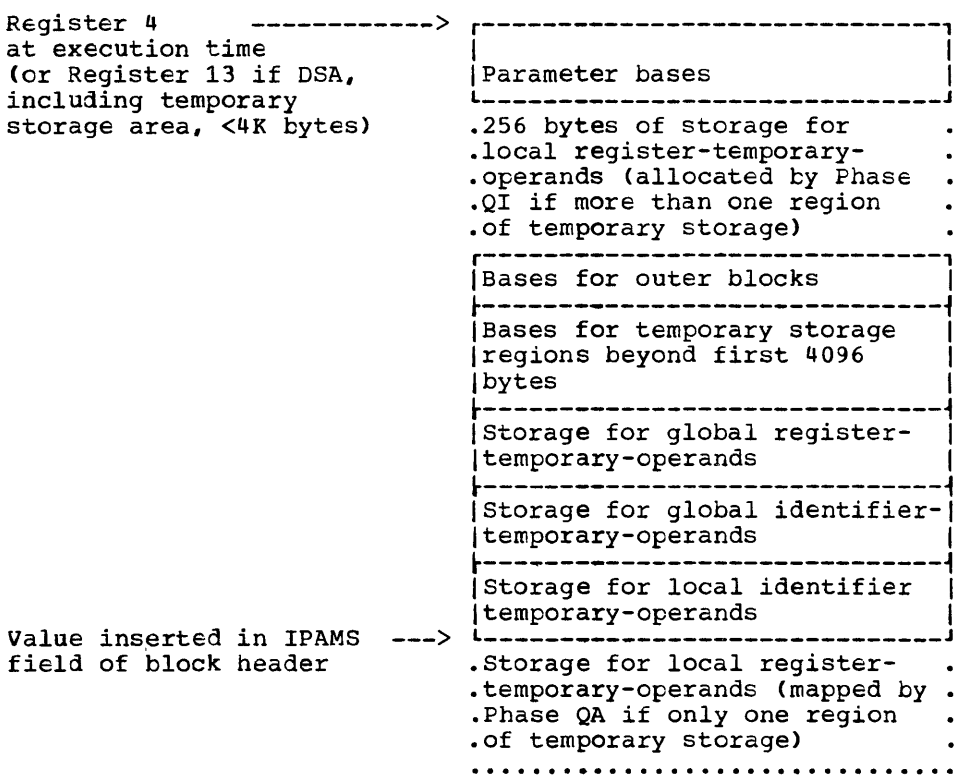
The flow unit information is also examined to determine the situation of the flow unit. If the flow unit exists in a loop, then a logical OR operation is performed to merge the addressing information into the back target, thus removing the addressing code from the loop.

If a flow unit has a back dominator, then addressing code previously generated is deleted and the addressing data is carried forward to generate code in the flow unit, thus avoiding duplication of code. As in the non-optimized case, the bases for outer blocks have storage allocated in the temporary storage area of the appropriate DSA.

Relocation of Temporary Storage Offsets

When the amount of temporary storage required for bases for outer blocks has been determined, most of the information required for the relocation of the temporary storage offsets allocated by Phase PI is available. All that remains to be done is for Phase QI to ascertain whether it has to make an allowance for the storage of local register-temporary-operands, or whether that storage can be mapped directly by Phase QA.

The need for storing register-temporary-operands implies that there are no free registers; therefore they must be stored in the first region of temporary storage for the block so that an additional register is not required to enable them to be addressed. Because the maximum number of local register-temporary-operands that can be active is restricted to 32, the maximum amount of storage that can be required for them is 256 bytes. If the known requirement for other items in temporary storage is less than 3840 bytes (4096 minus 256), the area for storage of local register-temporary-operands can be mapped directly by Phase QA when precise requirements are known. If the known requirement for other items in the temporary storage area is greater than 3840 bytes, Phase QI allocates the maximum requirement of 256 bytes in the first region of temporary storage for the block, immediately following the storage for parameter bases. When this requirement has been determined, the temporary storage area for the block is mapped as follows.



Information about the DSA for the current block is inserted in the appropriate block-header text table. The known total size of the parameter bases area is inserted in the IPAMS field, the total size of the DSA is inserted in the IDSASZ field, and the number of regions in the DSA (excluding the temporary storage area) is inserted in the IFCHN field. The offsets (from the start of the temporary storage area) of the local register-temporary-operands storage and the global register-temporary-operands storage are inserted in a KONST(12) text table generated after the block-header table.

The sequential scan of the text tables is resumed. As each text table that may contain a temporary operand is found, the RDIR routine is called to examine the operands. If a temporary operand is found, the RELOC routine is called. This routine examines the storage offsets inserted by Phase PI. These offsets are calculated from the origin of the temporary storage for that class of item. RELOC calculates the offset from the base of the temporary storage area. If the modified offset is greater than 4096, a new base number is allocated and the offset recalculated. The offset in the operand reference is overwritten with the relocated offset.

Loop Processing

Phase QI examines iterative DO-loops for situations where the code can be optimized. In general, this optimization consists of modifying text to enable Phase QA to allocate registers for a BXLE loop wherever possible. Because much of the information used by Phase QI is generated by phases in the Global Optimization Stage, the extent to which Phase QI can set up this optimization varies according to whether or not phases in the Global Optimization Stage have been executed.

The processing includes:

- Determination of situations where BXLE loops can be generated.

- Replacement or modification of DINC text tables.
- Tests for global temporary operands that are set within loops.
- Movement out of loops of parameter-addressing code.
- Determination of flow-unit range for a loop.
- Processing of based loop-control variables.

Testing for BXLE Loops: On input to the phase, each iterative DO-loop is preceded by an SCI text table, which indicates the loop comparand and increment values. The IOP2 field of the SCI text table indicates the type of the loop as follows:

SCI00 - unnested or innermost loop

SCI01 - all loops in nest other than the innermost

SCI02 - loop with multiple specifications (inner or outer)

In addition, each loop is also preceded by an ASSN, CONV, or MOVE text table, in which bit 0 of the IOP2 field indicates that the text table sets the initial value of the loop control variable (lcv). The lcv may also be set by library call. In this case the library call is preceded by a KONST(15) text table which indicates that an lcv is being set, and gives the identity of the lcv.

For each DO-loop, the SCI text table and the text table setting the lcv are examined to see whether the lcv can be permitted to flow into the loop in register 5. The criteria for this are:

- The lcv, the comparand, and the increment must be fixed binary values.
- The lcv, the comparand, and the increment must have the same scale. If the scales differ, shifting will occur during comparison or incrementation at the end of the loop and register 5 might not contain the correct value during subsequent executions of the loop.
- The loop must not have multiple specifications.
- The loop must not occur in a remote format list or within an edit-directed I/O statement which has a remote format list. This is because register 5 is used as a base for the format code.
- The lcv must not be an abnormal variable as indicated by the program optimization entry in the general dictionary. If global optimization has not been performed, this entry will not be available, and all variables except temporary operands are considered to be abnormal.

If these conditions are not satisfied, bit 0 in the IOP2 field of the lcv-setting text table is set off. If the conditions are satisfied, a KONST09 text table is generated after the SCT text table to ensure that register 5 is allocated. Further tests are then made to determine whether a BXLE loop can be generated. The additional conditions for this are:

- The loop must be unnested or an innermost nested loop.
- The SIZE condition must be disabled, unless the REORDER option is specified. This is because no interrupts are raised from a BXLE instruction.

- Both the comparand and the increment must be constants in the first region of static storage. If they are in any other region there may be difficulty in reloading then into registers 10 and 11 if they are lost during execution of a loop.

If the conditions are not satisfied, the SCI text table is deleted. If the conditions are satisfied, the SCI text table is copied to output to indicate to Phase QA that a BXLE loop is to be generated if registers can suitably be allocated.

Processing DINC Text Tables: Each iterative DO-loop is terminated by a DINC text table, followed by a BC text table. If the loop is not an optimizable inner loop, the operator code of the DINC text table is changed to PLUS. If the loop control variable enters the loop in register 5, an ASSN text table assigning the loop control variable to register 5 is generated immediately prior to the DINC or PLUS text table, thus ensuring that the loop control variable enters the next execution of the loop in register 5.

If, in the case of an optimizable inner loop, there are BADDR text tables (containing addressing code for the BC) between the DINC text table and its associated BC text table, the DINC text table is moved to immediately precede the BC text table. This enables Phase QA to replace this pair of text tables with a BXLE. If any other text tables appear between the DINC and BC text tables, a BXLE cannot be generated; the DINC text table operator is changed to PLUS and the SCI text table in the output text stream is changed to NULL.

If the BY clause associated with a loop contains a variable or an expression, the loop is terminated by two DINC text tables. The IOP2 field of the first one is set to X'09' to indicate to Phase QA that it is not the end of the loop. This DINC text table is changed by Phase QA to BXLE (executed if the BY clause is positive) and the second DINE text table is changed by Phase QA to BXH (executed if the BY clause is negative).

Examination of Global Temporary Operands: Phase QA moves load instructions which involve global temporary operands out of optimizable inner loops. This movement is not possible for global temporary operands that are set within the loop. In preparation for this processing, Phase QI examines all text tables within a loop in which a global temporary operand can be set. If a global temporary operand is found in such a position, a bit is set in a bit vector that is passed to Phase QA in the text, following the SCI text table. The bit vector is 32 bytes long and each bit offset is calculated from:

storage offset for temporary operand/4

If this value exceeds 256, the load instruction is assumed to be unmovable.

Movement of Parameter addressing Code: Text defining code to address a parameter consists of a sequence of BADDR text tables. The start of a sequence is indicated by a BADDR06 or BADDR07 text table in which the IBA2 field (see figure 5.93) contains a base number. The sequence can contain any BADDR text tables with an IOP2 value of 06-09, and the end of a sequence is indicated by the presence of any other text table or by the start of a new sequence.

As a loop is processed, subroutine BADCHK is called to search for and process parameter-addressing sequences. If such a sequence is found within an optimizable inner loop, it is moved outside the loop. This is enabled by the generation of a series of twelve NULL text tables following the SCI text table when this table is processed. These tables are overwritten by the BADDR text tables that are moved out. Each moved sequence is followed KONST10 text table, which indicates a dummy store of the resultant base. Inside the loop, the addressing sequence before each parameter is replaced by a KONST11 text table which represents a

dummy load of the stored base. At the end of the loop (preceding the DINC text table) a KONST11 and a KONST08 text table are generated to ensure that the base is in the correct register for subsequent executions of the loop. Counts of the number of sequences moved out of the loop are maintained in the KONST10 and KONST11 text tables.

Determination of Flow-unit Range: If there is a branch out of a loop, and if the loop-control variable is busy on exit, the loop-control variable must be stored before the branch out. QI inserts the identifying numbers of the flow units in the loop into the IREF3 field of the SCI text table, to enable Phase QA to detect possible branches.

Processing Based Loop-control Variables: If a loop-control variable is based, the value of the base at the end of the loop must be the same as its value at the loop head. For example, if a loop is based on a pointer P, the value of P at the head of the loop is used as the base of the loop-control variable at the end, even though the value of P may be changed in the loop.

Phase QE builds a stack which can contain 50 six-byte entries (one entry for each possible level of DO-loop nesting). If a loop-control variable is a Q-temp. which was defined in a BADDR01 text table, the loop-control variable is based. An entry which contains the Q-temp. number and the base number is made in the stack, and a KONST10 text table is generated to indicate a dummy store of the base. At the end of the loop, if the loop-control variable is represented by a Q-temp. in the operand 1 field of the DINC text table, the most recent stack entry is accessed. If the Q-temp. Number in text and in the stack are the same, the stack entry is deleted and a KONST11 text table is generated to indicate a dummy load of the base. Counts of the operations are maintained to enable elimination of redundant store and load operations by Phase QA.

REGISTER ALLOCATION (PHASE QA)

This phase assigns absolute register numbers to operands and operand bases which require the general or floating-point registers at execution time. The phase also allocates storage for temporary registers when required.

Registers are allocated in a way which enables execution to be performed as efficiently as possible. If the compilation has been globally optimized, use is made of the control-flow information so that, whenever possible, items are carried in registers within a flow unit and across flow unit boundaries.

Inner do-loops are optimized by generating text tables which indicate BXLE instructions and, where possible, carrying items into the loops in registers.

This phase will also recognize partially optimized blocks and will perform full register allocation for optimized DO groups.

PHASE INPUT

The only input to this phase is the main text stream, consisting of Type 2 text tables in logically sequential order, with base numbers inserted for all operands except temporary registers.

PHASE OUTPUT

Output from the phase consists of the main text stream, with actual register numbers inserted in the IREG1, IREG2, and IREG3 fields of the text tables, e.g., IREG1 = X'FD' indicates that, for the first operand in the text table, Register 15 is assigned as a work register and Register 13 is assigned as a base register. In addition, flag bits are set in the IST1 field of a text table to give the following indications:

IST1 bit no.	Indication when set
0	Operand 1 in a register
1	Operand 1 to be retained unmodified in a register
2	Operand 2 in a register
3	Operand 2 to be retained unmodified in a register
4	Operand 1 base in a register
5	Operand 2 base in a register
6	Operand 3 base in a register
7	Operand 3 to be stored

PHASE OPERATION

The main text stream is scanned sequentially. When the scan reaches a text table which affects register usage, either by requiring base registers or work registers, or by affecting the control flow within a compilation, a branch is made to a routine which processes that particular type of text table. Each routine is identified by the text-table mnemonic concatenated with a Z character, e.g., PLUS text tables are processed by the PLUSZ routine, CONV by the CONVZ routine, flow unit headers by the FLOWZ routine, etc. The branch to the appropriate routine is made by indexing an addressing vector with a value of 2*IOP1 for the particular text table.

Each routine calls a set of subroutines to find suitable registers for the text table operands. These subroutines access a table, created and maintained in the phase working area, which indicates the current content of registers. This table is called the Register Usage Table (RUT). It indicates the items within a flow unit that are already held in registers, and is updated as registers are allocated.

At the beginning of each flow unit, a new RUT is built. If the compilation has been globally optimized, up to five previous RUTs are saved. These saved tables are accessed for previous allocation information, which is used to set up initial conditions before allocating registers in the current flow unit. Whenever possible, registers are allocated so that items are carried in registers across flow unit boundaries, to be merged with other input. This use of registers reduces the number of storing operations, and thus reduces the space and time required for execution.

When allocating a register to an item, a hashing technique is used to convert the dictionary reference of the item to a decimal value between 1 and 15. The resulting value indicates the identifying number of the register to be allocated to the item if it is available. Registers allocated in this manner are referred to as preferred registers.

The initial status of registers used in inner Do-loops is saved in a special phase

The initial status of registers used in inner Do-loops is saved in the appropriate register usage table. Items required within the Do-loop are loaded into any spare possible. At the end of the loop, the initial status of the registers is restored by reference to the appropriate RUT. This processing is only performed when global optimization is specified.

The allocation of registers conforms to the execution time usage shown in figure 2.29.

Register Number	Dedicated Registers	Work Registers (plus special usage)	Preferred Registers (selected when available)	Comments
0		General use, or arithmetic operations		
1	Address of DTF for inline record I/O routines	General use, Arg. list pointer, or EDMK instruction		
2	Address of program base (or FCB for inline record I/O)			Normally saved by library routines
3	Address of base of first STATIC region			
4	Address of base of first temporary storage region (if total size of DSA >4K bytes)			
5		General use, or static chain back on entry to block	Register for Do-loop control variable	Used to pass arguments to library routines; Used to pass arguments to CGSRs
6		General use		
7		General use		
8		General use		
9	Address of program base when R2 used	General use		
10		General use	Do-loop control when BXLE instruction used	
11		General use		
12	Address of task communication area			
13	Address of current DSA			
14		General use or Return When BALR used		Normally saved by library routines
15		Entry		

Figure 2.29. Allocation of general registers for program execution

ELIMINATION OF UNNECESSARY STORAGE OPERANDS (PHASE QE)

The main function of Phase QE is to eliminate stores of global temporary operands and stores of loop control variables at the head of BXLE loops when these stores are not necessary.

Phase QE is only loaded and executed, following Phase QA, when the compiler option OPT(TIME) is specified.

PHASE INPUT

The input to this phase is the main text stream from Phase QA, consisting of Type 2 text tables in logically sequential order.

The four text tables immediately following each block header table (PROC, BEGIN, etc.) are bit strips.

1. GBVEC shows which global temporary operands do not require storage.
2. LCVEC shows which BXLE loops require loop control variable stores at their heads. Those BXLE loops which may require stores each have a KONST(0C) text table at the point where the store should be made.
3. BLCVEC shows of the KONST(10) text tables are to be changed to stores. The KONST(10) is a dummy store of a base of a based loop control variable or of the base of a parameter of the head of a loop out of which Phase QI has moved the parameter addressing code.
4. ACMVEC shows which of the global temporary operands used as accumulators do not require storage.

PHASE OUTPUT

The output from Phase QE is passed direct to Phase SA in the final assembly stage. It consists of a sequential text stream (Type 2 text tables) with KONST (0C) text tables either replaced by loop control variable stores or removed, and, where possible, with text-table store flags cleared in operations involving global temporary operand results. Unused temporary register storage has been eliminated by relocating temporary operands and appropriately modifying the DSA size of each block.

PHASE OPERATION

The input text stream is scanned sequentially, and all valid text tables recognized.

For each text table, subroutine RDIR is called to examine operand 3. If it is an ordinary global temporary, then the bit in GBVEC corresponding to that global temporary is tested. If it is a global temporary being used as an accumulator, then the bit in ACMVEC corresponding to that accumulator is tested, the bit offset having been obtained from the IREF2 field of the ACCUM text table at the head of the loop. In either case, if the bit is set off, the store flag in the text table is cleared. This operation is only carried out for the first 256 global temporaries encountered, the remainder are always stored.

The bits in GBVEC or ACMVEC will have been set on by Phase QA if either the appropriate temporary has not been retained in a register throughout its lifespan, or the address of the temporary is required (implying that it is being passed as an argument).

During the text scan, the KONST(0C) text tables contained in certain BXLE loops cause vector LCVEC to be examined. The bit in the vector specified by the IREF2 field of the KONST(0C) table is tested. If this bit is set on the KONST(0C) table is changed to a loop control variable store at the head of the BXLE loop.

The KONST(0C) table is also changed to a store if the loop control variable is used in an I/O ON-unit or if it is used in a computational ON-unit with ORDER specified for the current block. Otherwise the KONST(0C) table is eliminated. This operation is only carried out for the first 256 BXLE loops; the remainder always have a store at the head.

KONST(0C) tables only appear at the head of BXLE loops which may or may not require loop control variable stores. These are:

1. Single flow unit loops.

2. Multiple flow unit loops where the control variable is not busy on exit.

If, in addition to the above, it is found that the loop control variable need not be accessed or addressed in main storage, then the store is, again, not required.

Also during the text scan, KONST(10) tables cause vector BLCVEC to be examined. The bit in the vector specified by the IREF2 field of the KONST(10) table is tested. If this bit is set in, the KONST(10) table is changed to a store of a base. If the bit is set off, the KONST(10) table is eliminated. This operation is only carried out for the first 256 KONST(10) tables. The remainder always become stores.

A KONST(10) table must be changed to a store if the base is lost from its register before the end of the loop in which it is referenced. If this is the case Phase QA sets the appropriate bit.

This stage consists of seven phases. The first four phases examine the text generated by preceding phases of the compiler, and generate code defined in or indicated by that text. The first two of the code generation phases, Phases SA and SQ, are loaded and executed for every compilation. The requirement for Phases SD and SC is determined by Phases SA and SQ, according to the types of text tables contained in the main text stream. Phase SK completes the processing required before final assembly of the object module by Phase SI. Phase SM generates any object listings specified by compiler options.

OBJECT CODE GENERATION (PHASES SA, SQ, SD, AND SC)

Each of these phases examines the whole of the text stream and selects particular types of text tables to process. The processing consists of generation of machine instructions indicated by each type of text table and the contents of each individual text table. Together, these phases translate the whole of the text into machine instructions, and insert information which enables later phases to assemble the code into a relocatable object module. If the LIST compiler option is specified, the phases also generate information for use by the listing phase (Phase SM).

Because each of the five phases performs a similar function, using a similar method of operation, the description of the five phases is combined. Only major differences are mentioned.

The prime difference between the phases is the type of text tables each one processes. The list for each phase is too extensive to list here; a complete list can be determined from the directory tables contained in each phase. Some of the main classes of text table are shown below:

Phase	Text Tables Processed or Partly Processed
SA	All system or housekeeping text tables, e.g., PROC, ENTRY, BEGIN, ONB, CALL, ONS, PEND, CHANE, VDA, etc. All statement header and label text tables. All GOTO and GOOB text tables All binary to float and float to binary conversion tables. Some addressing text tables.
SQ	All binary arithmetic operations. All floating-point arithmetic operations. Further processing of ONS text tables. All simple string operation tables, for example, MOVES, ANDs, ORs, for fixed length strings. Further addressing text tables.
SD	All conversion operations. All decimal arithmetic operations. All remaining conversions.
SC	All remaining character and bit string operations. All remaining logical and concatenation operations. All remaining addressing and mapping operations

Phases SA and SQ are always required. They determine the need for Phases SD and/or SC.

PHASE INPUT

Input to Phase SA consists of the main text stream output from Phase QA or QE. This text stream consists entirely of Type 2 text tables. The input to phases SQ, SD, and SC consists of a mixture of Type 2 text tables not yet translated, object code generated by preceding phases, and markers inserted for use by the label resolution phase (SK), the object module assembly phase (SI), or the object module listings phase (SM). This mixed input/output is referred to as extended code.

Each of the phases accesses the storage dictionary for information required for code generation. Phase SA accesses the general dictionary for information about each block, and also for the reference of the names dictionary entry for each procedure name. Phase SA creates general dictionary entries to hold information about ON-cells; these entries are accessed by Phase SQ.

PHASE OUTPUT

The output from Phases SA, SQ, and SD consists of the extended code described above. Phase SA also generates entries in the general dictionary, as mentioned above. The output from Phase SC consists entirely of object code (hexadecimal representation of machine instructions), with markers inserted to indicate all entry points, labels, branch points, and other places where Phases SK, SI, or SM must take special action.

Within the output stream, items are aligned as follows:

- Text tables - halfword aligned.
- Markers (and any associated code) - byte aligned.
- Object code instructions (and any associated listing information) - byte aligned.

To force halfword alignment, padding bytes X'0E' are inserted in the extended code.

No item (as listed above) is allowed to span page boundaries.

Phases SA and SQ also set bits in the XOPPHS1 field of XCOMM to indicate which of Phases SD and SC are required.

PHASE OPERATION

Each phase consists of a root module, which is common to all phases, and a non-root module, which is peculiar to each phase. The non-root module contains information about the text tables that are processed by the phase, and about the object code to be generated in response to the text tables it processes. The root module contains routines to generate the required extended code output. Each phase that is executed (Phase SA and SQ always, Phases SD and SC as required) makes a single sequential scan of the text.

In order to determine whether it processes a particular type of text table and, if so, to locate information about the object code to be generated, each phase contains two directories. The first-level directory is a table of 256 one-byte entries, in which each entry corresponds to a possible text-table operator-code (IOP1) value. When a text table is seen during a scan of the text stream, its IOP1 value is used to index the corresponding entry in this directory. If the value in the entry is zero, it indicates that the type of text table is not processed by the phase. In this case, the text table is copied unaltered to the output stream, preceded by a 4-byte marker which indicates that it is an unprocessed text table. The marker is deleted when the text table is processed by a later phase.

If the indexed entry in the first level directory contains a non-zero value, it indicates that the type of text table is processed by the phase. The value in the entry is determined by the number of IOP2 codes that can be used with the type of text table that the phase processes, and execution of the macro that defines the first-level directory enables the entry to point at a group of 2-byte entries in the second-level directory. Each entry in the second-level directory corresponds to a text table identified by an IOP1 value and an IOP2 value, and execution of the macro that defines the second-level directory causes each entry to point at an offset from the start of the non-root module of the phase. The IOP2 code of the current text table is used to index the appropriate entry in the second-level directory, and thus to obtain the address of the code-generation information for that text table.

Information about the object code corresponding to a text table is defined in a dummy section. It consists of a code-skeleton array, a bit-strip array, and (in some cases) special-case coding and a control mask which indicates processing required for special cases. The use of this information is described in following paragraphs. Use of the directories to locate information about an imaginary text table 0901 is illustrated in figure 2.30.

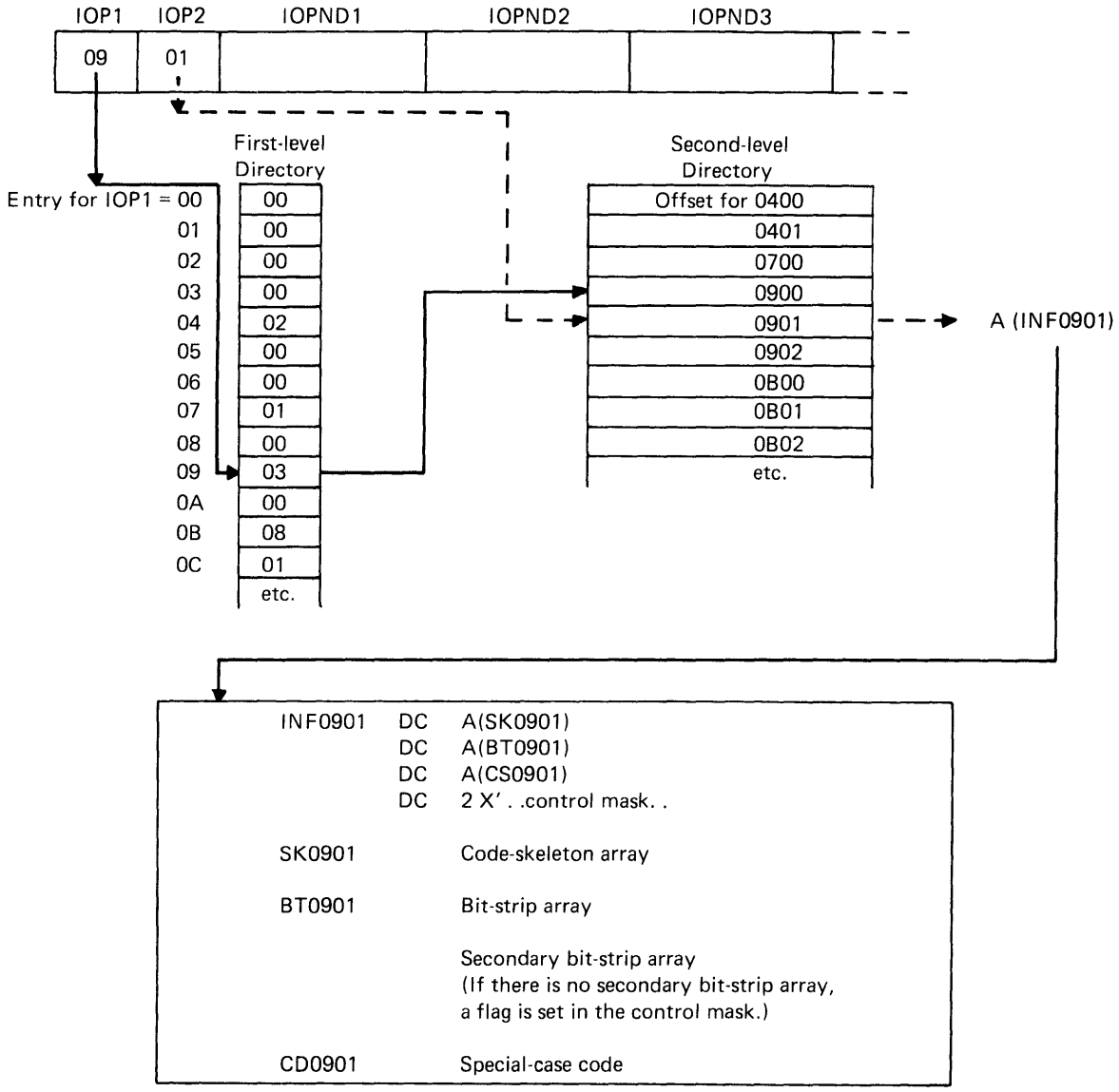


Figure 2.30. Use of directories to locate code-generation information

Between them, the directories for SA and SQ contain entries for every legal combination of IOP1 and IOP2. Obviously there are entries for the text tables which these phases will convert to extended code; in addition there are entries for the text tables which SC and SD will process. These latter entries simply direct the flow of control to a routine which sets the appropriate bit in XOPPHS1 to indicate the need to load either SC or SD. The text table is then transferred to the output stream, preceded by the 'untranslated' text table marker.

Standard Information in Text Table Area

The levels in the level two directory indicate an area in the phase where information regarding a particular text table will be found. The first 14 bytes of this information are as follows:

A(code skeleton array)
A(bit strip array)

A(any special case code)
2 bytes information mask

If any of the above addresses are 0, it indicates that the item does not exist. For instance, a number of tables do not produce code and so there will be no skeleton or bit array. Further, this scheme allows text tables to share a skeleton or special case code where appropriate.

Code-skeleton Arrays

The information area for each type of text table processed by a phase contains an array of code skeletons. Each code skeleton is the mnemonic representation of an object-code instruction; typical code skeletons are shown below:

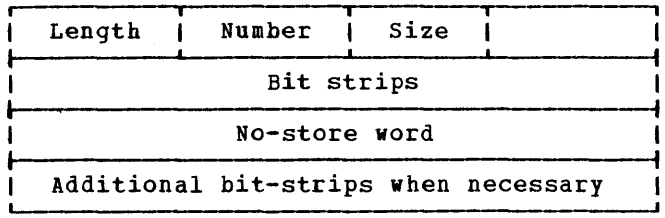
```
L,R3,operand 3
NI,01,operand 1
LM,R1,R3,operand 2
SLL,R2,8
```

The array contains a code skeleton for each instruction that may be required in the object module to represent the text table. The contents of various fields in the text table control which particular code skeletons are required. Selection is made by use of a bit vector selected from the bit-strip array for the text table.

For some text tables, the code skeleton array contains skeleton instructions with the mnenomic 0F. There is no machine instruction corresponding to this mnemonic; it indicates that the code skeleton consists of the hexadecimal representation of object code which is to be copied directly to the output stream. For processing purposes, code skeletons of this type are handled as pseudo RR instructions.

Bit-strip Arrays

The information for each text table type indicates a dummy section, BTOP, which contains an array of bit strips from which one is selected. The selected bit strip is used as a vector to indicate the instructions in the code skeleton array that are to be generated. The general layout of a bit strip array, which is word aligned, is shown below:



Length = length of a bit-strip in bytes (1, 2, or 4).

Number = number of bit strips in the array (4, 8, or 16).

Size = number * length

Each bit that is set in a bit strip indicates an instruction in the code skeleton array that is to be generated. The bit-strip array contains a bit strip for each possible combination of instructions required by the text table. The required bit strip is selected by examining the status byte, IST1, in the text table. Phase QA sets bits in this byte to indicate the status of the operands (whether they are currently held in

registers or not) Bits 0 to 3 of IST1 are used to index the bit strip array.

Each bit in the No-store word corresponds to an instruction in the code skeleton array. Each bit that is set on indicates a store instruction. Bit 7 in IST1 is examined to see if store instructions are required, and whether the No-store word is to be used to modify the bit strip. If so, a sequence of logical operations is then performed with the selected bit strip and the no-store word, to eliminate selection of unnecessary store instructions.

For some text tables, an additional bit-strip array may be used to select additional code skeletons. The additional instructions are required by conditions indicated in the IST2 flag byte of the text table, set by various processing phases and sometimes modified by this phase. For example, additional instructions may be required according to the scale and precision of some operands. The last four bits of the IST2 byte are used to select the second bit strip in a manner similar to the selection of the first bit strip. The two bit strips are then merged by use of a logical OR operation. The resulting bit strip is used by the routine SHIFT to select the appropriate code skeletons in the code-skeleton array.

Special Case Coding

For some types of text table, special processing is required before code skeletons are selected. For example, before an MVC instruction is generated, the length of one or more operands may have to be determined. This processing is performed by compiler code located at an area named CDCP, following the bit-strip-array for the text table. Compiler code may also modify the contents of IST2 before it is used to select a secondary bit strip.

Extended-code Generation

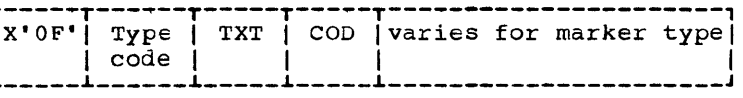
When a code-skeleton sequence has been selected from the array, routines and subroutines in the phase root module are used to generate the required output. The mnemonic code of each code skeleton is used in turn to index a 256-byte table. This table contains an entry for each mnemonic code that may be used. Each entry contains a value which, as shown below, indicates the basic format of the instruction code to be generated, and which is also a branch vector indicating the routine or subroutine to which control is to be passed for code generation.

Entry Value	Instruction Format	Routine or Subroutine
11	Branch RX instruction	BRRX
12	Branch RS instruction	BRRS
15	RS instruction using 2 registers	LMSTM
16	RS instruction using 1 register	RSSH
20	Ordinary RR instruction	RRR
01	Pseudo RR instruction	PSEUDO
21	Ordinary RX instruction	RXX
28	SI instruction	SI
29	LA instruction	RXX/BRRX
30	SS instruction containing 2 lengths	SS2L
31	SS instruction containing 1 length	SS1L

The selected routine or subroutine completes all operand, register, immediate and length fields required in the instruction. The subroutine BSDP is called to convert 6-byte operands from the text table into the

base and offset form required in the object code, accessing the storage dictionary as required. Absolute register values are obtained from the IREG1, IREG2, INDXA, and INDXB fields of the text table. Length and immediate values may be absolute values in the code skeleton, or may be calculated from information in the text table.

In addition to the markers inserted to indicate unprocessed text tables, other markers are inserted in the extended code, for use by the label resolution phase (Phase SK), the object module assembly phase (Phase SI), and the object code listing phase (Phase SM). The general format of a marker is as follows:



Bytes 0 1 2 3

X'0F' - identifies a marker.

Type - type of marker.

TXT - normally the length of the marker but, if the marker is associated with following generated code, TXT = length of marker + code.

COD - normally zero, but if the marker is associated with following generated code, COD = length of code.

For unprocessed-text-table markers, type = 0. These markers are deleted from the extended code by Phases SQ, SB and SC. Other markers remain in the code until deleted by Phases SI or SM, or until the end of compilation. The formats of the various markers used are shown in figure 5.125.

Identification of Returned VARYING CHAR Strings

To ensure that the library interface is aware when a returned string is VARYING, the compiler generates code in the prologue of any entry point returning VARYING. This code sets the bit in the returns string descriptor (passed by the calling program) to indicate VARYING. The library SORT interface routine may then interpret this bit.

This sequence of processing is initiated by Phase KT by setting on IST2 bit 6 in a MOVE text table if the returned string type is VARYING CHAR.

If IST2 bit 6 is on, then Phase SQ generates the following code:

```
L     R,x(R1)          Load return pointer
OI    6(R),X'80'       Set VARYING bit in return descriptor
ST    R,y(13)          Store return address in DSA
```

LABEL RESOLUTION (PHASE SK)

This phase examines the executable code generated by the compiler to reflect the source program instructions, and makes sequences of code individually addressable by allocating region numbers to sequences that occupy 4096 bytes of storage or less.

The allocation of region numbers makes it possible to assign an address to each label, thus enabling branching code to be generated. A table, known as the label table, is built, in which the addresses of all labels in the compilation are passed to the final assembly phase (Phase SI).

Whilst examining the code to determine region sizes, etc., this phase performs some final optimization functions, including removal of redundant instructions, modification of some inefficient instructions, removal of unreferenced labels, and modification of some register allocations.

This phase generates code to support the FLOW and COUNT compiler options, if they are specified. It also places the required compiler subroutines at the head of text.

PHASE INPUT

Input to the phase consists of the extended code output from Phase SQ, SD, or SC. This consists of a series of machine instructions, with various markers inserted in the stream.

PHASE OUTPUT

Output from the phase consists of the extended-code stream, with compiler-generated labels inserted in front of these items selected as region entry points. A label number, and the offset of the entry point from the start of the first program region, is inserted in each entry-point entry in the general dictionary. Each block-header entry in the general dictionary has the size of the code in the block inserted, and also the number of entries required in the statement-numbers table which is generated in response to the GOSTMT option. The label table created by this phase is output on one or more text pages. Absolute offsets of the regions from the start of the program are output on a further text page whose reference is passed to Phase SI in the XADCPG field in XCOMM.

PHASE OPERATION

Sequence of Processing

The XNLAB field of XCOMM is accessed to ascertain the total number of labels (user-supplied and compiler-generated) in the program. This value indicates the number of 8-byte entries that are required in the label table, to which an allowance is added for additional labels that may be generated by the phase. The appropriate number of pages is then acquired and initialized with zeros.

Part of the pseudo constants pool is scanned and, for those label constants that have an allocation in static storage, an entry is partly built (i.e., flags are set) at the appropriate offset in the label table pages.

Label constants appearing within branch-tables for optimized SELECT statements are processed in the same way by means of a scan of the associated constants pool, the text reference for which is held in the XBCBREF field of XCOMM.

The extended-code text stream is then scanned twice. During the first scan, when a new output text stream is built, the code is examined to see if there are any instructions that can be merged, or are redundant and can be deleted. Each branching instruction is examined, and storage is allowed for addressing code to be inserted according to the location of the branching instruction target. This enables the ultimate length of sequences of code to be estimated fairly accurately. The sequences of code are divided at suitable break points, and a region number is allocated to each code sequence, similar to the way in which base numbers are allocated to data storage areas. As each label is seen in this scan, an entry is made in the label table, and the appropriate region number is assigned to the label. A record is kept of the number of branching instructions transferring control to each label.

Two code counts are held in each entry in the label table during the first scan. The first count assumes that all forward branches are to labels within the current region. The second count assumes that they are not. At the end of each block a test is made to discover whether all of the branches found were to the current region. If not, a flag is set indicating that a second scan is required.

The second scan modifies the output from the first scan. All region boundaries have been established and the amount of code required at each branching can be calculated according to whether the target label is internal or external to the region containing the branching instruction. With this information available, the exact offset of each label from the origin of its containing region can be calculated. These offsets are entered in the label table and thus each label is made addressable. Any label that is not referenced is deleted from the text and from the label table. During this scan, the code may be further optimized, but this does not affect the region to which the code belongs.

The absolute offset of each region from the start of the program is inserted in another text page, the reference of which is passed to Phase SI in the XADCPG field in XCOMM. The entries in this text page are three bytes long, one entry per region.

Elimination of Redundant Instructions

During the first scan of the text, this phase searches for situations where translation of text tables into machine code has resulted in an inefficient sequence of code. An example of this situation is where a number of MVC instructions are used to move contiguous items of data. In such a situation it may be possible to use only one MVC instruction. Another example is where an item is loaded into a register which already contains the item. In such a case, the LOAD instruction is redundant and is deleted if global optimization is specified. Instructions are deleted by not copying them to the output stream during the first scan, and by overwriting them with X'0E' bytes during the second scan.

Establishment of Region Numbers and Boundaries

Region numbers are allocated to sequences of code, so that every label within the code sequence can be addressed as an offset from an identifiable base.

The markers inserted by the code generation phases are used to determine the code sequences to which region numbers are to be applied. A region number is applied to each block (PROCEDURE, BEGIN, and on-unit). The nominal size of a region is 4096 bytes. If the code contained in a program block occupies more than 4095 bytes of storage, a new region number is applied to the next labeled item. This ensures that no label has an offset greater than 4095 bytes from the start of the region. Some unlabeled instructions may have greater offsets but do not require addressing.

In calculating the size of a region and the position of its region boundary, allowance is made for a load of the base register in branching instructions. (Branching instructions are identified by markers inserted by the code generation phases.) A branch to a label within the same region as the branching instruction requires the normal four bytes of code. An unconditional branch to a label in another region requires an extra four bytes of code, and a conditional branch to a label in another region requires an extra eight bytes of code. When an instruction that indicates a branch backwards is seen, the allowance for addressing can be calculated exactly. When a forward branch is seen, it is not possible to determine the region in which the branched-to label will be contained, and the maximum allowance is made for addressing. However, a count is maintained (in the label table) of the number of forward branches to each label. The label counts are reset to zero whenever a new region boundary is established. If a branched-to label is reached before a new region boundary is established, the count enables the allowance for addressing to be adjusted.

Building the Label Table

An entry is made for each label existing or generated in twelve the code. All entries are eight bytes long and have the following format:

Byte No.	Content
0-1	Offset of label within a region.
2-3	Number of region containing the label.
4	Flags.
5	Forward-branch count.
6-7	Statement number of the label.

The flag bits are set as follows:

Bit 2	Label is not first with this code count.
Bit 3	Label is referenced in another statement.
Bit 4	Label is referenced in static.
Bit 5	Label is referenced.
Bit 6	Label has been met in scan.
Bit 7	Label is at the start of a region.

In the first scan, a label is generated and applied to each region's real entry point. The identifying number of the compiler-generated label for the apparent entry point is inserted in the general dictionary entry for the entry point, but is not required in the label table entry.

The label table entries are built during the first scan of the code. It is possible to make entries in the forward-branch-count field because label table entries are made in the same sequence as labels appear in the code. When the flag-bit-6 is set, actual branch-addressing allowances can be made instead of estimates. The offset values calculated during the first scan include any forward-branching addressing estimates.

The region boundaries are established during the first scan, but a second scan is made for the purpose of relocating the label offsets when accurate allowances can be made for branch addressing. The offsets are inserted in label table entries during this scan. The TAs of the label table text pages are passed to Phase SI in the XPHSCM field of XCOMM.

Insertion of Alignment Code

In the extended code input to the phase, a marker with a code byte of X'20' appears before any RX instruction with operands that require alignment. During the first scan of the text, the necessary alignment code is generated and the markers are removed. The amount of code generated depends upon the particular instruction. For example:

```
the instruction      LH   4,1(6)

becomes              MVC  WKSP+16(2),1(6)
                     LH   4,WKSP+16

the instruction      STD  2,FIELD(7)

becomes              STD  2,WKSP+16
                     ST   7,WKSP+32
                     LA   7,FIELD(7)
                     MVC  0(8,7),WKSP+16
                     L    7,WKSP+32
```

Implementation of the FLOW and COUNT Options

Two scans of text are always required if FLOW or COUNT has been specified.

During the first scan of the text, tests are made to ascertain whether the FLOW or COUNT compiler options have been specified. If so, code to call a library subroutine is generated at each branch-out or potential branch-in point. The code sequence generated is either:

```
    L    15,A..IBMBEFLB
    BALR 14,15
    DC   X'....'    (a halfword containing
                    the statement number)

 or L    15,A..IBMBEFLB
    BAL  14,0(0,15)
    DC   X'....'    (a halfword containing
                    the statement number)
```

The second example of code enables the library subroutine to cancel the branch-out of the statement when the COUNT option is in effect.

The first two bits of the statement-number constant are used as flags to indicate the following branching information:

```
    Bit 0   On:  branch-in point.      Off:  branch-out point.
    Bit 1   On:  change in current     Off:  no change in
                 block name.                 block name.
```

Flow code is not generated for a branch within a statement. If, at the time of the first scan, it is not known whether a branch is within a statement, a marker is generated before the flow code. During the second text scan the marker is removed, and the flow code is also removed if the branch is within a statement.

Flow code is generated for "branches" round procedures and on-blocks. These branches are not executed as blocks are denested, but flow code is generated as if the branches were executed at this time. The places at which such code is required is detected by finding gaps in the statement numbers in the statement header markers.

If there are no branches from one region to another, the second scan in SK is not needed, unless FLOW or COUNT have been specified. Two code counts are held in each entry in the label table during the first scan; an optimistic (assuming that all forward branches are to labels within the current region) and a pessimistic. At the end of each block a test is made as to whether any of the branches in it were to another region; if so a switch is set indicating that a second scan is required.

OBJECT MODULE ASSEMBLY (PHASE SI)

The prime function of this phase is to collect all the data and instructions processed by preceding phases, and to assemble them in the order required in the object module. The assembled object module is then transmitted to the SYSLNK and/or SYSPCH data sets (as specified in the compiler options) in records of the appropriate length. If output to SYSLNK is required, this phase opens the data set and allocates part of its working storage as a buffer area.

This phase also organizes data and instructions in such a way that the listings phase (Phase SM) can produce any listings specified in the compiler options.

PHASE INPUT

Input to the phase consists of the stream of extended code (instructions and markers), output by Phase SK. In this stream, all the instructions are complete, except that code is required in branching instructions.

In addition to scanning the extended code stream, the phase accesses the label table and the text page containing the absolute offset of each region from the start of the program (output from Phase SK), the pseudo constants pool (output from Phase PA), and the general, names, and storage dictionaries.

PHASE OUTPUT

The main output from the phase is a series of 80-byte (card-image format) records which together comprise a relocatable object module. The main content of the records is machine instructions, and external and/or internal addresses to be resolved by the linkage editor program. The records are referred to as TXT (text) records, RLD (relocation list dictionary) records, and ESD (external symbol dictionary) records.

The ESD records contain all the external symbols and external storage requirements for the object module. For example, they contain any symbols defined in this module that are referred to within another module. They also contain all symbols referred to within this module that are defined in another module.

The TXT records contain the machine instructions, and any constant data acted upon by those instructions, that are eventually loaded into main storage for execution. The code required to complete branching instructions is generated and inserted by this phase. TXT records are organized as a number of control sections. Each TXT record contains a field which identifies the control section in which the instructions and data contained in the record appear, and also their offset from the assembled origin of the control section.

An RLD entry is generated for each item that must be relocated by the linkage editor, such as relocatable address constants. Each RLD entry contains the offset of the item from the assembled origin of the control section in which it appears. An END record is generated to indicate the end of the object module. The END record can contain a transfer address, for use by the linkage editor.

If the DECK and/or LINK options have been specified, the object module records are copied into the XPFBF1, XPFBF2, and/or XOFBF buffers, and transmitted to the data sets assigned to SYSPCH and/or SYSLNK. Output to SYSPCH is in the form of 81-byte records (80 bytes plus 1 byte

control character). Output to SYSLNK is in the form of 322-byte records (four 80-byte records concatenated, and preceded by two bytes containing the number and length of the records).

Other output from the phase is passed to Phase SM on text pages. This output consists of:

1. The extended code stream as input, but with branching instructions completed.

2. An internal representation of the object module, which enables the external symbol table to be listed.

3. Information about items in the static CSECT.

4. The pseudo constants pool.

Each of the items listed above is output on a separate chain of text pages.

PHASE OPERATION

Sequence of Processing

Processing is organized so that records are output to build the object module in the sequence of ESD, TXT, RLD, and END records.

The ESD records are generated first. As ESD entries are built they are copied to the required output buffer(s), three entries per 80-byte record, and also copied onto output text pages, for use by Phase SM. Some ESD requirements are standard for each compilation, and the information about them exists within the phase. For other ESD entries, various fields in XCOMM are accessed to obtain information required in the entry or to determine the necessity for optional ESDs. Information for ESD entries for input/output modules and for external entries to the outer block is obtained from the general dictionary. Other information is found by scanning the pseudo constants pool.

TXT and RLD records are created next. TXT records are generated as required in various control sections. As these records are generated, RLD records are also generated where a situation requiring relocation is recognised. Both types of records are copied to the second stream of text pages (following the ESD records) but only the TXT records are copied to the output buffers at this time for transmission to SYSPCH and/or SYSLNK. When all the control sections have been built, the RLD records, saved in the text pages, are copied to the output buffers so that they appear in the object module after the TXT records. When the last RLD record is output, an END record is output to indicate the end of the object module.

The information required for generation of TXT and RLD records is obtained from various places. The PLISTART control section is required in every object module, and the information for it exists within the phase. Other control sections, e.g., PLIMAIN, PLIFLOW, are generated only if required by specified options. The need for these control sections is determined by examining various fields in XCOMM, but the information for the required records exists within the phase.

Information for building control sections for external items is obtained during a sequential scan of the pseudo constants pool, and the records are output in the order in which they are detected.

The address constants that appear in the static control section are found during a scan of the label table (output from SK), examination of

various fields in XCOMM, and a scan of the pseudo constants pool for items flagged as being external. During the building of the static CSECT, a check is made to see whether the GOSTMT option has been specified. If so, a scan is made of the extended code input stream and a statement-number table is built at the end of the static CSECT.

During the building of the static CSECT, another output stream is built on a third chain of text pages. This stream contains details of the first three classes of item for which entries are made in the static CSECT, and is built to enable Phase SM to list items in static storage if the MAP option is specified.

A second scan of the extended code input stream is made for the purpose of generating records to build the program control section. Where required, instructions are completed or additional instructions are generated.

When all records for the program CSECT have been output to the second stream of text pages, the RLD records and the END record are copied to the output buffers.

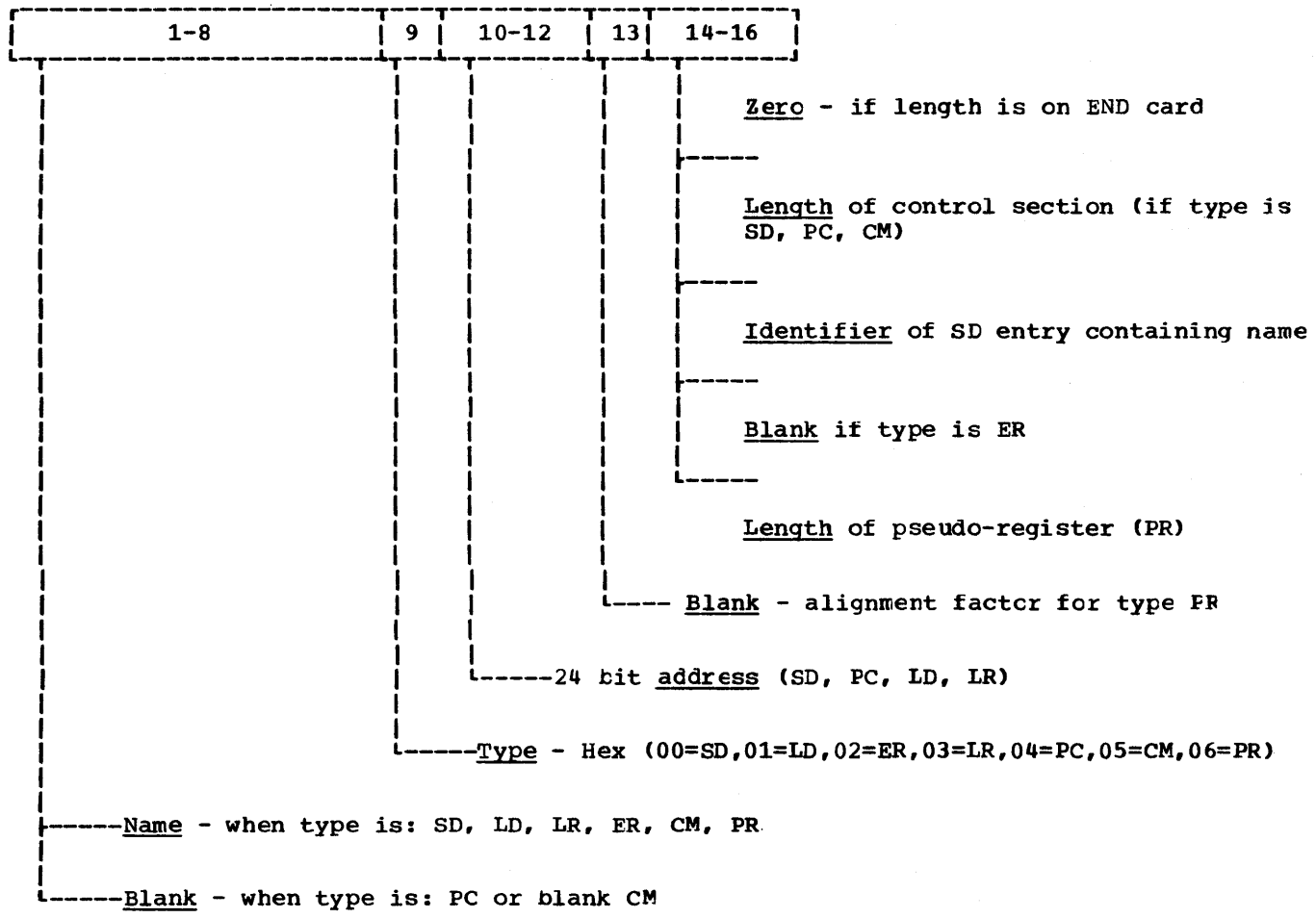
On completion of processing, control is passed to Phase SM if any of the object listing options have been specified. Otherwise, the XDIAG macro is used to pass control to Phase UA if there are any diagnostic messages to be printed, or to the compilation-end routine in Phase AA.

Generation of ESD Records

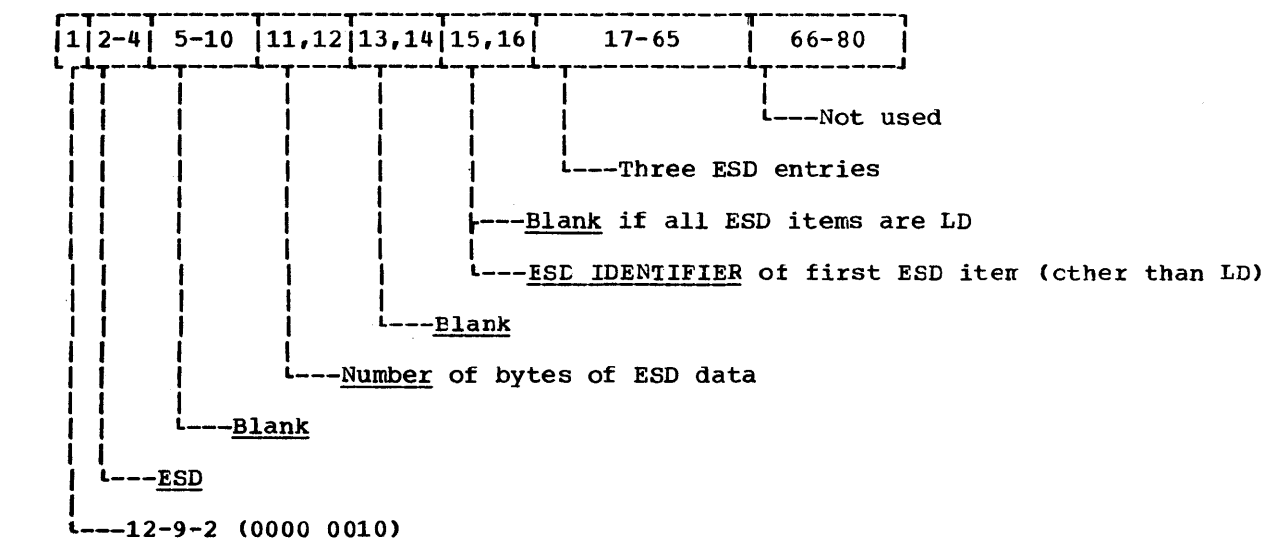
The five types of ESD entries that may be generated by the compiler are listed below.

ESD entry type	Purpose and contents
SD	Section definition: provides control section name, assembled origin, and length.
PC	Private code definition: provides assembled origin and length of an unnamed control section.
LD/LR	Label definition: specifies the assembled address and the associated SD of a label that may be referred to by another module. The LD entry is termed LR (label reference) when the entry is matched to an ER entry.
ER	External reference: specifies the location of a reference to another module.
CM	Common: indicates the amount of main storage to be reserved for common use by different phases.

The format of an ESD entry is as follows:



Each 80-byte output record contains three 16-byte ESD entries. The format of an ESD record is as follows:



ESD records are created in this format and output to the second chain of text pages, and to the output buffers for transmission to SYSLNK and/or SYSPCH.

The first ESD entry to be generated is an SD type entry defining the PLISTART control section. The entry is a standard one for all compilations, and the format is defined within the phase.

Entries are then generated to define the program and static control sections. The sizes of these CSECTS will have been stored in XCOMM by Phase SK. ESDs are then generated for PLIMAIN (if OPTIONS(MAIN) is specified), PLIFLOW (if the FLOW option is specified), PLICOUNT (if the COUNT option is specified), and any other CSECTS required by compiler or PL/I options.

ESD entries are also required for each compiler-generated subroutine or library routine that is called. The format of entries for these routines are defined within the phase. The requirement for them is determined by examining the bit strips in the XLIBSTR and XCOMSTR fields of XCOMM.

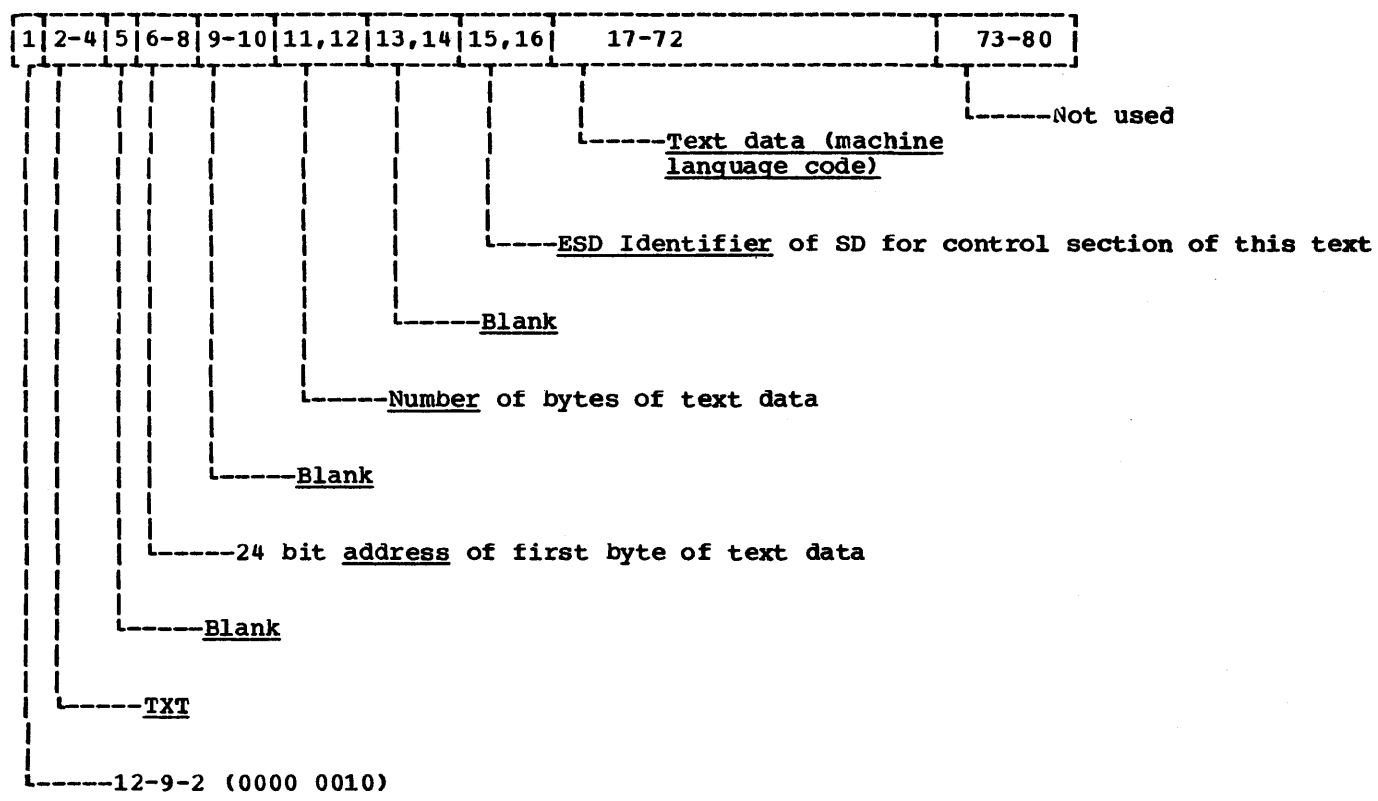
The need for ESD entries for LIOCS modules is determined by examining the chain of general dictionary entries headed by the entry in the XLIOCS field of XCOMM. The need for ESD entries for external entries to the outer block of the compilation is determined by accessing the XBH field of XCOMM to find the reference of the block header entry for the outer block. The entry-point entries for any external entries to this block are also accessed to provide information for ESD entries. In certain cases where an interlanguage communication option is specified on an entry point, the outer block will have a count of zero and the dictionary entries for entry points on the next block will have to be examined.

The storage dictionary is then scanned for other items requiring ESD entries, e.g., external variables, external symbol tables, files, external conditions, etc. The relevant entries in the storage dictionary are accessed to obtain the reference of the names dictionary entry for the item, and all the requisite ESD entries are generated. For STATIC EXTERNAL uninitialized variables appearing within an RLABL option for RPG, storage must not be reserved by PL/I since the storage for such variables is allocated within the RPG program. This is achieved by generating an ER-type ESD instead of a CM-type for the variables.

Generation of TXT and RLD Records

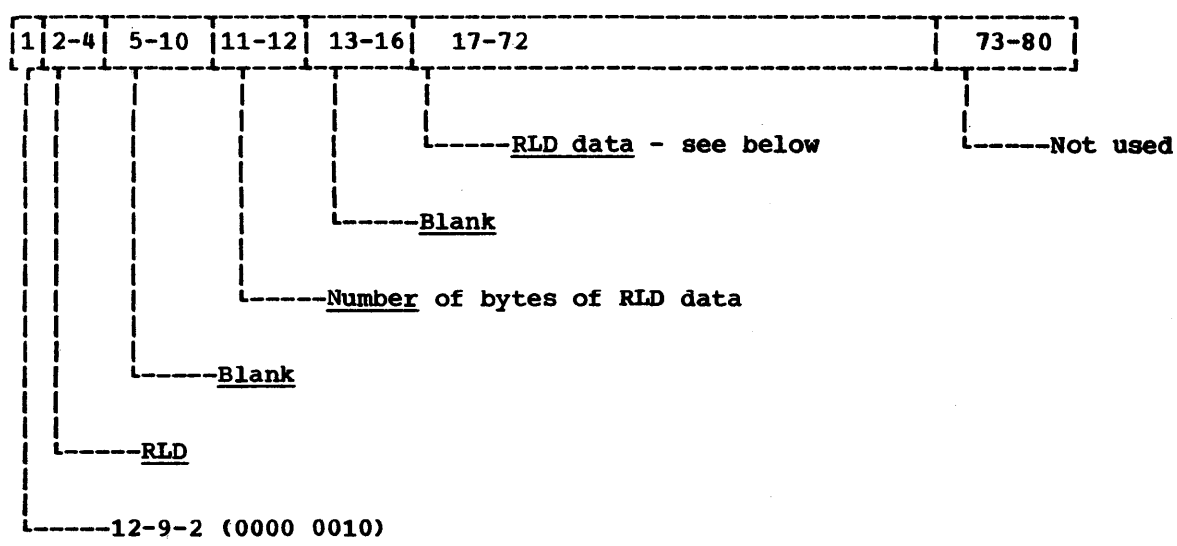
When all ESD records have been output, TXT and RLD records are generated as required.

The format of a TXT record is as follows:

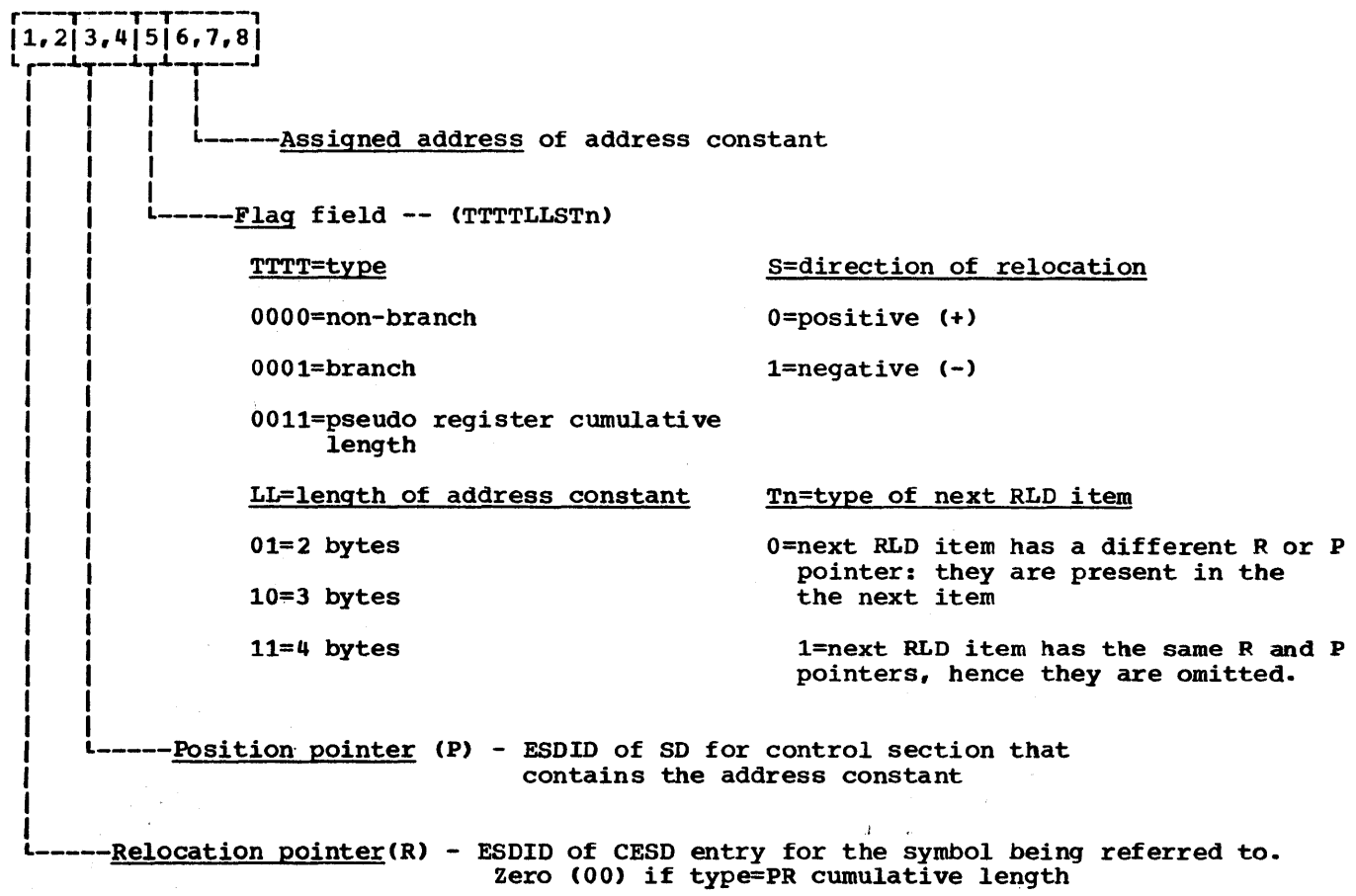


The number of instructions contained in one record varies according to the length of the instructions. All of the text data field (bytes 17-72) is used, and instructions may overflow to the next record.

The format of RLD records is as follows:



The number of RLD entries varies from record to record. The basic format of an RLD entry is as follows:



If the relocation pointer and position pointer of consecutive RLD entries have the same value, they are not repeated. Therefore entries may be eight or four bytes long.

The first control section for which TXT records are generated for all compilations is the PLISTART module. This module contains the address of PIR (program initialization routine) and code to branch to it, the address of SYSPRINT, and the address of PLIMAIN if a PLIMAIN control section is generated for the compilation.

Optional Control Sections: The XCOPT macro is used to determine the options that are specified and for which control sections must be built, e.g., PLIMAIN, PLIFLOW, etc. TXT and RLD records are generated as required, the information being contained within the phase.

The pseudo constants pool is then scanned for external items requiring control sections. These items include static external variables, static external symbol tables, external conditions, and external file control blocks. The code for each of these items exists within the pseudo constants pool, but a number of items within each control block may require modification, (usually the addition of a relocation factor, defined at the beginning of the pseudo constants pool). The TXT entries are generated in the order in which they are found during the scan, and RLD entries are generated when their requirement is detected. On finding the "initial" string item associated with a suitably declared PLIXOPT variable, this string is scanned and analysed at compile-time and an IBMBPOPT control section is generated for high-speed interpretation at program initialization on execution under DOS/CICS.

The Static Control Section: Records required for building the static control section are generated. The label table is scanned for items flagged as indicating the start of a program region, and entries containing their address constants are generated. If the size of static storage is greater than 4096 bytes, (the size of the static storage is inserted in XCOMM by Phase SK), address constants for the second and any subsequent regions are generated by this phase and appropriate entries are output. The bit strips in XLIBSTR and XCOMSTR are then scanned again to determine the compiler-generated subroutines and the library routines for which address constants are required. As these adcons are generated, the appropriate TXT and RLD records are output. In addition to copying these records to the second stream of text pages, all the records output thus far for the static control section are also copied to a third stream of text pages. This data is collected to enable Phase SM to list items in static storage if the MAP compiler option is specified. Dictionary references are retained in entries in this data stream.

The building of the static CSECT is continued during another scan of the pseudo constants pool (the second scan). Output from this scan forms the constants pool. TXT records are built for all items not flagged as external items. Records for external items will have been output already, except for external address constants, for which records are generated during this scan. Any items which require relocation are relocated at this time, even though RLD entries may be generated to enable further relocation by the linkage editor.

At this point a scan of the branch-tables label constants pool is incorporated in order to set up the tables required in the static CSECT for use by optimized SELECT statements.

If the GOSTMT option has been specified, a scan of the extended code input stream is then made, for the purpose of building the statement-number table in the static control section. This table and branch-tables for optimized SELECT statements are located immediately after the storage allocated for static variables. Whenever a statement number, or a statement-change marker is found during the scan, an entry containing the statement number and the address of the statement is built in the statement-number table.

Finally, if the TSTAMP option was specified at compiler installation time, phase SI will place the date and time of compilation at the end of the static CSECT (within the 20 bytes preceding the fullword location addressed by the first word of the CSECT). This is achieved by moving in the XCOMM fields XDATE and XTIME which have previously been set up by the control phase.

The Program Control Section: A second scan of the extended code input stream is made for the purpose of building the program control section. Records are output as the code is scanned. When a branching instruction is found, the label table is accessed and code to indicate the appropriate label offset is inserted to complete the instruction. Instructions required to load addresses before or after branching instructions are generated by this phase, and inserted in the code stream.

When the end of the extended code input stream is found, an END record is generated and output to the second chain of text pages. The RLD records in these text pages are then copied to the output buffers, so that they appear in the object module after the TXT records. The END record is then copied to the output buffers, to indicate the end of the object module.

OBJECT MODULE LISTINGS (PHASE SM)

This phase collects information about various items or areas of code in the object module, and passes this information to the SYSLST data set for printing in suitably formatted lists. The phase is only loaded and executed if one or more of certain mutually independent compiler options are specified. These compiler options, and the information listed if they are specified, are as follows:

AGGREGATE	Aggregate-length table.
STORAGE	Storage requirements for each CSECT or DSA.
ESD	List of external symbols dictionary entries.
MAP	Static internal storage map.
OFFSET	Tables of offsets and statement numbers.
LIST	Object code listing.

PHASE INPUT

The input data available for use by the phase includes:

- The extended code stream, output from Phase SI.
- The second text stream output from Phase SI, consisting of an internal representation of the object module.
- The third text stream output from Phase SI, containing information about items in the static control section.
- The text page containing the pseudo constants pool, output from Phase PA and modified by Phase SI.
- The names, variables, general, and storage dictionaries.

The particular input data used by the phase depends on which of the relevant compiler options has been specified, as detailed in the description of processing for each option, given below under the heading "Phase Operation."

PHASE OUTPUT

Output from the phase consists of information to the SYSLST data set to allow the printing of the listings appropriate to the specified compiler options. The content and format of these listings are described in the following descriptions of option processing.

PHASE OPERATION

The appropriate bit settings of the XNSYGBT field in XCOMM are checked to determine which of the compiler options requiring object listings have been specified. Where applicable, the following processing is performed.

Processing the AGGREGATE Option

The variables dictionary is scanned sequentially for array or structure variables. When a data aggregate entry is found, the names and general dictionaries are accessed to obtain the name and length of the found item. All the found data aggregate entries are chained together, by name, in alphabetic order, by means of a prescan of the names dictionary. This chain is then rescanned, and the aggregates listed against their sizes. For each adjustable aggregate found, the size of the item is replaced in the length table by a comment, and no name is printed for aggregate temporaries. The listing is completed by the printing of the sum of the lengths (in bytes) of fixed-size aggregates.

Processing the STORAGE Option

The storage requirement for each program procedure is obtained from the general dictionary procedure entries.

To obtain the names and lengths of the program and static control sections, the second text stream output from Phase SI is scanned and the information obtained from the first three entries of the external symbols dictionary. These entries, which make up the first ESD record, define the PLISTART, program, and static CSECTS. Full details of the composition of ESD entries are given in the description of Phase SI, earlier in this section.

Processing the ESD Option

To obtain the information required for the ESD listing, a scan is carried out of the second text stream output from Phase SI, and a list made from each ESD entry of the name (up to eight characters), a 2-character type identifier, a 3-byte address constant, and the lengths of all entries other than types LD, ER, and LR. Also included in the printed listing is a 2-byte identification number for each entry, generated by Phase SM.

Following the ESD listing, a single line of source program statistics is printed, giving the numbers of source records and program text statements (obtained from the XCOMM fields XNSREC and XTSTAT respectively), and the size, in bytes, of the object program.

Processing the MAP Option

The chain of text pages initialized by Phase SI and containing information about items in the static CSECT is scanned, and a list of adcons for library modules and compiler-generated subroutines produced on scratch pages.

The text page chain containing the pseudo constants pool is also scanned sequentially, and all static storage entries (constants, DEDs, descriptors, locators, labels, various adcons, DTFs, symbol tables, etc.) accessed.

The final static internal storage map listing comprises a 3-byte location counter for each item, followed by the item and its description, incorporating, where possible, the source program name.

A second scan of the pseudo constants pool is carried out if static external items (constants, DTFs, condition CSECTS, environment blocks, etc.) are encountered. These items are then listed separately.

The MAP option also produces a variable storage map. This map shows how PL/I data items are mapped in main storage. It names each PL/I identifier, its level, its offset (in both decimal and hexadecimal form) from the start of the storage area, its storage class, and the name of the PL/I block in which it is declared.

Processing the OFFSET Option

The table of offsets and statement numbers is produced by performing a scan of the extended code stream from Phase SI. In this stream, the start of each new statement is indicated by a marker, inserted by Phase SA. This marker will be one of X'0B' (statement number), X'1D' (statement number from where code came), and X'24' (statement number reverted to). Full details of these and all other extended-code markers will be found in section 5, "Data Area Layouts," figures 5.124 and 5.125.

When one of these three markers is encountered in the text scan, the current value of the location counter and the statement number are printed.

Processing the LIST Option

The main input text stream to the phase consists of a mixture of markers and generated code. The markers, inserted by Phase SA, consist of a code byte, X'0F', a qualifying byte, (one of X'00' through X'36'), and a length field (see figure 5.125). They are used to indicate such items as label numbers, dictionary references, statement numbers, etc., and the presence of a marker in the text stream will result in an appropriate comment being printed in the object listing. Each comment incorporates information carried within the marker, either directly or via a dictionary entry.

Following each generated code instruction in the input text stream are certain information bytes (3 bytes for RX and SI instructions, and 6 bytes for SS instructions). These bytes contain flags to denote that the address in the assembled instruction refers to a compiler-generated subroutine, a library module, decimal workspace, the base or length of a variable, a locator, a DED, or other qualifier. In addition, a dictionary reference may appear in the information bytes to enable the name of the variable to be decoded.

If no listing information is supplied, instructions may be preceded in the text stream by markers type X'10' or X'23'.

A sequential scan of the main text stream is carried out, and the object listing built up, normally in double-column format, in a one-page buffer in the phase storage area.

The format of the listing is so designed as to resemble as closely as possible programmer-coded assembler language. Each column contains:

- A 3-byte counter, which is incremented and printed at the start of each instruction.
- The assembled instruction in hexadecimal form, and printed with 2-2-1-3-1-3 half-byte spacings.

- The interpreted instruction, in assembler language, with operands consisting of register numbers, immediate values, storage addresses in base and offset form with optional index register or length fields, or identifier names, possibly qualified with an offset and preceded by qualifying prefixes such as VO, BASE, DED, etc.

When the end-of-program marker (X'14') is encountered, if no compiler diagnostics have been generated, a message to this effect is printed. Finally, the time taken for compilation is calculated and printed. Control is then passed to the resident control phase, Phase AA, and compilation terminated.

If compiler diagnostics do exist, control is passed to the diagnostic message-editor, Phase UA.

EDITING OF DIAGNOSTIC MESSAGES (PHASE UA)

Diagnostic message entries are generated on the error pages by the compiler phases by means of the XMESG macro. Phase UA receives these pages as input, and, by reference to various tables, produces the final (sorted) listing of messages. These messages appear in the compiler listing immediately following the cross-reference list, and are grouped according to their severity. The messages produced by the initialization stage will appear on the first page of the listing - not with the other messages produced by the error editor. Messages pertaining to the syntax checking of options will appear immediately following the printout of the corresponding *PROCESS card or parameter field. The specification of the FLAG option (described below) indicates that messages of only selected severity must be listed.

The major functions of Phase UA are:

1. To sort the message entries in the message page stream into severity-code order, statement-number order within each severity code, and, finally, generation-sequence order within each statement number.

2. To insert, and where necessary, to decode, the arguments supplied to the message.

3. To print out the message, with inserts, depending on the specification of the FLAG option.

Phase UA can be loaded and executed at different stages in the compiler operation, depending on the reason for its call.

Under normal conditions, with an object-module listing required, Phase UA is loaded, via the XDIAG macro and Phase AA (control phase), and executed following Phase SM (object-code listing phase).

If no object module listing is required and is so specified in the compiler option list, Phase UA is loaded and executed in the same way, following Phase SI (object module assembly phase).

If the NOCOMPILE option has been specified, (together with a value list: see "Implementation of Compiler Options," below) Phase UA will be loaded, via the XDIAG and XPST macros and Phase AA, following the dictionary build stage, provided that diagnostic messages have been generated by the syntax analysis and dictionary build stages. Compilation will then be suppressed or not, depending on the severity of the messages and the NOCOMPILE value list. If compilation is not suppressed, Phase UA will again be loaded and executed as the the final phase of the compiler (i.e., following Phase SI or Phase SM, as detailed above).

If an 'unrecoverable' error occurs at any point in the compiler, Phase UA is immediately loaded, via the XDIAG macro, Phase AA, and the error-handling routine, and executed.

If a program check interrupt occurs at any point in the compiler, Phase UA is again loaded, via the XSTOP macro and Phase AA, and executed, the diagnostic and error messages being listed following the abort dump, if such a dump has been specified.

PHASE INPUT

By means of the XMESG macro, a calling sequence is generated to a routine in XROUT. This routine uses a number of fields in the communication area (XCOMM) as follows:

XMPRF Contains the page reference of the current message page. This is zero if no message has been created.

XMREM Indicates the amount of space remaining in the current message page.

XSTAT Is a slot containing the number of the statement currently being processed by a phase.

XMNUM Contains an optional numeric argument for a message if one is specified as an argument to the XMESG macro.

XMDRF Contains an optional dictionary reference which is to appear in the message.

XMDTP Is a single byte used to contain a character indicating the type of dictionary to which XMDRF applies. Both the dictionary reference and the type code are specified as arguments to the XMESG macro.

In addition to the fields described above, registers RB and RC are also used if text is supplied as an argument to the message. Two forms of such text are permitted by the XMESG macro:

1. Implicit text pointers, where text is assumed to accupy the N bytes preceding the address in register 1. (The value of N is a constant but it may be changed in the XMESG macro definition.)

2. Explicit text pointers, where a pointer to the start of the text and the length of the text are supplied explicitly to the macro. These values are loaded into registers RB and RC respectively by the calling sequence generated.

The XMESG routine in XROUT generates a message entry (the format of which is described in the XMESGP DSECT) in a stream of pages chained from XMPRF in XCOMM. The routine calculates the size of the message entry to be made, and checks that the current message page contains sufficient space for the entry. If enough space is available, the page is brought into main storage, and the message entry is made by moving into the page the contents of all the XCOMM fields described above, plus the message text if the length is not zero. It should be noted that, although these fields are not necessarily used in the message, no attempt is made to determine this in XMESGR since testing would be more time-and-space consuming than moving the fields in every case. Whether or not the fields are actually used is determined by the structure of the message.

If the current message page does not contain enough space to accommodate the latest message, it is not brought into main storage and a new page is requested. The old page reference is then placed in the new page, and the new page reference is placed in the XCOMM field XMPRF. The message pages are thus chained in reverse order. The latest message is then entered on the new page in the manner described above.

PHASE OUTPUT

The output from Phase UA consists of diagnostic and compiler-error messages raised by error conditions occurring in all compiler phases with the exception of those forming the preprocessor stage.

The compiler-error message has the message number IEL0230I or IEL0970I if severe errors are detected and the format

$ COMPILER ERROR NUMBER n DURING PHASE pp.

This message is described in detail in appendix B to this publication.

The diagnostic messages output by Phase UA can be identified with the phase during the execution of which the appropriate error condition occurred by means of the message number as described in section 6, figure 6.1.

TABLES USED BY THE DIAGNOSTIC-MESSAGE EDITOR PHASE

The tables used in the operation of Phase UA are all produced by the XMTAB macro and are as follows:

MCDE: The table is generated by the XMTAB macro, when XMTAB is called with MCDE as an argument, and consists of 1000 single-byte entries, one for each possible message. Each byte consists of a set of flags indicating the severity code, and information concerning parameters to the message and editing of it.

The MCDE table is used to set the severity code when building up the sort units. Its use avoids the necessity of specifying the severity code and the statement-number information both in the message text and in the macro call to XMESG, which creates the entry in the error message page stream.

KEYREF: The KEYREF table is produced by the XMTAB macro when called with the argument MESSAGE. XMTAB produces the table by a nested call to the XMCDE macro, with COUNT as a second argument.

KEYREF is used to obtain the appropriate keyword from the keyword table (KEYTAB) using the code obtained from the scan of the coded message string (refer to the the sub-heading "Message Decoding" in the description of the phase operation given below).

The table consists of 16 pairs of 2-byte entries, one pair for each permissible length of words contained in messages. Thus the first entry is for 1-letter words (and special and parameter keywords), the second for 2-letter words, and so on. The first number of the pair is the number, in bytes, counting from the start of the keyword table (KEYTAB), of the last word of that number of letters. For example, if there are 12 single-letter words and 10 2-letter words, the first members of the first two entries in KEYREF are 12 and 22 respectively. The second member of each pair is the offset, in bytes, of the first word with the next highest number of letters from the start of KEYTAB. Thus, considering again the previous example, the second members of the first two entries in KEYREF would contain 12 and 32 respectively.

KEYTAB: The KEYTAB table is produced, with KEYREF, by a call to the XMTAB macro.

It consists of one DC instruction for each keyword which may appear in a message. The keywords are arranged in the table in order of length, shortest first, and for ease of updating they are arranged in alphabetic order within each length category. The order of the appearance of the keyword in the table determines the code that is assigned to the word in the coded form of messages. However, this fact is relatively unimportant since the keyword table and the message coding operation are both carried out by the XMCDE macro; the order in the table depends on the order of the arguments to the XKEY macro calls that are nested within XMCDE.

MESREF: This table is created by the XMTAB macro with an argument MESSAGE. It is used to access the coded message for a particular message number.

MESREF consists of 1000 halfword constants, one for each message. If message text for a particular message has not yet been supplied to the XMTAB macro, the relative halfword constant has a value of -1; otherwise the constant is the relative address of the corresponding coded message from the start of the message table (MESTAB).

MESTAB: The message table is produced by the XMTAB macro with MESSAGE as an argument. The individual messages in the table are coded by the XMCDE macro calls nested within the XMTAB macro.

MESTAB consists of strings of coded error messages with one or more code bytes for each word in the message. Each message is preceded by a byte which specifies the length of the message. The coded strings appear in order of message number, purely for convenience of updating the macro that produces the table.

All the tables described above are interrelated, but they may be classified in two groups:

- Glossary of keywords (KEYREF and KEYTAB).
- Lists of messages (MCDE, MESREF, and MESTAB).

The interrelationships between the tables are generated automatically by the relevant macros. In order to add messages or translate then into other languages, two areas of the macros require change; the message list in the XMTAB macro and the keyword list in the XMCDE macro.

The Message List

The message list appears at the end of the XMTAB macro as a series of nested calls to the XMCDE macro in the form:

```
XMCDE N,S,W1,W2,W3,W4,.....Wx
```

where

N — is a decimal number which identifies the message.

S — is the severity code of the message. This may be T, S, E, W, or I for 'unrecoverable', 'severe', 'error', 'warning', or 'informatory'. (Note that this is the only place where the severity code is defined.)

W1-Wx — are the words of the message.

If a statement number is to be applied to the message, the character '$' is coded as an argument following the severity code field.

One of the following special control characters may also appear anywhere in the word list:

Z — signifies that the preceding and following words must be concatenated.

Q — signifies that the following word must be enclosed in quotation marks.

All the words in the word list must appear in the keyword list, either explicitly or without one of the endings ATION, ING, ED, ES, LY, E, S, and (S).

If required, certain messages can be generated with extra phrases, the wordings of which are set up by invocations to the XMCDE macro, with number parameters 41 through 45. The position of this wording in the

message is denoted by the appearance of one of the 'alternative text' markers A1 through A5 in the XMCDE invocation for the message. The appearance of the wording is requested by setting on the X'40' bit in XERFLGS at the time of invoking the XMESG macro.

The Keyword List

The keyword list appears in the XMCDE macro in the form:

```
.XLn XKEY C,W1,W2,W3,W4,......Wx
     AIF (&XK7).X2
     XKEY C,Wx+1,Wx+2,......
     AIF (&XK8).X2
```

n — is the number of letters in the keywords W1 to Wx, and 1≤n≤16.

C — is &C1 if n≤8 and &C1&C2 if>8.

Note: The assembler has a limitation of 128 characters (for DOS) per macro call. In order to overcome this restriction, if the total number of letters in the argument list exceeds two records (one record with the macro call and one continuation record) a second (unlabeled) call to the macro is made, separated by the statement AIF (&XK7).X2. The end of all keyword lists for a particular word length is terminated by the statement AIF (&XK8).X2

PHASE OPERATION

The diagnostic-message editor phase operates in two stages, to sort and decode the message.

The Message Sort

Sorting is achieved by a sequential scan of the chain of message pages. As each message is encountered, a 12-byte 'sort unit' is created, consisting of the severity code, the statement number, the message number, generation-sequence number, and the 5-byte text reference of the message. The information regarding the severity code, and whether or not a message contains a statement number, is obtained from the MCDE table that is generated as a by-product from the XMTAB macro.

As the sort units are generated, they are placed in a new page stream. When each page is full, or when the end of the input stream is reached, the page is sorted by means of the SCHELL sort routine before the next output page is obtained.

On completion, the new output stream is rescanned and the sorted pages merged into new pages, the old pages being discarded as they are emptied.

At the end of the message sort process, two page streams exist: the original message stream and a stream of pages containing sort units in their correct sequence.

Message Decoding

When the message sort is complete, the sort-unit page stream is scanned sequentially. As each sort unit is encountered, the page reference of the corresponding message is obtained and converted to an address. Each message is maintained in the diagnostic-message editor in the form of a string of code bytes generated by the XMTAB macro. The appropriate message string is obtained from the table by indexing the message reference table (MESREF) with the message number.

The message string is scanned and each byte or group of bytes is converted to a keyword code, which is used to reference the table of keywords (KEYTAB) by means of the keyword reference table (KEYREF). The keyword may be one of three types:

1. Ordinary keyword. In this case, the keyword is moved directly into the print buffer.

2. Special-purpose keyword. This type is used to indicate either that the preceding and following keywords are not to be separated by a blank, or that some multiple of 256 must be added to the following code to construct the keyword code.

3. Parameter keyword. If a keyword of this type is encountered, the appropriate parameter is obtained from the message entry, is decoded or translated as necessary, and then placed in the print buffer.

When the print buffer is full, or when the message is complete, the message is printed and the scan moves on to the next sort unit.

Message Editing Facilites

In addition to the facilities described above (i.e., statement number, dictionary entry, text, and numeric arguments), the following inserts can also be generated in a message:

- Two text parameters.
- A single attribute specification.
- Two attribute specifications.
- One attribute specification and a text parameter.

These inserts may be generated by their being passed as arguments to the XMESG macro. In all the cases listed above, a character must also be set as an argument in the severity-code field of the XMCDE macro. Thus, although the XMESG macro may generate a text string containing two text inserts for a message, the string will be decoded as one insert unless this character is specified in the XMCDE call for that message. The character used in this case is 'E', which must be concatenated with the severity code field. This sets a bit in the MCDE vector.

In order to pass these arguments, explicit pointers must be set up to indicate the start and length of a formatted text string. The string would start with a code byte indicating which of the above four cases is concerned. This would be followed by a single-byte attribute specification and/or the text to be included, preceded by its corresponding length specification, depending on which of the four combinations is being passed.

If an attribute specification is passed in this manner, the single byte is decoded by Phase UA and the corresponding attribute is moved into the print buffer.

A further editing facility accumulates several generations of the same message page and then prints it only once (i.e., instead of the message appearing in the listing N times for N different statement numbers, it is printed only once following a list of statement numbers to which that particular diagnostic applies). Once again, for this facility to be implemented at message-decoding time, the corresponding bit must be set in the MCDE vector. This is achieved by concatenating the character 'C' with the severity-code field argument to the XMCDE macro. This accumulating facility can be applied only to those messages generated by the XMESG macro and having the statement number as the only parameter.

When, by reference to the MCDE table, a message in the message page stream is found to be of the cumulative type, two sort units are created for it. The first is created in the normal way, with the statement number field as specified in the XMESG call. The second, however, is created with the normal statement number field set to zero, the actual statement number being saved elsewhere in the sort unit. An entry is then made in a push-down stack, which always contains the lowest statement for which a particular cumulative message is generated. In addition to the statement number, the entry also includes the message number and the page reference of the first 'zero-statement-number' sort unit for the message.

Each time a further generation of the same message is encountered in the page stream, another sort unit with a zero statement number is created and the push-down stack is then checked to see if the new statement number is lower than the previous entry for that message. If this is the case, the previous entry in the stack is replaced by the new number and another sort unit is created with the statement number in the normal statement number field.

When the whole of the message page stream has been scanned, therefore, for a cumulative message which has been generated for N statements there will exist:

- N sort units with zero-statement-number fields (the actual statement number being held in the sort units).
- An entry in the push-down stack which contains the lowest statement number for which the message was generated, and a reference to the start of the zero-statement-number sort units.
- A normal sort unit for the message, containing the lowest statement number.

It should be noted that, if the first generation of the message in the message page stream should happen not to be the lowest statement number for which the message is relevant, a redundant sort unit will be created. This would be ignored at message-decoding time.

During the message sort, all the zero-statement-number sort units are shifted to the beginning of the sort-unit stream, and the normal sort unit for that message is moved to the correct position for printing. When this normal sort unit is encountered at message-decoding and printing time, the stack entry is checked to see if the message has been printed already (i.e., if this is in fact one of the above-mentioned redundant sort units). If the message has not been printed, the zero-statement-number sort units are referenced and all printed out in the position where the lowest statement number would have been printed.

Implementation of Compiler Options

FLAG Option: Diagnostic messages produced during compilation are grouped in order of severity. The FLAG option specification indicates the minimum severity level for which diagnostic messages, if produced,

must be listed. The severity level is specified in the value list of the FLAG option, as follows:

FLAG(I) All diagnostic messages listed.

FLAG(W) All diagnostic messages except 'informatory' messages listed.

FLAG(E) All diagnostic messages except 'warning' and 'informatory' messages listed.

FLAG(S) Only 'severe' errors and 'unrecoverable' errors listed.

Note that the specification of FLAG is equivalent to FLAG(I). The severity levels are discussed fully in the Programmer's Guide for this compiler.

If, due to the specification of the FLAG option, any levels of messages are to be suppressed, suppression takes place early in the operation of the phase and no sort units are created for those levels of messages.

COMPILE or NOCOMPILE Option: Phase UA also implements the option

```
COMPILE|NOCOMPILE[ (W|E|S) ]
```

NOCOMPILE(S) (default option) specifies unconditional compilation following the syntax analysis and dictionary build stages unless 'severe' or 'unrecoverable' errors are encountered.

The specification of NOCOMPILE makes the compilation conditional upon errors detected during these two stages. The value list specifies suppression of compilation if an error equal to or exceeding the severity indicated by the value list is encountered during operation of the above two stages.

NOCOMPILE(W) Compilation suppressed when any 'warning' messages and errors are detected.

NOCOMPILE(E) Compilation suppressed when any errors or 'severe' errors are detected.

NOCOMPILE(S) Compilation suppressed when any 'severe' or 'unrecoverable' errors are detected.

If NOCOMPILE is specified, or the specified severity value is equalled or exceeded, Phase UA returns control to Phase AA.

LINK or NOLINK Opion: The option

```
LINK|NOLINK[ (W|E|S) ]
```

specifies that link editing will or will not be suppressed.

NOLINK(S) (default option) specifies that link-editing is to follow compilation unconditionally unless 'severe' or 'unrecoverable' errors are encountered.

The specification of NOLINK followed by a value list will suppress link-editing according to the severity level of messages produced during compilation. The NOLINK value list interpretation is as for that described above for the NOCOMPILE option.

THE COMPILER DUMP PHASE (PHASE AI)

Note: Uses of the facilities provided by this phase are described in section 6, "Diagnostic Aids."

Phase AI provides printed dumps of the contents of various areas of main storage. If dumps of the text and dictionary areas are required, the compiler page-handling routines are used to copy requisite pages into main storage so that they can be dumped.

The loading and execution of Phase AI is optional, depending upon options specified by the user in the *PROCESS statement. However, if the DUMP option is specified for any member of a batch other than the first it must also be specified for the first member. This is because Phase AE has Phase AI loaded, and it then remains loaded throughout compilation. Phase AI requires approximately 16K bytes of storage on the compiler partition. This phase can be executed in the case of an abnormal termination of the compilation, or can be executed after the execution of any specified phase. Dumps can be formatted or unformatted.

PHASE INPUT

Phase AI accesses the XDPOPT field (in XCOMM) which has values set by Phase AE to indicate the required features of a dump, as specified in the *PROCESS statement. It then scans the requisite areas of main storage. The compiler communication area is accessed as required during execution.

If dumps of text or dictionary are required, Phase AI has any required pages that have been spilled copied into main storage.

PHASE OUTPUT

Phase AI provides printed dumps of the hexadecimal values of the contents of specified areas of main storage. Areas that can be dumped include:

Registers
Compiler communication area (XCOMM)
Phase work area (XSTG)
Text
Dictionary
Page header information
Current error page

Options exist to enable the dumping of a particular range of statements or particular text pages. Similarly, particular dictionary sections can be specified.

For all dumps, headings are printed to show the type of dump (abort or interphase), and the compiler phase during or after which the dump was provided. Subheadings indicate each section of storage dumped.

If a formatted dump is specified, the following features are provided:

1. Text

    a) Page header tables are formatted.

b) Each statement or text table is formatted, with appropriate headers for most fields.

c) Flow unit headers are formatted.

2. Dictionary

a) Each dictionary section is identified.

b) Each dictionary entry is formatted, with appropriate headers for each field.

3. Working Storage

a) A particular area of working storage can be formatted. At the moment, this facility only applies to the register usage table built in XSTG by Phase QA.

PHASE OPERATION

On entry, Phase AI tests whether it has been called to provide a dump on abnormal termination of a phase or to provide an interphase dump. It then prints the appropriate heading.

The XDPOPT field in XCOMM is accessed to determine which of the dump options have been specified and which data areas are to be dumped. Various routines copy the required areas of main storage into the XPRINT buffers for printing. Subheadings are printed at appropriate places.

The compiler page-handling routines are used to copy text and dictionary pages from the spill file into main storage in the required sequence. The directory and normal dictionary-accessing routines are used to produce a sequential dump of each dictionary section. Text-scanning routines appropriate to the stage of compilation are used because of the different text types and chaining sequences.

FORMATTED DUMPS: If a formatted dump is specified, each item (e.g., a dictionary item or a single statement in text) is printed on a separate line and spaced out into its component fields. A heading is provided to identify these fields.

Formatting strings are used to space out the fields in each item. There is one formatting string for each type of item. The string is in the form:

```
X'aabbcc-----dd'
```

where

```
aa is the number of fields in the item
bb is (2*length-1) of the last field
cc is (2*length-1) of the next-to-last field
...
dd is (2*length-1) of the first field
```

For example if an 8-byte item consists of three fields, the first being four bytes long, the second being three bytes long, and the third being one byte long, its formatting string would be:

```
X'03010507'
```

If the 8-byte item is, say, X'0102030405060708', the above formatting string produces the following printout:

```
01020304 050607 08
```

Each item has associated with it a code byte, which is a multiple of 4 (for indexing purposes). In the case of the names, variables, and storage dictionaries, the code byte is the same as that of the dictionary concerned; this is possible because there is only one item type (i.e., dictionary entry format) for each of these dictionaries. The general dictionary, however, has several item types (the format of a constant entry, for instance, is different from that of an entry-point entry). Thus a table is used to translate general dictionary entry code bytes into item code bytes. Text items are assigned code bytes depending upon text type; on any particular invocation of the dumping routine, the required code byte is picked up by using the name of the current compiler phase and scanning down the table PHSTAB.

Adcons for titles and formatting strings are placed in their respective tables in item code byte order; hence the item code byte may be used to pick up the item's title and formatting string.

Formatted text dumps: The routine TXDUMP decides upon the format code byte appropriate to the type of text being handled. The PRSQTP routine uses this code to produce a formatted dump of one text page. It performs certain housekeeping (including the printing of page reference, location, page header and/or flow unit header) and then calls SQLTXT (if the text is sequential) or NOSQTP (if the text is non-sequential).

SQLTXT ensures that no printing is performed unless the text is within a statement range specified (if any). It employs the code byte to invoke a scanning and printing routine appropriate to the type of text being handled; the text is spaced into a print buffer, using the formatting string technique. The routine which performs this spacing is TXHERE (an entry point of FMTBUF).

NOSQTP uses the XNEXT macro to chain down text; TXHERE is invoked to format the entry in the print buffer.

Formatted dictionary dumps: Each dictionary page to be dumped is accessed by the DICDMP routine (as for unformatted dictionary dumps). The formatted printout is achieved by routine FORMDC, which basically contains one scanner for each dictionary. As the scan encounters a dictionary entry, the FMTBUF routine is invoked; this routine spaces out the entry (using the formatting string for the entry) into a print buffer.

SECTION 3: PROGRAM ORGANIZATION

INTRODUCTION

This section describes the organization of the compiler, and the structure of the component parts of the compiler.

At the beginning of the section are descriptions of the general organization of the compiler program; how it is divided into a number of phases which can be loaded individually and loaded. The basic sequence of phase loading is illustrated in a fold-out figure at appendix C, to enable cross-reference from any part of the manual. The conditions affecting the sequence of phase loading are shown in figure 3.1. General features of the structure of a phase are also described.

Following the general descriptions are descriptions of the physical organization of each phase, arranged in the basic sequence of phase loading. These descriptions are in tabulated form. Each table contains a list of the labels that identify the main sections of code in a phase, in the order in which they appear in a phase listing. Each label is accompanied by a type-key, which indicates whether a label identifies a control section, a dummy section, a routine, a subroutine, a routine or subroutine entry point, or a table, and a brief description of the function of that section of code. These descriptions are intended to provide a positional guide to areas of code in the phase listings which may require examination. With each list showing the physical organization of a phase there is a flowchart showing the logical organization of the phase.

BASIC ORGANIZATION OF THE COMPILER

The compiler consists of 52 phases. Each phase consists of a collection of System/360 Assembler Language instructions and definitions, which can be executed and used to perform one or more particular functions in the compilation process. The phases reside in the core-image library. Each phase is identified by a symbolic name, consisting of the characters "PLIO" followed by two uniquely-identifying alphabetic characters. Each phase can be loaded individually. For descriptive purposes, phases which perform related functions can be grouped into stages, but these stage names have no other function.

The first phase to be loaded when the compiler is invoked is Phase AA. This phase remains resident in main storage throughout all compilations. It provides services for the other phases, communicating with the system control program when the system facilities are required, e.g., for the loading of phases, input/output operations, etc. The remaining phases are loaded individually in sequence, so that only Phase AA and one other phase are in main storage at any one time. Each phase is executed individually, but all phases use the services provided by routines in Phase AA.

Not all the phases are loaded and executed for every compilation. Some phases are required each time, but the requirement for other phases is determined acording to the specification of certain compiler options or the inclusion of certain PL/I language features in the source program. In relation to the sequence of phase loading, phases can be divided into three general categories as follows:

1. Non-optional phases, i.e., those phases which perform functions mandatory to all compilations. This category includes Phases AA, AE, EA, EE, GA, GI, GE, GM, IA, ID, IE, II, KA, KT, KV, KK, KX, PC, PA, PE, PI, QI, QA, SA, SQ, SK, and SI.

2. Optional phases that perform functions for which the requirement is determined by the specification of certain compiler options, e.g., preprocessing, global optimization, some listing features, compiler

dumps, etc. This category includes Phases CA, BA, CE, IK, OA, OE, OI, OM, QE, SM, UA, and AI.

3. Optional phases that perform functions for which the requirement is determined by the inclusion of certain language features in the source program, e.g., built-in functions, stream or record oriented input/output statements, etc. This category includes Phases EI, IM, IQ, KE, KI, KL, KM, KQ, OC, OX, SD, and SC.

Note: A summary of conditions affecting the sequence of phase loading is shown in figure 3.2.

The resident control phase, Phase AA, is loaded whenever the compiler is invoked. The transient control phase, Phase AE, is loaded in response to a LOAD macro instruction generated by Phase AA. This instruction is generated automatically for the first or only member of a batch. On completion of the processing of a batch member, the compilation-end routine tests whether the XBATCH switch in XCOMM has been set, by either Phase AE or Phase EA, to indicate that there is another batch member to compile. If so, control is passed to the routine containing the LOAD macro instruction, so that Phase AE can be loaded to re-initialize the compiler partition prior to the next compilation.

During processing of the compiler options, Phase AE sets bits in the XNSYGBT field of XCOMM to indicate the compiler options that have been validly specified or applied by default. It then scans a skeleton list of phase identifiers in the XPHSL field of XCOMM, modifying identifiers if necessary (see "Facility for Testing Modified Phases" in section 6), and setting flags to indicate the phases after which a compiler dump is required in accordance with options specified.

If the DUMP option facilities are required for any member of a batch, the option must be specified before the first member of the batch. If the option is so specified, Phase AE will allocate storage accordingly and issue a LOAD macro instruction to have Phase AI loaded. Phase AI then remains in main storage throughout all compilations in the batch, and can be executed as required according to options specified on any batch member.

Apart from Phases AA and AI, each phase uses an XPST macro instruction to indicate to the phase-loading routine (in Phase AA) the identity of the next phase to be loaded. The last two characters of the phase name are used as an operand in the XPST instruction (refer to figure 3.1 "Phase Loading Operations"). An index number is also used in the instruction, and this enables the phase-leading routine to access the appropriate entry in the XPHASE field of XCOMM to ascertain whether certain options are applied to the phase.

In some cases, the next phase to be loaded is obligatory, e.g., Phase PA always specifies Phase PE as the next phase. In such a case, the phase contains the appropriate XPST instruction at the end of its processing instructions. In other cases there are alternatives in the sequence of phases, e.g., Phase KK can be followed by any one of phases OC, OX, or KX. In such a case, a phase performs a series of tests to determine which of the alternatives is required. If a following phase is an optional phase which is dependent upon whether one or more particular compiler options have been specified, one or more XCOPT macro instructions are used to check whether significant bits in XNSYGBT have been set. If a following phase is an optional phase which is dependent upon the presence of certain language features in the source program, the requirement for the phase may be indicated in one of two ways. Certain types of statement are chained together by Phase KA and linked from fields in XCOMM. The presence of a text reference in one of these fields indicates a requirement for the phase which processes statements in that chain, e.g., a value in XRIOCH indicates that Phase KQ is required. An XIF macro instruction is used to test whether a particular statement-type chain exists, indicating a requirement for a particular phase. The presence in the source program of other language features is indicated by the setting of appropriate bits in the XOPPHS1 field of XCOMM by various phases. The requirement for phases which subsequently process these language features is determined by the use of one or more XTM macro instructions to test for the setting of the appropriate bits in XOPPHS1. (Indications given in XOPPHS1 are shown in figure 3.3.) A phase may use a combination of XCOPT, XTOPT, XIF, and XTM instructions

to test the requirement for various alternative phases. These testing instructions are accompanied by branching instructions to or around related XPST instructions.

In response to an XPST instruction, the phase-loading routine first uses the index number to access and examine the entry in XPHASE for the current phase, to determine whether an interphase dump is required. If a dump is not required, or after a required dump has been provided, the entry in XPHASE for the phase indicated in the XPST instruction is further examined. If the facility for testing modified phases (see section 6) has been used, the entry indicates the modified phase name. If it is a valid phase name, any page-handling changes required in preparation for the new phase are carried out. A LOAD macro instruction is then issued, and the system control program loads the required phase into the phase area.

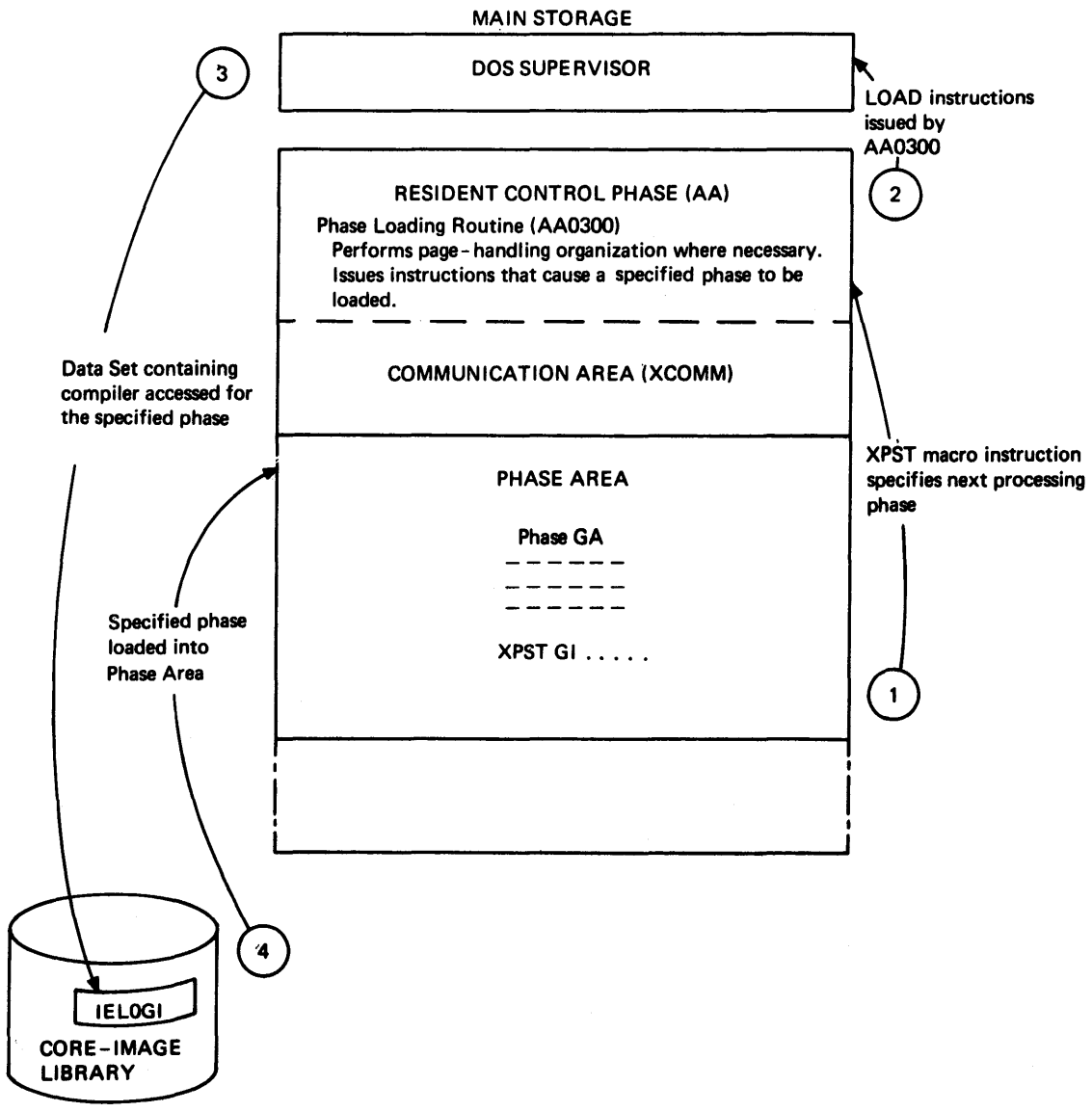


Figure 3.1. Phase loading operations

Loading of Diagnostic Stage Phases

In general, none of the processing phases is executed more than once during each compilation. Exceptions to this are the phases that produce diagnostic information, Phases UA and AI. The method used to indicate the requirement for Phase AI to produce interphase dumps has already been described. The DUMP option can also be used to specify that this phase is to be executed to provide a dump (compiler-abort dump) in the case of a program-check interrupt. In such a case, both Phase UA and Phase AI receive control via the interrupt-handling routine in Phase AA, which is called in response to the STXIT macro issued by Phase AE at initialization time. Any program-check interrupt that occurs during compiler operation causes control to be passed to the interrupt-handling routine. One form of program-check interrupt, the occurrence of an unrecoverable compiler error during execution of a processing phase, causes an XSTOP macro instruction to be invoked. This instruction contains a decimal-numeric operand which uniquely identifies the compiler error, and an appropriate error message is written onto the current error-message for later printing by Phase UA. The XSTOP instruction then terminates phase execution and, as with other program-check interrupts, control is passed to the interrupt-handling routine. This routine has Phase AI loaded to provide a compiler-abort dump if one is required by compiler options. It then has Phase UA loaded to print the compiler error message and any diagnostic messages generated prior to the interrupt.

In special circumstances, Phase AI can be called by use of the XDYDP macro instruction at any time during compilation. This facility can also be controlled by use of the DYSTMT option. Use of these facilities is described in section 6, "Diagnostic Aids." Other conditions affecting the use of Phase UA are described in the following paragraphs.

Effects of the NOSYNTAX, NOCOMPILE, and NOLINK Options on the Phase Loading Sequence

The NOSYNTAX and NOCOMPILE options can be used to control the number of compiler phases that are executed. If these options are used with a qualifying value, then the execution of certain phases is made conditional, dependent upon the severity of diagnostic errors discovered during the execution of preceding phases.

If the NOSYNTAX option is specified, compilation is terminated after execution of Phase CA, unless that phase generates any diagnostic messages, in which case Phase CE is also loaded and executed for the printing of those messages. If the SOURCE option is also specified, then Phase EA is loaded after Phase CA or CE, and partly executed for the printing of the source listing before compilation is terminated.

If the NOSYNTAX option is specified with a qualifying value, then the execution of phases subsequent to Phase CA (other than as mentioned above) is conditional, dependent upon whether any diagnostic messages with severity level greater than that of the qualifying value have been generated.

If the NOCOMPILE option is specified, compilation is terminated after execution of Phase GM, unless any diagnostic messages have been generated, in which case Phase UA is loaded and executed for the printing of those messages. If the NOCOMPILE option is specified with a qualifying value, then the execution of phases subsequent to Phase GM or UA is conditional, dependent upon whether any diagnostic messages with a severity level greater than that of the qualifying value have been generated. Note that if an attribute and cross-reference table is requested when the NOCOMPILE option is specified with a qualifying value, phase IK will be executed to provide the table regardless of where compilation is terminated.

Use of the NOLINK option does not restrict operation of the compiler, but inhibits link-editing of the object module produced by the compiler. Use of the NOLINK option with a qualifying value makes link-editing of the object module conditional, dependent upon whether any diagnostic messages have been generated during compilation with a severity level greater than that of the qualifying value.

Testing for the conditions mentioned in the preceding paragraphs is performed by use of the XDIAG macro. This macro is used in Phases CA, EA, GM, IK, SI, and SM, and in the interrupt-handling routine in Phase AA. Thus indications of the requirement for loading of Phases CE and UA are only generated via the XDIAG macro, which contains the appropriate XPST instructions.

Notes:
1. The tests for loading subsequent phases are applied in the order indicated. Hence, for a list of phases, the last is the 'fall through' case.
2. A phase can only be loaded by another when the loading phase has itself been loaded and executed, and when the specified conditions apply. Thus, with the exception of Phase UA, any one phase will only be loaded and executed once during a compilation.

Optional or not	Phase	Loads these phases if these conditions apply	Is loaded by these phases if these conditions apply
Not	AE	CA if MACRO option specified. BA if CHARSET(48) and/or CHARSET(BCD) and/or INCLUDE and not MACRO EA otherwise.	AA
Opt	CA	CE	AE if MACRO option specified.
Opt	BA	CE	AE if CHARSET(48) and/or CHARSET(BCD) and/or INCLUDE and nct MACRO.
Opt	CE	EA if SOURCE or NOSYNTAX(qualifier with qualifier not exceeded. AA otherwise	CA BA
Not	EA	AA if not SYNTAX. EE otherwise	AE if not MACRO and neither CHARSET(48), CHARSET(BCD), nor INCLUDE.
Not	EE	EI if source program contains stream I/O statements. (Decision internal to EE.) GA otherwise.	EA if SYNTAX.
Opt	EI	GA	EE if source program contains stream I/O statements (decision internal to EE).
Not	GA	GI	EE if source program does not contain stream I/O statements (decision internal to EE).
Not	GI	GE	GA
Not	GE	GM	GI

Figure 3.2. (Part 1 of 6). Conditions determining phase loading sequence

Optional or not	Phase	Loads these phases if these conditions apply	Is loaded by these phases if these conditions apply
Not	GM	UA (via XDIAG) if NOCOMPILE or NOCOMPILE(qualifier) and XMPRF ≠ 0. AA (automatically in XDIAG for UA) if XMPRF = 0 and NOCOMPILE or NOCOMPILE (qualifier) with severity or errors exceeding qualifier. IA otherwise.	GE
Not	IA	ID	GM if COMPILE and XMPRF = 0 or if XMPRF ≠ 0 and COMPILE or NOCOMPILE (qual.) with qualifier not exceeded. UA if bit 7 XOPPHS1 set and NOCOMPILE (qualifier) with qualifier not exceeded.
Not	ID	IE	IA
Not	IE	II	ID
Not	II	IK if XREF or ATR. KA otherwise.	
Opt	IK	KA	
Not	KA	IM if bit 0 XOPPHSI set. IQ if bit 1 XOPPHS1 set. KE if XSUBCH not empty. KI if XDOCH not empty. KT otherwise.	II if neither XREF nor ATR. IK otherwise.
Opt	IM	KA if bit 0 XOPPHS1 set. KE if XSUBCH not empty. KI if XDOCH not empty. KT otherwise.	KA if bit 0 XOPPSH1 set.

Figure 3.2. (Part 2 of 6). Conditions determining phase loading sequence

Optional or not	Phase	Loads these phases if these conditions apply	Is loaded by these phases if these conditions apply
Opt	IQ	KE if XSUBCH not empty. KI if XDOCH not empty. KT otherwise.	KA if bit 0 XOPPHS1 not set and bit 1 XOPPHS1 set. IM if bit 1 XOPPHS1 set.
Opt	KE	KI if XDOCH not empty. KT otherwise.	KA if bits 0 and 1 XOPPHS1 not set. and XSUBCH not empty. IM if bit 1 XOPPHS1 not set and XSUBCH not empty. IQ if XSUBCH not empty.
Opt	KI	KT	KA if bits 0 and 1 XOPPHS1 not set and XSUBCH empty and XDOCH empty. IM if bit 1 XOPPHS1 not set and XSUBCH empty and XDOCH not empty. IQ if XSUBCH empty and XDOCH not empty. KE if XDOCH not empty.
Not	KT	KL if either XFILCH or XRIOCH not empty. KQ if XSIOCH not null. KU otherwise.	KA if bits 0 and 1 XOPPHS1 not set and both XSUBCH and XDOCH empty. IM if bit 1 XOPPHS1 not set and both XSUBCH and XDOCH empty. IQ if both XSUBCH and XDOCH empty. KE if XDOCH empty. KI
Opt	KL	KM if XRIOCH not empty. KQ if XSIOCH not empty. KV otherwise.	KT if either XFILCH or XRIOCH not empty.

Figure 3.2. (Part 3 of 6). Conditions determining phase loading sequence

Optional or not	Phase	Loads these phases if these conditions apply	Is loaded by these phases if these conditions apply
Opt	KM	KQ if XSIOCH not empty. KV otherwise.	KL if XRIOCH not empty.
Opt	KQ	KV	KT if both XFILCH and XRIOCH empty and XSIOCH not null. KL if XRIOCH empty and XSIOCH not empty. KM if XSIOCH not empty.
Not	KV	OA if OPTIMIZE KK otherwise.	KT if both XFILCH and XRIOCH empty and XSIOCH null. KL if both XRIOCH and XSIOCH empty. KM if XSIOCH empty. KQ
Opt	OA	KK if program unsuitable for optimization. OE otherwise.	KV if OPTIMIZE.
Opt	OE	OI	OA when program suitable for optimization.
Opt	OI	OM	OE
Opt	OM	KK	OI
Not	KK	OC if bit 3 XOPPHS1 set. OX if bit 4 XOPPHS1 set. KK otherwise.	KV if NOOPTIMIZE OA if OPTIMIZE but program unsuitable for optimization. OM if OPTIMIZE and program suitable for optimization.
Opt	OC	OX	KK if bit 3 XOPPHS1 set.
Opt	OX	KX	KK if bit 3 XOPPHS1 not set and bit 4 XOPPHS1 set. OC

Figure 3.2. (Part 4 of 6). Conditions determining phase loading sequence

Optional or not	Phase	Loads these phases if these conditions apply	Is loaded by these phases if these conditions apply
Not	KX	PC	KK if bits 3 and 4 XOPPHS1 not set. OX
Not	PC	PA	KX
Not	PA	PE	PC
Not	PE	PI	PA
Not	PI	QI	PE
Not	QA	QE if OPTIMIZE. SA otherwise.	QI
Opt	QE	SA	QA if OPTIMIZE.
Not	SA	SQ	QA if NOOPTIMIZE. QE
Not	SQ	SC if bit 6 XOPPHS1 set. SK otherwise.	SA
Opt	SC	SK	SQ if bit 5 XOPPHS1 not set and bit 6 XOPPHS1 set.
Not	SK	SI	SQ if bits 5 and 6 XOPPHS1 not set. SC

Figure 3.2. (Part 5 of 6). Conditions determining phase loading sequence

Optional or not	Phase	Loads these phases if these conditions apply	Is loaded by these phases if these conditions apply
Not	SI	SM if at least one of MAP, EXTREF, STORAGE, CODE, AGGREGATE, and OFFSET. UA (via XDIAG) if XMPRF ≠0. AA otherwise.	SK
Opt	SM	UA (via XDIAG) if XMPRF ≠0 AA otherwise.	SI if at least one of MAP, ESD, STORAGE, LIST, AGGREGATE, and OFFSET.
Opt	UA	IA if bit 7 XOPPHS1 set and NOCOMPILE(qualifier) with qualifier not exceeded.	GM (via XDIAG) if NOCOMPILE or NOCOMPILE(qualifier) and XMPRF ≠ 0. SI if none of MAP, EXTREF, STORAGE, CODE, AGGREGATE, and OFFSET and (via XDIAG) if XMPRF ≠ 0. SM (via XDIAG) if XMPRF ≠ 0.

Figure 3.2. (Part 6 of 6). Conditions determining phase loading sequence

Note: Bits set to '1' if phase required, '0' otherwise.	
Bit	Remarks
X...	Phase IM required. Bit set by Phase GA if program contains: COBOL environment option. OPTIONS(COBOL or FORTRAN) on ENTRY statement or declaration.
.X..	Phase IQ required. Bit set by Phase KA if program contains: MAP table. CONCAT table. ALLOC table. FREE table. String arguments to functions. Bit set by Phase GE if: XAGHEDA or XAGHEDS not empty.
..X.	Not used.
...X	Phase OC required. Bit set by Phase KA if program contains: Any Compare. String assignment. Concat, And, Or, Not operators. Bool built-in functions. TRANSLATE. CONV.
.... X...	Phase OX required. Bit set by Phase KA if program contains: Complex built-in functions. String pseudovariable or function. REPEAT, HIGH, LOW built-in functions. Compare - string or complex. Conversions on strings. Bit set by Phase GA if program contains: Complex variables. Bit set by Phase GM if program contains: Complex constants.
.... ..X.	Phase SC required. Bit set by Phase SA if program contains: Strings.
....X	Phase IA required. Bit set by the XDIAG macro if: NOCOMPILE(qual.) parameter specified.

Figure 3.3. Indications of XOPPHS1 bit settings

BASIC STRUCTURE OF A PHASE

Each phase consists of one or more modules. A module is the smallest unit of the compiler that can be assembled individually, but a module cannot always be loaded and executed individually. This is only possible where a phase consists of a single module. The maximum size of a module is 28K-8 bytes, which is the size of the main storage area allocated as the phase area. Many modules (and phases) in the compiler are smaller than 28K bytes.

Where a phase consists of more than one module, the first module is referred to as the root module, and all other modules are referred to as non-root modules. The root module acquires all the working storage required by all modules in the phase. In almost all cases, all the modules that comprise a phase are loaded together. The exception is the compile-time preprocessor phase (Phase CA). For that phase, the root module and one non-root module are loaded together. The root module remains in main storage throughout execution of the phase, but overlay segments containing other non-root modules are loaded to overwrite the non-root module.

The general format of a compiler module is shown in figure 3.4. The order of some of the items shown varies from phase to phase. Most of the items shown are self explanatory, but references to some of the macros used in compiler module assembly are explained below.

1.	Comments describing the function of the module, (or phase if root module).
2.	Private DSECT definitions, with comments.
3.	General compiler DSECTS, invoked by COPY statements.
4.	COPY statements providing general and private EQU definitions.
5.	XBUG, setting the debugging level for the module.
6.	USING instructions for module addressability.
7.	XINIT, to provide necessary initialization code, and a USING instruction for macro and private storage.
8.	Code, consisting of macro invocations, assembler instructions, and comments.
9.	XROUT, to provide macro subroutines as required.
10.	XSTG, to provide storage required by macro subroutines.
11.	Private storage definitions required by the module (in the XSTG CSECT).
12.	XENDSTG, to delimit all storage.
13.	END.

Figure 3.4. General format of a compiler module

Item 8 in figure 3.4 refers to code consisting of macro invocations and assembler instructions. This is the area of code which performs the main functions of the phase. When the module is assembled, the invocation of many of the macros results in the generation of code inline. In cases where the function performed by the macro requires a large number of assembler language instructions, the minimum amount of code is generated inline, together with a call to a uniquely-labeled subroutine which contains code to perform the function. At the same time, a flag bit is set in a global bit variable (GBLB) uniquely associated with the macro. This operation is performed at each invocation of a macro.

Every module contains an invocation of the XROUT macro. The full expansion of this macro includes all the general subroutines called by other macros in the phase. The subroutine code is generated conditionally as a result of a series of tests made on the GBLB variables set by macros invoked in the general code. If a particular GBLB variable is set, XROUT invokes a subroutine macro within itself, which in turn generates the subroutine called by the macro in the general code area. In most cases, the macro, the subroutine macro, the subroutine itself, and the GBLB variable, have similar or identical

names. For example, one or more invocations of the XSRCH macro within a module (to search the dictionary for an entry identified by name) will set the GBLB variable &SRCH, and generate a call (or calls) to the XSRCH subroutine. When the XROUT macro is invoked, the assembly-time macro code generated will detect the setting of XSRCH and invoke the XSRCHR subroutine macro, which in turn will generate the XSRCH subroutine within XROUT.

The XSTG macro is invoked to allocate all storage required by macros, and read/write storage required by the phase. If the phase consists of a single module, the storage allocated by XSTG is a CSECT. If the phase consists of a root module and one or more non-root modules, the storage allocated by XSTG in the root module is a CSECT. Storage allocated by XSTG in a non-root module is a DSECT within the root module CSECT, its offset from the CSECT origin being specified in the OFS parameter. This enables the commoning of storage for macros invoked in more than one module within a phase.

Use of the XBUG macro is described in the DIAGNOSTIC AIDS section of this manual.

The module initialization macro, XINIT, is invoked once at the start of each phase. The format of the invoking statement is:

```
symbol | XINIT | PAGEHDR=
```

The optional keyword parameter PAGEHDR=, used for XOUT and XBREAK, specifies the address of the page header required to be put at the start of each page. The page header length is taken from the length specified for the constant or storage area given in this parameter. If the page header is not a Type-2 text table, this parameter is a 2-operand sublist, in which the first sub-operand specified the header and the second sub-operand is 'NOT' - e.g., PAGEHDR= (HEADER, NOT).

Other initialization automatically performed by XINIT includes dynamically filling in the start and end of storage addresses in appropriate fields in XCOMM, saving the page-header address and length, inserting the page size into the header, initializing the output stream if necessary, and initializing the trace information fields if operating in debugging mode. Many of these functions are performed by the XINIT subroutine generated by XROUT.

ORGANIZATION OF INDIVIDUAL COMPILER PHASES

The following pages contain flowcharts for the individual phases of the compiler. Each flowchart is accompanied by a table that indicates the physical organization of the phase. Each table contains a list of the labels appearing on the main sections of code. Beside each label name is a type key, and a brief description of the function of the section of code.

The type-keys, appearing in the column headed "Type", have the following meanings:

R = routine S = subroutine T = table E = entry point

Note: A number of phases contain code sequences that are only executed in the OS version of the compiler. This feature is indicated at appropriate places in the following tables.

Resident Control Phase (Phase AA)

Name	Type	Base registers	Function
IEL0AA	CSECT	R9	Initial entry point to the resident control phase (AA).
AA0000	R		Compilation initialization routine.
AA0204	R		Select spill algorithm on factor in XTEMP.
AA0208	R		Change text mask, and apply AA0204.
AA0212	R		Set text page count, and apply AA0204.
AA0216	R		Reset mask, and spill oldest text.
AA0220	R		Set dictionary page count only.
AA0224	R		Spill newest text.
AA0229	R		Apply factor to text and dictionary.
AA0300	R		Phase-loading routine, using AA0204 to AA0228 to select the optional page spill code.
AA0400	S		Scan the phase list in XCOMM for a specified phase.
AA0500	R		Compilation-end routine.
AA0600	R		Program interrupt-handling routine.
AA1000	S		Return control to a phase.
TTTT04	S		Debugging routine to dump all page headers.
AA4000	R		Page-handling routine.
AA4100	R		Access a dictionary page identified by its dictionary reference.
AA4200	R		Get a dictionary page into core, when the page is known not to be in core.
AA4500	R		Obtain a new directory page.
TACN00	S		Search a list of page chains for a specified page.
FPCN00	S		Search a list of page chains for the first page.
UDCN00	S		Transfer a page from an UNMOVABLE to a MOVABLE chain.
AA5700	S		Maintain the directory page chains.
PECN00	S		Add a page to the start or end of a chain.
AA6000	s		Spill file supervising subroutine.
STTA00	S		Save a re-usable track address.
AA7000	S		Provide a track address for a new page.
AA7500	S		I/O interface subroutine for the spill data set.
AA8000	R		Discard a list of text pages.
YTIME...	T		Constants, save slots, and 2600 byte page buffer area (XMBUF).
IJGWZRZU	CSECT	RF	LIOCS module for the spill data set.
XCOMM	CSECT	RD	Communication area.

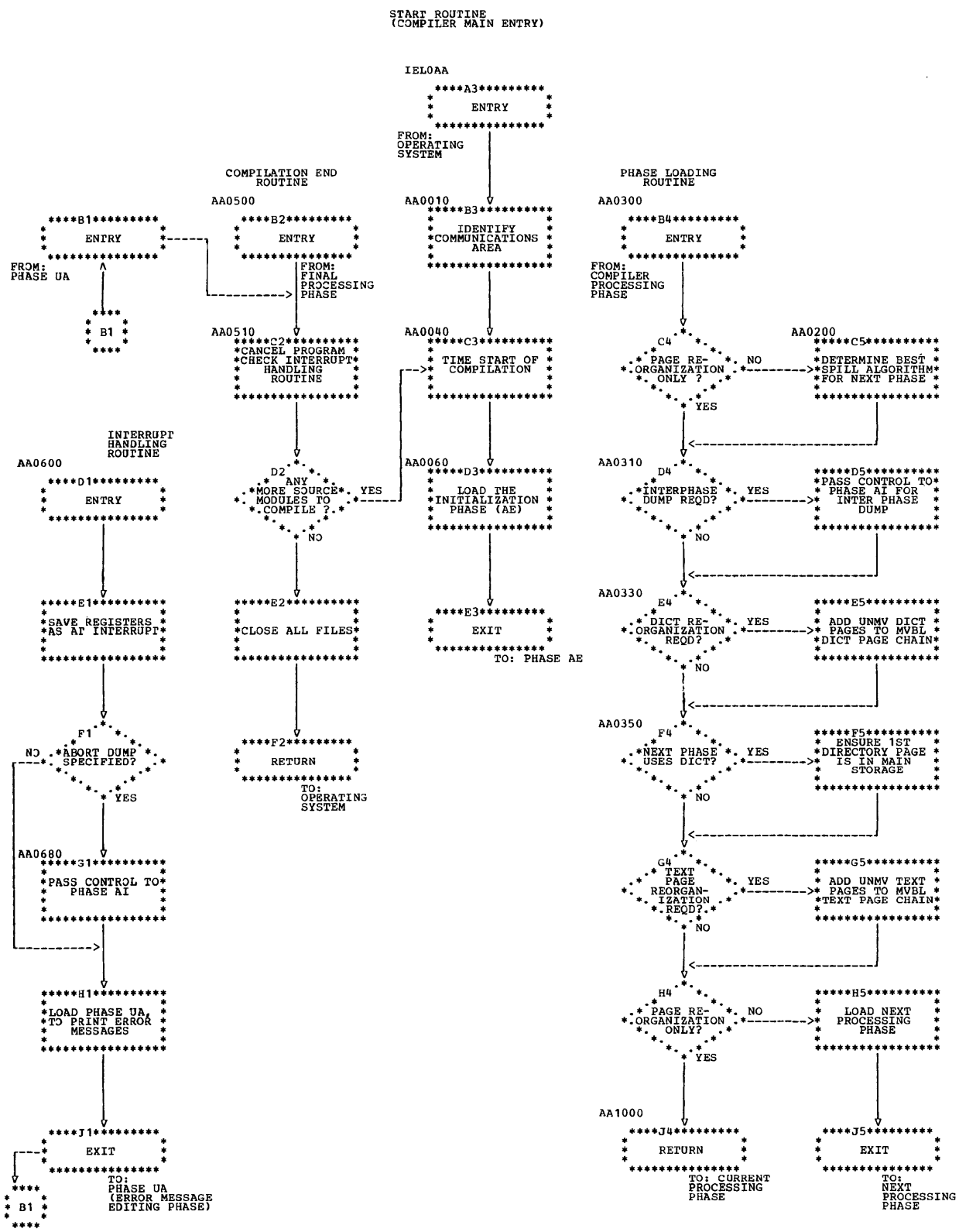


Chart 3.1. (Part 1 of 2). Resident Control Phase (Phase AA

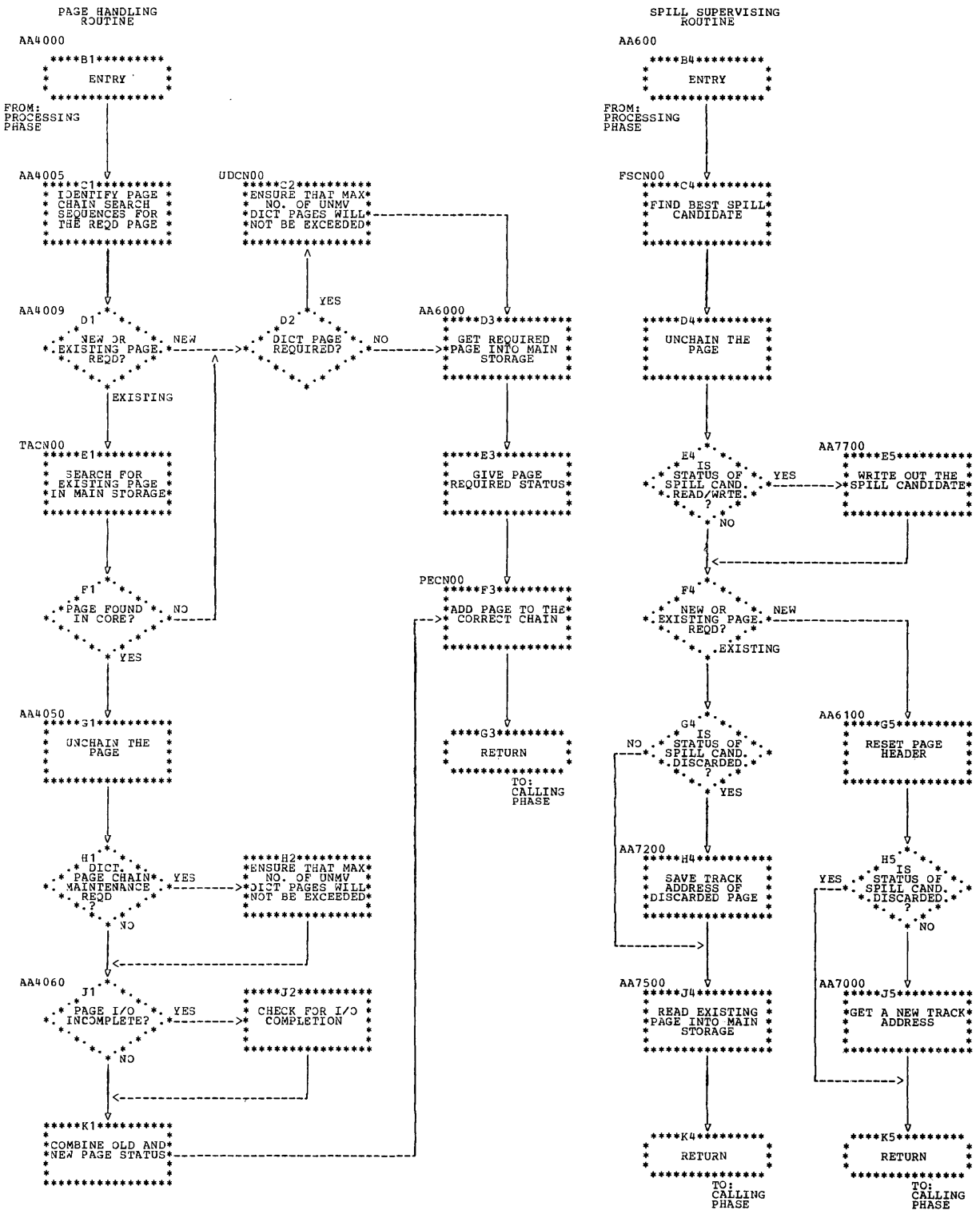


Chart 3.1. (Part 2 of 2). Resident Control Phase (Phase AA

Initialization Phase (Phase AE)

Name	Type	Base registers	Function
IEL0AE	CSECT	R8,R9, RB,RC	Entry point to Phase AE.
AE0010	R		Perform preliminary initialization of XCOMM, and register initialization.
AE2000	R		Open the input and print files, and initialize partition.
AE0040	R		Perform remaining initialization of XCOMM.
PROCOPS	R		Process the compiler options.
AE4000	R		Optionally open and initialize the punch and load files.
AE4260	R		Batch processing routine.
AE5000	R		Determine page space availability and hence page size.
AE5200	R		Initialize and open the spill data set.
AE5400	R		Initialize the page spaces.
AE5600	R		Initialize the dictionary tables in XCOMM.
AE6000	R		Pass information identifying the interrupt-handling routine to the operating system.
AE7000	R		Determine the next phase to be loaded and return control to the phase loading routine in AA.
			Subroutines used by PROCOPS:
ARGNUM	S		Calculate values of option arguments written in the form (A1,A2).
ERRSPC	S		Diagnose an incorrect option specification, and continue scan.
OPARG	S		Check parenthesis level for an option.
OPTAR	S		Process the optimization option.
SZEPR	S		Scan the SIZE option.
DMPSCN	S		Scan the DUMP option.
FNDICT	S		Analyze the dictionary type of a character string.
DYSTPR	S		Process the DYSTMT option.
XPRNTN	CSECT	R9	Print-file subroutine.
XPRNT	E		
IJJCPDON	CSECT	RF	LIOCS module for input and print data sets.
XSTG	CSECT	RA	Private storage for Phase AE.

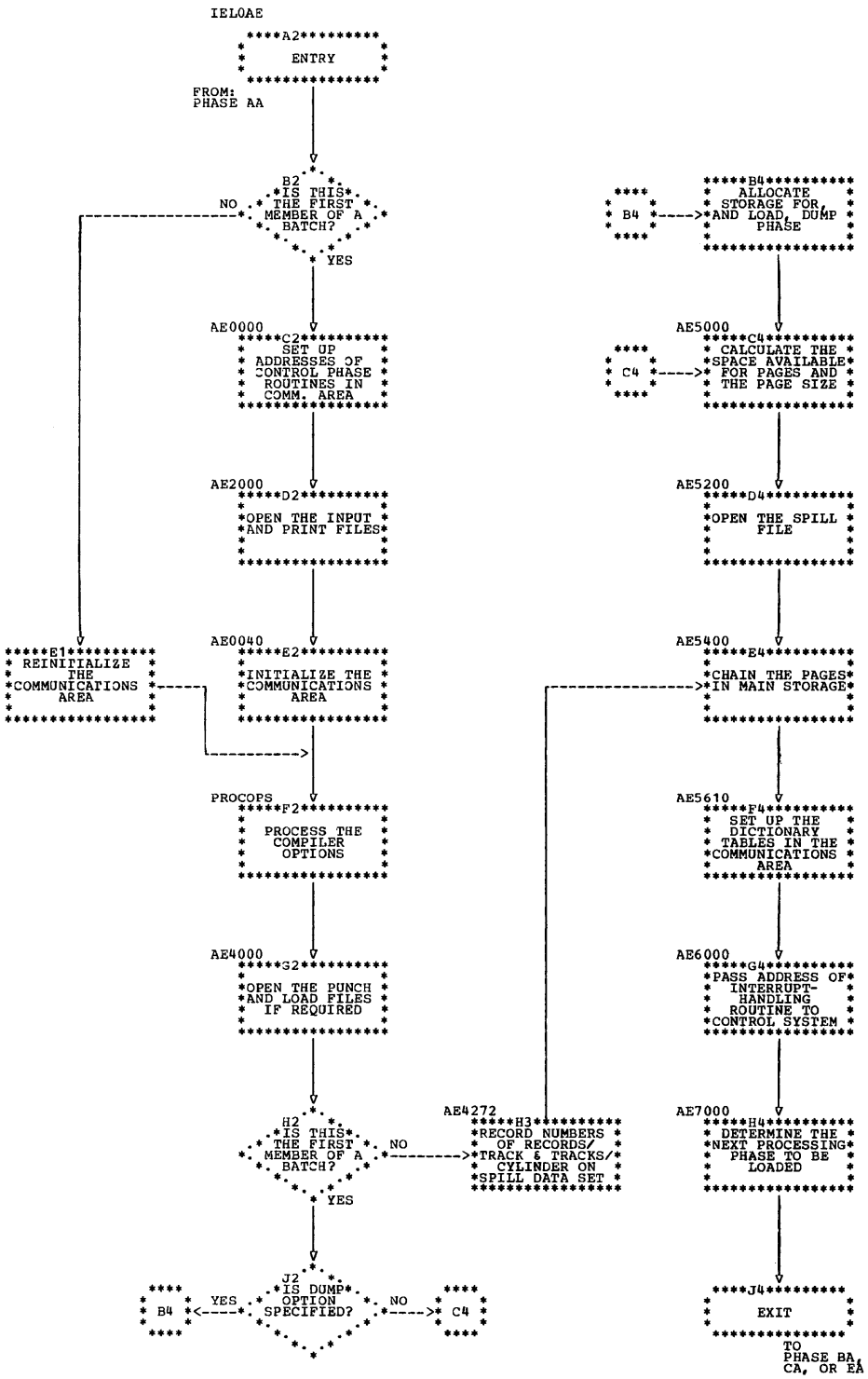


Chart 3.2. Initialization Phase (Phase AE)

48-Character Preprocessor Phase (Phase BA)

Name	Type	Base registers	Function
IEL0BA	CSECT	R8,R9	Entry point to Phase BA.
BA2020	R		Search for first valid character of 48-character symbol.
BA2140	R		Search for valid second character of 48-character
BA2220	R		)
BA2230	R		) Routines for checking characters following a valid
BA2240	R		) 48-character combination. Appropriate routine is
BA2250	R		) selected by TRT on result.
BA2260	R		)
BA3000	R		Decide where to continue scan of buffer if char pair not valid.
BA3100	R		After alphabetic characters which are not part of a 48-character symbol, look for delimiter at end of identifier.
BA4000	R		Replace 48-character symbol by 60-character version.
BA4100	R		Entered when 48-character symbol spans two records.
BA6000	R		Deals with quotes.
BA6040	S		Branched to from BA6000 if quote is in last byte of record.
BA7000	R		Deals with comments.
BA7040	S		Branched to from BA7000 if * is in last byte of record.
BA8004	R		Output 60-character record.
BA8010	R		Entry point for initial read. Read next 48-character record, translate to EBCDIC if necessary, output 48-character record.
BA9000	R		End phase processing.
TRT1	T		TRT table for valid first character, quote, or LCA.
TRT2	T		TRT table for valid second character, quote, or LCA.
TRT12	T		TRT table for valid first two characters of 48-character symbol.
TRTD	T		TRT table for valid delimiter.
TRT60	T		TRT table for 60-character symbol and associated information.
TRTQ	T		TRT table for quote only.
TRTC	T		TRT table for '*' at end of comment only.
TREBCDIC	T		BCD-to-EBCDIC translation table.
REP0	T		Table of 60-character symbols and associated lengths. Accessed by TRT60.
XINIT	CSECT		)
XMESGR	CSECT		)
XOUT	CSECT		)
XOPGE	CSECT		)
XMESGI	CSECT		) XROUT
MESGE	CSECT		)
XBREAK	CSECT		)
XBREAK2	CSECT		)
XOUT2	CSECT		)
XSTG	CSECT		Private storage for Phase BA.

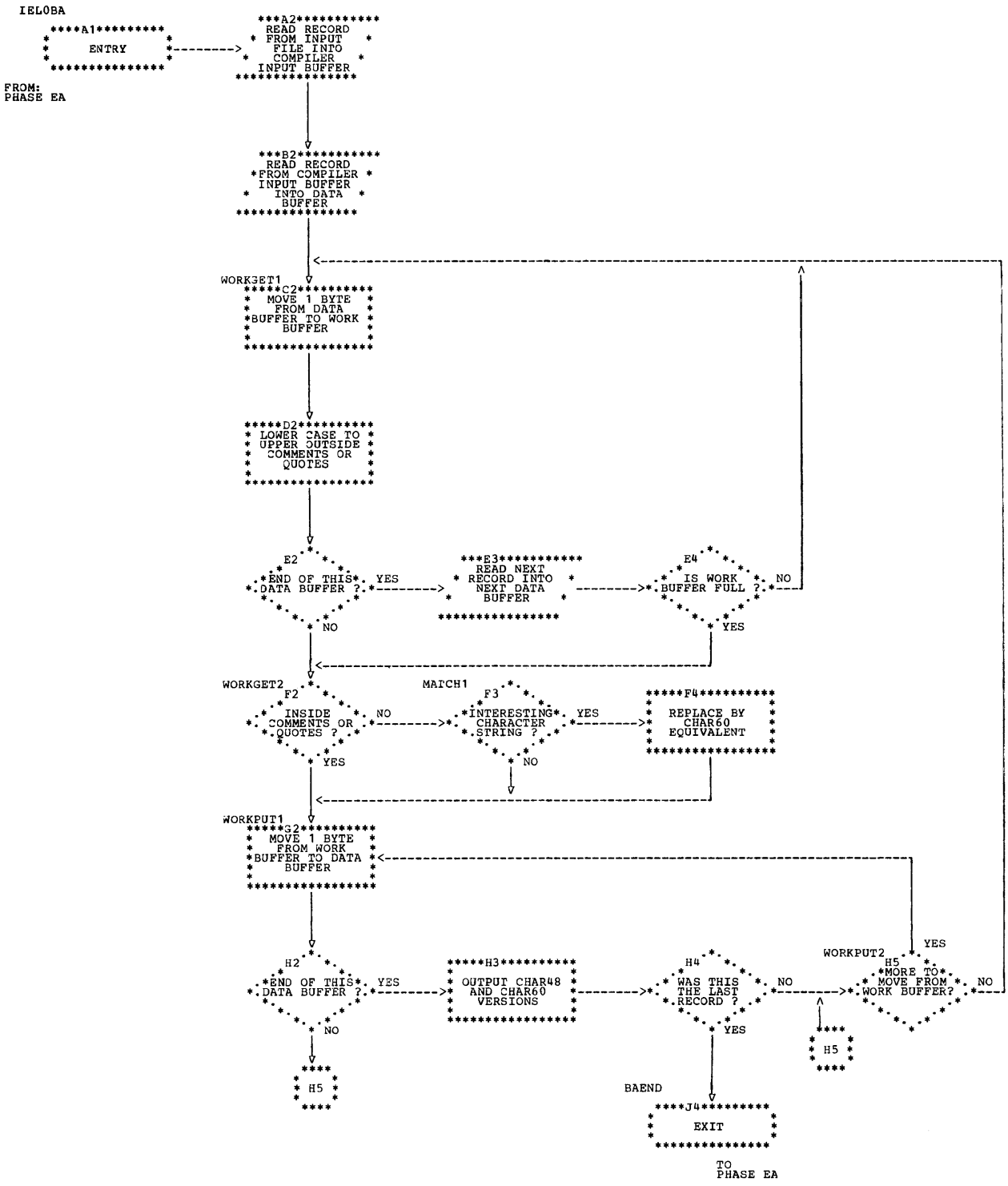


Chart 3.3. 48-character Preprocessor (Phase BA)

Name	Type	Base registers	Function
IEL0CA	CSECT	R9	Entry point to root module CA.
ROOTPH	S	R9	Load initialization module and pass on control.
INIT	R	R9	Perform interphase initialization and control. Load sub-phases CB and CC.
DISCTR	R	R9	Enter a TTR into a list maintained in XSTG.
BCKUP1	R	R9	Back the input cursor 1 byte.
BCKUP2	R	R9	Back the input cursor 2 bytes.
COMENT	R	R9	Read and print comments.
COMEND	R	R9	Read and delete comments.
NUNFRE	E		Entry point in routine GETIVB for when there are no free IVBs and a new one is generated in the preprocessor variables dictionary.
HASH	R	R9	Produce a 7-bit index which, when added to the base of the hash table, identifies the particular slot in the hash table to be searched for the EBCDIC name being hashed.
INPUT	R	R9	Read in an input record either from the source data set or from included text. Translate to EBCDIC if necessary and calculate the beginning and end of significant text. Set margin indicators if required via subroutine AGUT1.
UPNEWL	R	R9	Called when an update linecount character is encountered. Puts the new line count into XNEWLIN which is a temporary linecount buffer.
GNC	R	R9	Step up input cursor to point to the next character in a particular input stream.
GNCF	E		Special entry point for when routine GNC is called from routine FINDPC.
OUTPUTC	R	R9	Output a single character into one of the three output media: IVB, text block, or external record.
ENDIVB	E		Entry point in routine OUTPUTC used to close out an IVB chain and insert the end of IBV mark and count.
SRHDIC	R	R9	Search the dictionary for the presence of a named item.
TOKSCN	R	R9	Examine text, character by character, recognizing and returning each token. The routine handles the peculiarities of the 48-character set, the existence of line numbers in the text, and the existence of compressed blanks.
TKSCNS	E		Special entry point to routine TOKSCN.
STRING	R	R9	Scan the limits of a string constant and output according to entry points detailed below.
STRING	E		Scan and output the string unchanged.
STRNGD	E	R9	Delete one layer of quotes.
SKPSTR	E	R9	Skip over a string.
INRD	R	R9	Read from the %INCLUDE file one record at a time using the special DOS system macro DTFSL.
			Second section of Root Module CA.
LISTER	R	R8	Output source in lieu of the XBUF macro because the listing control statements %SKIP and %PAGE require that the line is printed after processing. This fact and the interpretation of %SKIP mean that a check has to be kept on the current number of lines in the page.
LISTER2	E	R8	Entry point to routine LISTER. Used when it is required to skip a given number of lines on the current page, or to leave a count which forces the next call to LISTER to start a new page.

Continued on next page

FREVAL	R	R8	Release a chain of IVBs containing a no longer needed value, and return the chain to a free-list.
GETIVB	R	R8	Remove an IVB from the free-list for use by the calling routine.
ASKEY	T		48-character keyword table used in routine TOKSCN.
XINIT	CSECT	R9	)
XMESGP	CSECT	R8	)
XMESGR	CSECT	RF	)
XRFSEQ	CSECT	R9	) XROUT - Held in root segment for common
XRFAB	CSECT	R9	) use by Root Module CA and
XDIREC	CSECT	R9	) sub-phases CB and CC.
XDSTAT	CSECT	R9	)
XPRNTN	CSECT	R9	)
XTXPG	CSECT	R9	)
XSTG	CSECT	RA	Private storage for root module CA.

Compile-time Statement Preprocessor (Sub-phase CA1)

Name	Type	Base registers	Function
IEL0CA1	CSECT	R8	Entry point to sub-phase CA1.
MTAB	T		Translate table.
WWEBCDIC	T		EBCDIC to BCD translation table.
WWBCD	T		BCD to EBCDIC translation table.
KY1TAB	T		Keyword table.
KY2TAB	T		Keyword pointer table.
XSTG	DSECT	RA	Private storage for sub-phase CA1 - storage in Root Module CA.

Name	Type	Base registers	Function
IEL0CB	CSECT	R8	Entry point to module CB.
PH1SCN	R	R8	Move text into text blocks until a compile-time statement is found and encode the statement into the instrument set.
PH1EOF	E		Entry point to routine PH1SCN for unexpected 'end-of-file' from other routines.
STMNT	E		Entry point to identify statement type, and build the label list.
11A2	R	R8	Process and stack THEN and ELSE clauses.
DONE	R	R8	Close out compile-time statements and destack THEN and ELSE clauses.
DONE2	S	R8	Test where to go on completion of the statement and return any non-empty label list.
CLSTHN	S	R8	Close THEN and ELSE clauses.
REFSTK	S	R8	Get dictionary entry for variable on the top of the push
DO	R	R8	DO statement processor.
DTEST	T		Contains the prototype for the XCODE to test for the end of the do-loop. Used by routine DO.
PARSE	R	R8	Check syntax of and generate code for compile-time expressions. The following tables are all used by this routine.
XACLSTR	T		Token class translation table.
XAOPRES	T		Precedence table.
XAOPCD	T		Operation XCODE table.
XATRAN2	T		Transfer table if preceded by operator.
XATRAN1	T		Transfer table if preceded by operand.
IEL0CB0	CSECT	R8	Second section of module CB.
INCLUD	R	R8	INCLUDE statement processor.
LABELS	R	R8	Process and check the LABEL list.
GOTO	R	R8	GOTO statement processor.
ACT	R	R8	ACTIVATE and DEACTIVATE statements processor.
ELSE	R	R8	ELSE statement processor.
ELSUB	S	R8	Locate the dictionary entry for the variable at the top of the push-down stack and insert the current value of the output pointer.
IF	R	R8	IF statement processor.
PAGE	R	R8	Handle the listing control markers %PAGE and %SKIP.
SKIP	R	R8	Insert special markers into the output stream so that on the second scan the same markers may be output, for the benefit of Phase EA. Also format INSOURCE.
CNTRL	R	R8	Insert special marker into text. No insource format action.
DOTEST	T		Contains the prototype for the XCODE to test for the end of the do-loop. Used by routine DO.
XSTG	DSECT	RA	Private storage for module CB - storage in Root Module CA.
IEL0CB1	CSECT	R8	Entry point to module CB1.
PUSHV	R		Move up to next slot on push-down stack and check if it is above the top.
POPV	R		Move down to next slot on push-down stack and check if it is below the bottom.
ADCONS	R		Obtain the dictionary reference of a constant, enter it into the dictionary if necessary.

Continued on next page

ADDSP	R		Add a preprocessor-generated item to the dictionary.
ADICT	R		Add a named item to the end of the appropriate hash chain and return the dictionary reference.
EOFCHK	S		Process undefined labels in the outer block of code.
PROCHK	S		Process undefined identifiers in a procedure.
DELETE	R		Skip over bad text up to the end of a statement, field, or procedure.
CMASCN	E		Entry point to routine DELETE, to find comma or semicolon and skip if comma.
SMISCN	E		Entry point to routine DELETE, to find semicolon.
PRSCN	E		Entry point to routine DELETE, to find end of procedure.
FINDPC	R		Scan source text, character by character, searching for a % character. Until this is found, characters are placed into text blocks.
RQRPC	E		Initial entry point to routine FINDPC.
EATAB	T		Type processing offset table. Used by routine FINDPC.
IDSRCH	R		Obtain the dictionary reference of an identifier, enter it into the dictionary if necessary.
KYWDSR	R		Determine whether the current token is one of a selected group of keywords.
OPNIVB	R		Set up output pointers to IVBs.
CLSIVB	S		Restore output pointers to original state.
OUTDIC	R		Put out code for compile-time action on variables.
TOKEN	R		Handle unwanted tokens for routine PH1SCN (module CB) and output diagnostics for tokens in error.
TKTAB	T		Token type and routine offset table. Used by routine TOKEN.
TRMIVB	R		Move text saved into the output stream.
RMVCHK	R		Remove a dictionary entry from the check list.
OPNLAB	R		Provide the next label on the LABEL list.
ATRCHK	R		Check for attribute confliction and consistency of use.
RDIVB	S		Save the current status and position of the scan pointer to allow IVBs to be read.
FINIVB	S		Restore current status and scan pointer position.
UPDLIN	R		Generate an update line count instruction.
ASSIGN	R		Assignment statement processor.
RETURN	R		RETURN statement processor.
XSTG	DSECT	RA	Private storage for module CB1 - storage in Root Module CA.
IEL0CB2	CSECT	R8	Entry point to module CB2.
TTTOP	R		Transfer to routine on XCODE value - used by DECLAR. routine.
DECLAR	R		DECLARE statement processor.
			The following routines are used in the DECLAR routine:
D1D3A	R		Left factoring parenthesis routine.
D1F3	R		Right factoring parenthesis routine.
D1G2	R		Checks validity of identifier.
D2C3A	R		Declared identifier routine.
D2C1	R		Comma routine.
D2K3	R		Stack maintenance routine.
D2H3	R		Builtin routine.
D2H4	R		Entry routine.
PROC	R		PROCEDURE statement processor.
END	R		END statement processor.
XSTG	DSECT	RA	Private storage for module CB2 - storage in Root Module CA.

Name	Type	Base registers	Function
IEL0CCD	CSECT	RF	This CSECT is inserted for DOS in order to branch to routine PH2SCN.
DOSDTF	CSECT	RD	This special macro DTFSL is included here at its logical place. However, the linkage editor is used to make the CSECT generated part of the next phase overlay. The function of DTFSL is to retrieve books from the source statement library.
IEL0CC	CSECT	R8	Entry point to module CC.
OUTPUT	R		Handle output of tokens.
SRTNEW	E		) Entry points for routine OUTPUT. Initialize
ENDNEW	E		) and close replacement values respectively.
SYNCH	S		Check if line count has changed and close out current buffer if it has.
CONVRT	R		Handle conversion between three datatypes used in the preprocessor.
ZACOP	R		Transfer contents of IVBs one to another.
GETDIC	R		Locate a dictionary entry for a variable with reference in text, perform some error checking on the entry, handle indirect references, and return both the relative and absolute addresses of the entry.
PH2SCN	R		Scan the text output by the first scan, perform replacements, and pass control to the interpreter when compile-time text is found.
DABTAB	R		Branch offset table. Used by routine PH2SCN.
DAOTAB	T		Branch offset table. Used by routine PH2SCN.
TPEEK	R		Check procedure reference argument list for syntax.
INCONT	R		Initialize %INCLUDE dataset.
XSTG	DSECT	RA	Private storage for module CC - in storage Root Module CA.
IEL0CC1	CSECT	R8	Entry point to module CC1.
INTPRT	R		Interpret compile-time code generated by the first scan.
ZATRV	T		Interpreter transfer vector.
CHKPDS	R		Resolve indirect references on the top of the push-down stack.
ADINVK	S		Set up argument list and invoke a procedure.
CHKLST	R		Process arguments being passed to invoke a procedure.
POPSTK	R		Remove one item from the top of the push-down stack.
ZAGET	R		Act as two-stream input.
ZAPUT	R		Output, via routine OUTPUTC (root module CA), the result character for logical operations.
ZALGCL	R		Process all logical operations.
ZACOMP	R		Process all comparison operations and produce a 'true' or 'false' result.
ZARITH	R		Process all arithmetic operations.
ZATRAN	R		Consists of two transfer operations: transfer, and transfer on false. The effect of a transfer is to cause the interpreter to alter the location from which it obtains an encoded operator.
ZATRAC	R		Execute a user-requested GOTO after checking its legality.
ZAASSIGN	R		Assign a new value to an identifier. If necessary, the old value is disposed of.
VALCHK	R		Check item to be added to push-down stack.
PUSHSTK	R		Add item to push-down stack.
ZARTN	R		Request a return from a procedure.
ZACVT	R		Convert a stack item to the type required by the RETURNS attribute.

Continued on next page

ZARTNS	E		Entry point to the routine ZACVT to return control to the final scan.
ZAABORT	R		Terminate processing on request.
ZACONCAT	R		Concatenate two strings.
CATFUT	S		Used by subroutine ZACONCAT to get new IVB.
ZAPAGE	R		Interpret the listing control statements %PAGE, %SKIP, and %CONTROL.
ZASKIP	R		
ZACNTRL	R		
XSTG	DSECT	RA	Private storage for module CC1 - storage in Root Module CA.
IEL0CC2	CSECT	R8	Entry point to module CC2.
PH2SCN	E		Entry point to CSECT IEL0CC2. The starting point for the final scan.
FUNCTN	R		Direct invocations of built-in functions to the appropriate invoked routine.
FUNCN2	E		Entry point to routine FUNCTN from routine INTPRT (module CC1).
ZJSUBS	R		Unstack SUBSTR function arguments, convert, and perform validity checks.
ZJLENG	R		Unstack and check arguments for the LENGTH function, perform validity checks, and return fixed item to top of push-down stack.
ZJINDX	R		Check arguments to INDEX function and return the fixed decimal result to the top of the push-down stack.
ZJIGNC	R		Scan second input stream to implement the INDEX function.
CHEXAR	R		Check that a built-in function has not been passed too many arguments and remove any excess ones.
LOCIVB	S		Obtain an IVB from the preprocessor variables dictionary.
CLOUT	R		Close output buffer and output record to SYSUT1 in text pages. Normally invoked via the entry point CLSBUF from routine OUTPUT (module CC) or OUTPUTC (root module CA), or on an unrecoverable error.
CLSBUF	E		Entry point to routine CLOUT.
PUNCH	R		Test if MDECK option is specified. If so, punch onto SYSPCH the last record put onto SYSUT1 by routine CLOUT.
PUNTAB	T		EBCDIC to BCD translation table.
TEMPNT	R		Terminate compile-time processing and pass control to the next phase. If there are no diagnostic messages, Phase EA is called. Otherwise Phase CE is called to print the messages.
			XROUT for module CC2 (not used by any other phase, so held locally).
XSTG	DSECT	RA	Private storage for module CC2 - storage in Root Module CA.

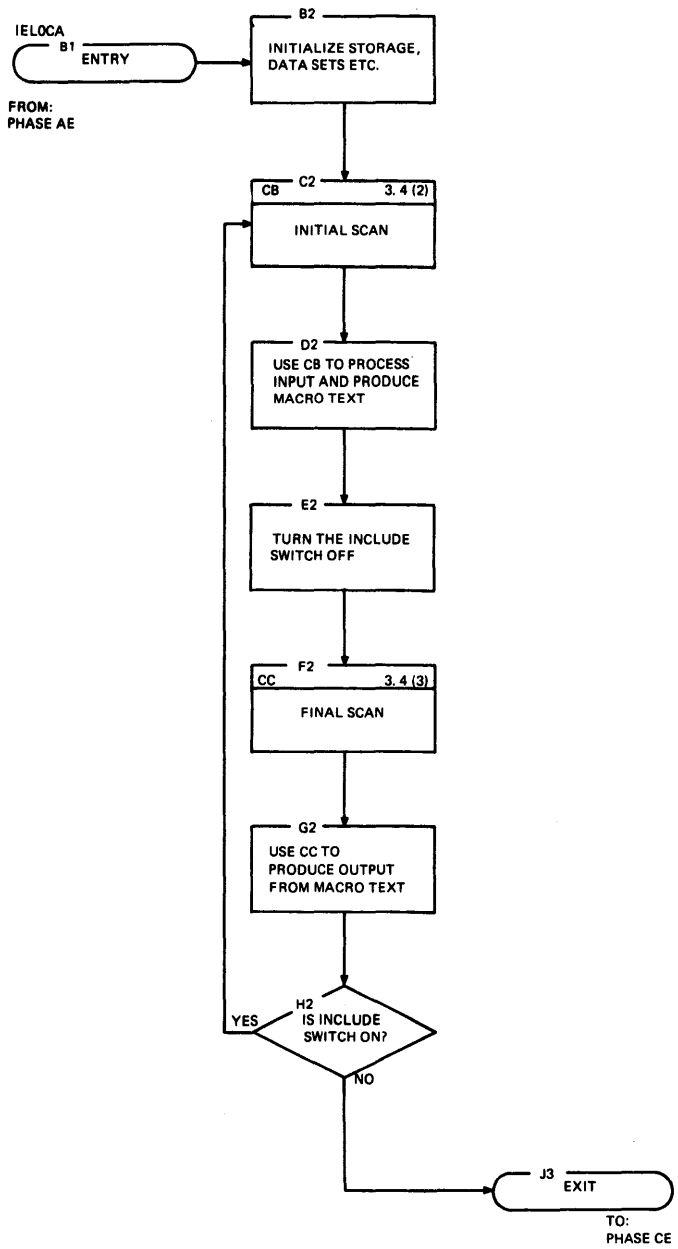


Chart 3.4. (Part 1 of 3). Compile-time Statement Preprocessor (Sub-phase CA)

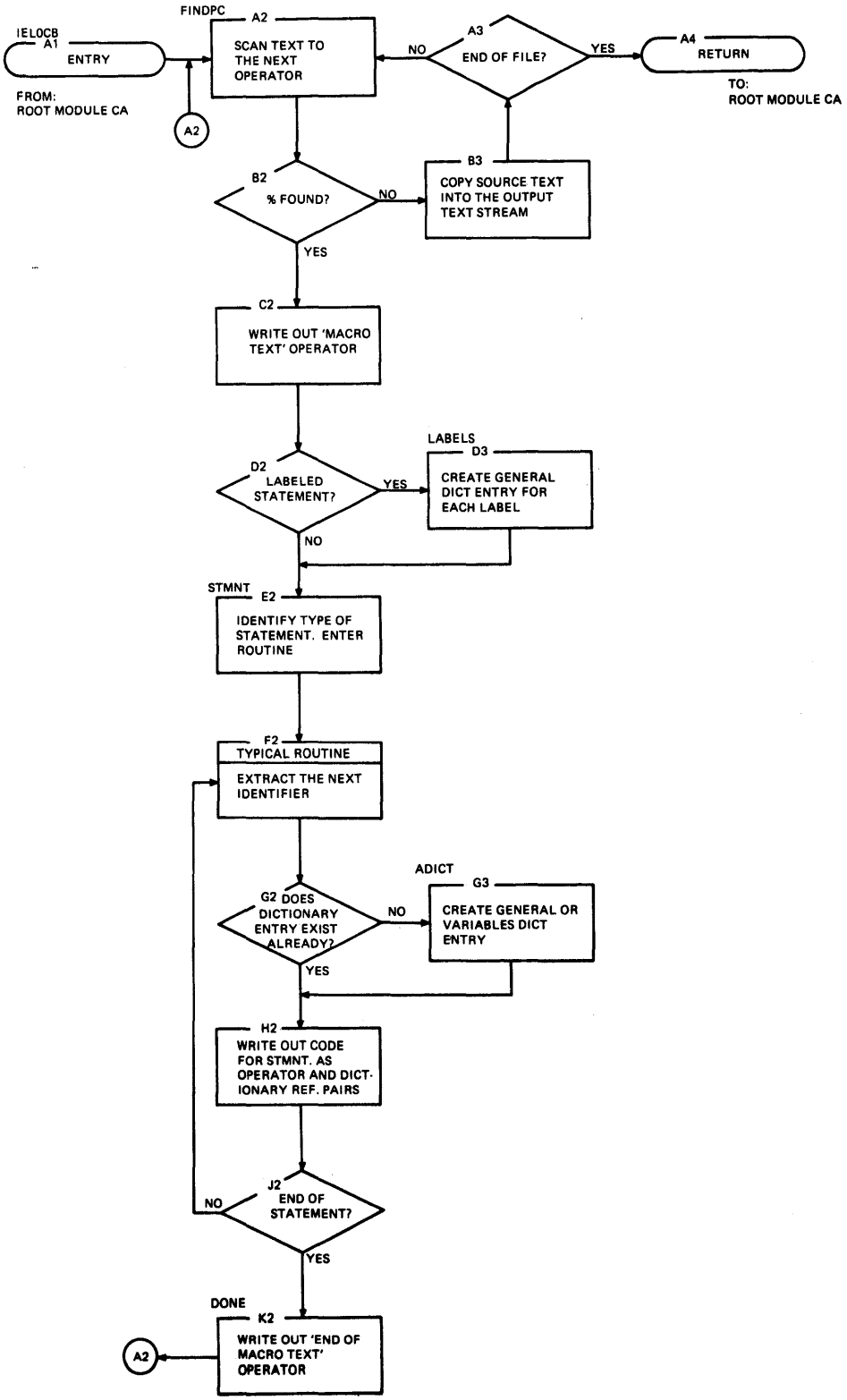


Chart 3.4. (Part 2 of 3). Compile-time Statement Preprocessor (Sub-phase CB)

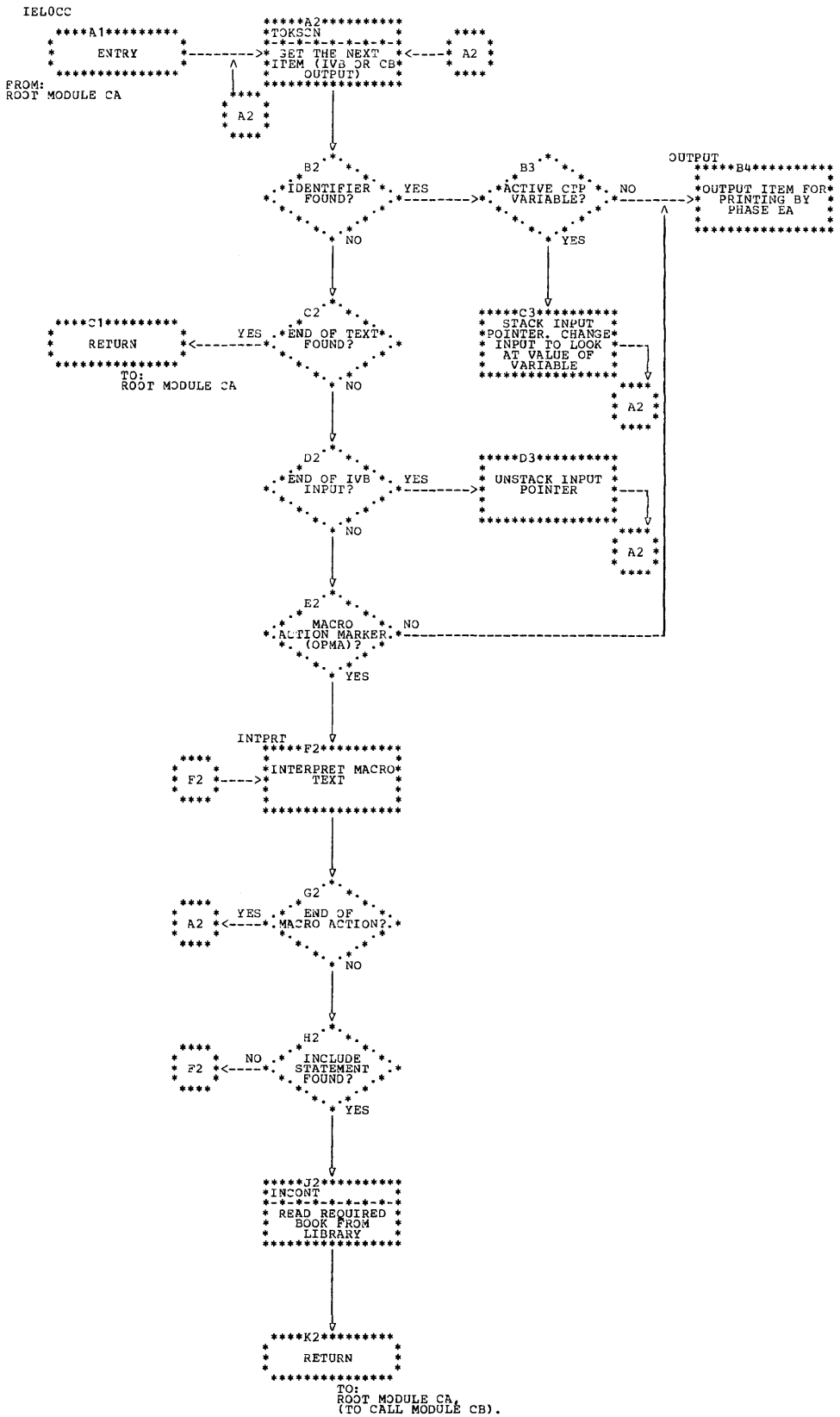


Chart 3.4. (Part 3 of 3). Compile-time Statement Preprocessor (Sub-phase CC)

Name	Type	Base registers	Function
IEL0CE.	CSECT		Entry point to the root module of Phase CE
CEC	R		Initialize registers, storage, etc.
UF1	R		Test FLAG option.
U1	R		Build sort units, i.e., generate a page stream of message entries from the input message stream for sorting purposes.
SU1	DSECT		) Describe sort units.
SU2	DSECT		)
SCHELL	R		Sort routine to sort all sort units in one page into order of severity, line number, and message number.
U97	R		This routine takes the sorted pages and merges them into a single text stream.
U55	R		Initialize print dataset. Print error-message page headings.
U38	R		Start of message printing loop. Determine the message number of the next message to be printed.
MED	DSECT		Describe entries in the table of edited messages.
UR3	R		Skip over a special purpose edited-message sort unit.
U100	R		Determine whether an edited message is to be printed, and if so, obtain the start of the list of special purpose edited-message sort units.
U99	R		Print the message severity subheading before the first message of a new severity level, and the message introduction information (i.e., message number, severity code, and statement number or numbers, if edited).
U90	R		Examine the message table (MESTAB) to determine whether a message has been provided for the number in the error message entry. If not, print a diagnostic.
U35	R		Message interpreting routine. Examine each code of a coded message to determine whether it is: 1. a keyword, 2. a level marker, 3. an end-of-message marker, 4. a text parameter, 5. a statement number parameter, 6. a dictionary parameter, 7. a quote marker, 8. an alternative text marker. If it is a level marker, multiply the level number by 256 to add to the keyword code following and to construct the correct keyword code with which to reference the keyword table (KEYTAB).
U32	R		Obtain a keyword from KEYTAB, given the keyword code.
U115	R		Add a keyword to the print buffer.
U26	R		Handle 'no blank' characters by concatenating words in a sub-buffer. Concatenate punctuation characters with keywords, if necessary.
U28	R		Scan to the next message, or invoke the control phase clean-up routine if the end of the message stream is reached.
U27	R		Convert a statement number to character form, when found in the middle of the message.
U29	R		Access a names dictionary entry when a dictionary reference parameter is encountered in a message.
U30	R		Determine a text parameter's type, and place the appropriate text in the message stream.

Continued on next page

UEDLST	S		Search edit table for a particular message number entry. If found, check that statement number in table entry is lowest encountered so far. If there is not table entry for that message, make one.
UBUF	S		Has three entry points as follows:
UBUF	E		Normal entry point. Text pointed to by RB, the length of which is in RC, is moved into the print buffer, and the line length, ULINE, updated by the amount moved in. If the line is full, it is printed, and the next buffer initialized. ULINE is incremented by one more than the length of the text moved into the buffer, to leave a space between each item on the line.
UBUF1	E		As for UBUF (E), except that the previous buffer is printed and a new line established for the new text item. A space is not left after this item, because the first item moved into the buffer is always the message identification letters.
UBUF3	E		As for UBUF (E), except that no allowance is made for space after each item.
ZTRAN1	T		Internal code-to-EBCDIC translation table.
XINIT	CSECT	R9	)
XDISC	CSECT	R9	)
XRFAB	CSECT	R9	)
XDIREC	CSECT	R9	)
XPRNTN	CSECT	R9	)XROUT
XBREAK	CSECT	R9	)
XTXPG	CSECT	R9	)
XBRIC2	CSECT	R9	)
XPRNT	CSECT	R9	)
XSTG	CSECT	RA	Private storage for Phase CE.
CET	CSECT		Second module, containing message tables.
MESTAB	T		Table of coded messages.
MESREF	T		Table relating message number to coded message in MESTAB.
KEYTAB	T		Table of keywords used in messages.
KEYREF	T		Table relating length of a keyword to its position in KEYTAB.
MCDE	T		Table of codes giving, for each message number: 1. severity of message, 2. editing requirements for message, 3. presence or absence of line number parameter in message.

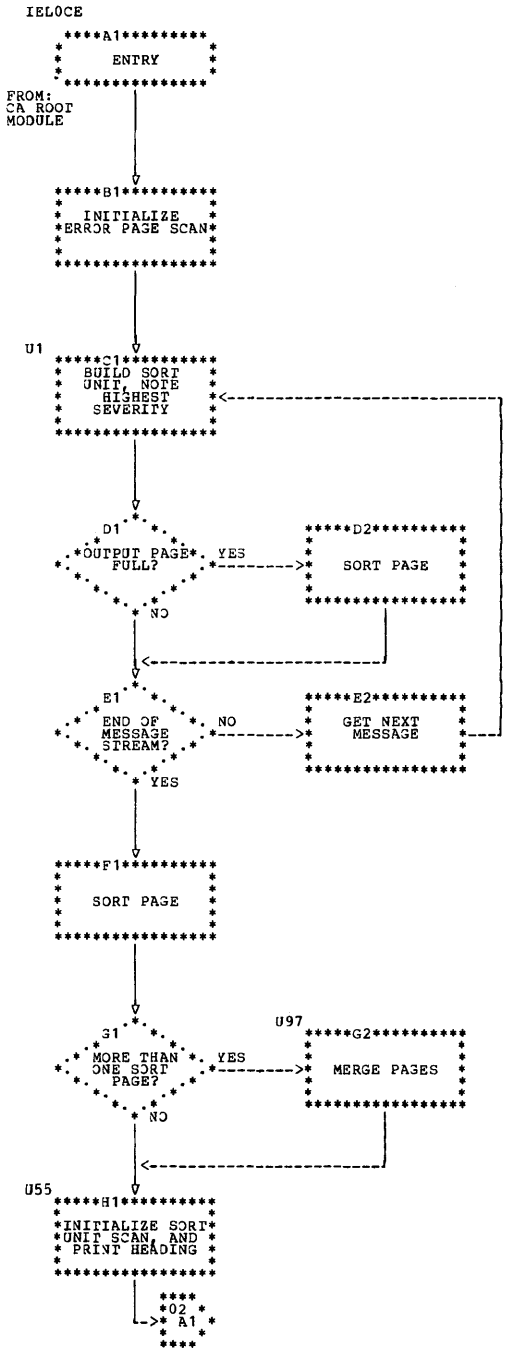


Chart 3.5. (Part 1 of 2). Compile-time Preprocessor Error Editor (Phase CE)

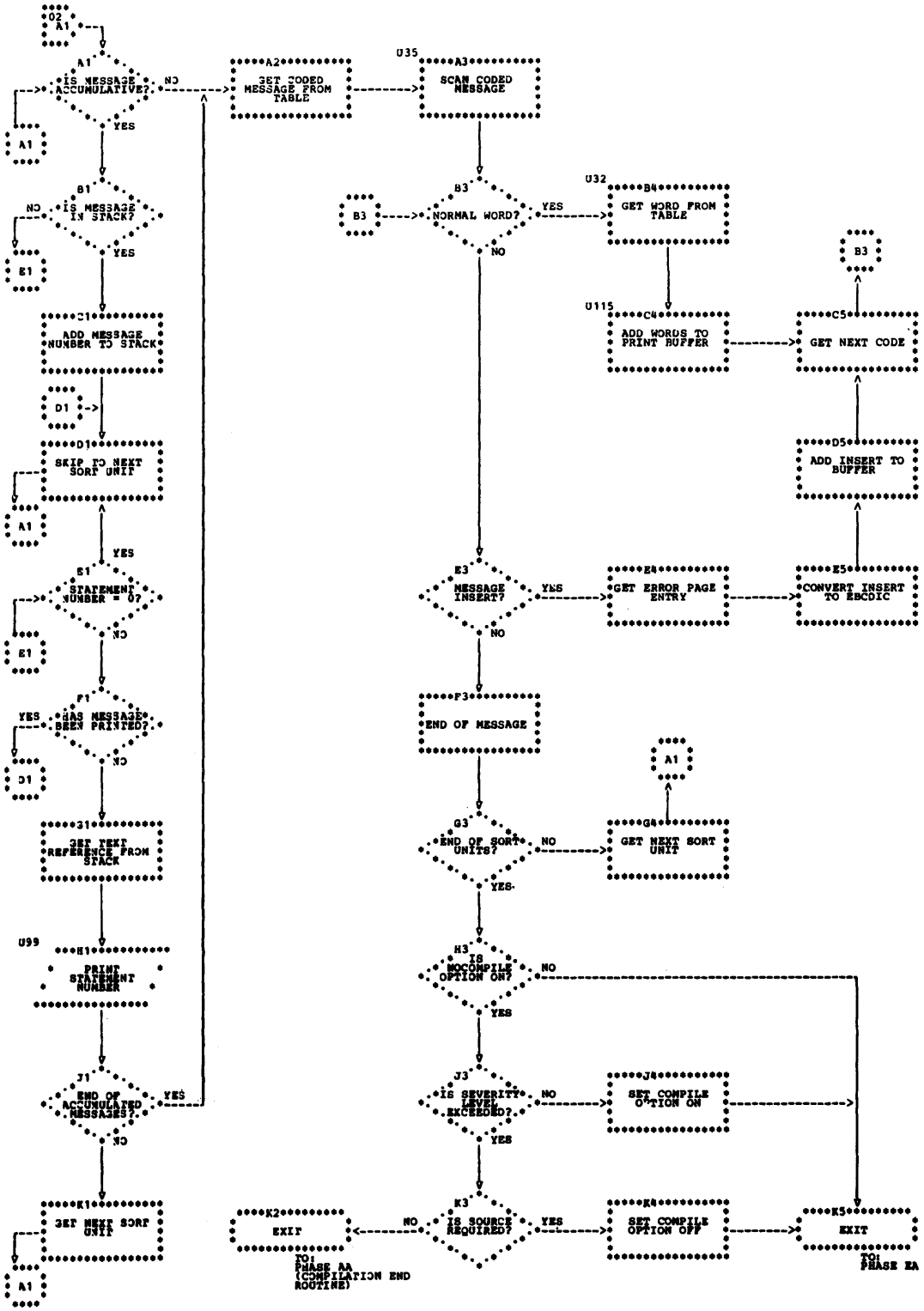


Chart 3.5. (Part 2 of 2). Compile-time Preprocessor Error Editor (Phase CE)

Syntax Analysis Pass 1 (Phase EA)

Name	Type	Base registers	Function
IEL0EA	CSECT	R6,R9	Entry point to the root module of Phase EA.
EAA	R		Initialize registers, storage, input/output, etc. Call module EAC for main scan of text.
SINGLE	S		Prccess and output an expression enclcsed in parentheses.
EXP	S		Prccess an expression.
OPTOR	S		Write out operator code and obtain next non-blank character, if the current input is an operator.
SQUID	S		Process subscripted/qualified identifier.
IDENTR	S		Test input identifier for validity, and output it.
ACONST	S		Process an arithmetic constant.
DECINT	E		Entry point to ACONST for a decimal integer constant.
MOVR1	S		Scan to next byte and call READR if end of last section in read area.
SCONSTR	S		Process a string constant.
SCONST	E		
INCR	S		Get the next non-blank character in the current record.
BUMP	E		
STRNG	S		Skip over a string constant without moving the text.
SKPLST	S		Delete some invalid text and issue a diagnostic.
READS	S		Read an input record into the read area, convert it
READR	E		to internal format, and test for invalid characters.
XINIT	CSECT	R9	)
XMESGR	CSECT	RF	)
XPRNTN	CSECT	R9	)
XBREAK	CSECT	R9	) XROUT
XTXPG	CSECT	R9	)
XBRIO1	CSECT	R9	)
XSTG	CSECT	RA	Private storage for Phase EA.
EAC	CSECT	R7,R8	Entry point to module EAC.
ECBASE1	R		Initialize registers, storage, and scan of text.
RSTART	R		Scan text, identifying verbs and prefixes, and calling the relevant statement processing routines.
BADST1	R		Deal with unrecognizable statement start.
NULINS	S		Insert a null statement in the output text stream.
ERRORS	S		Examine erroneous statement as an assignment.
CHECKS	S		Check push-down stack for IF or ON nesting.
NONEX	S		Process non-executable statement (i.e. IF X THEN;, ELSE;, or ON condition;) and insert NULL statement.
LBLGEN	S		Generate label for an unlabeled PROC or ENTRY statement.
INLBL			Process initialization of label variable.
NOPROC	R		Generate dummy PROC and ONPROC statements.
LSCRD	S		Handle end-of-file on the input stream, and end-of-space in the read area.
EACHNR	S		Load the chain field for a block-heading statement.
STACKER	S		Stack an item in the push-down stack.
NTRY	R		Process PROC or ENTRY statement.
PROCST	E		
LEAVST	S		Process LEAVE statement.
WHENST	S		Process WHEN statement.
SELECTST	S		Process SELECT statement.
OTHERST	S		Process OTHERWISE statement.
PRNTST	S		Process %PRINT statement.
NPRNST	S		Process %NOPRINT statement.
ENDST	R		Generate END statement for PROC, BEGIN, or DO statement.
ATNST	R		Process RETURN statement.
DOST	R		Process DO statement.

ASSIGN	R		Process ASSIGN statement.
BGINST	R		Process BEGIN/CNB statement.
ONLAB	S		Process label on an ON statement.
GOTOST	R		Process GOTO statement.
IFST	R		Process IF statement.
ELSEST	R		Process ELSE phrase.
ONST	R		Process ON statement.
PAGERTN	R		Process listing control keywords PAGE and SKIP.
SKIPRTN	E		
CHKRTN	R		Check the syntax of checkout compiler verbs found in input stream. (applies to OS version only.)
DCLDFST	R		Output dummy NULI statement for DCL/DFT following THEN, ELSE, or ON.
STAT2	R		Copy out a statement which is not handled by this phase.
EAT	DSECT	R5,R9	Module containing tables and subroutines. Note that actual tables are on text pages created by phase EC.
ZTRAN1	T		External (EBCDIC) to internal translate table.
LETTAB	T		Table for XLET macro.
BLTAB	T		TRT table for the INCR routine in EAA.
KYWD	S		Subroutine to replace a PL/I keyword by an internal character.
STATID	T		Verb names.
MISCID	T		Miscellaneous tables used by the KYWD subroutine.
ONID	T		Condition names
LISTER	S		Subroutine to control the source listing format.
LISTER2	S		Subroutine to handle SPACE and EJECT on the source listing.
CHECKLST	S		Process names in a CHECK list.
ABORT	R		Abort routine, to print buffer and exit to phase UA.
XTRCEL	S		Label trace subroutine.
POPLST	R		Condition prefix routine.
DOSTMT	S		DO statement subroutine.
FREEST	R		FREE statement routine.
WAITST	R		WAIT statement routine.
FRST	R		FETCH and RELEASE statement routine. (Applies to CS version only.)

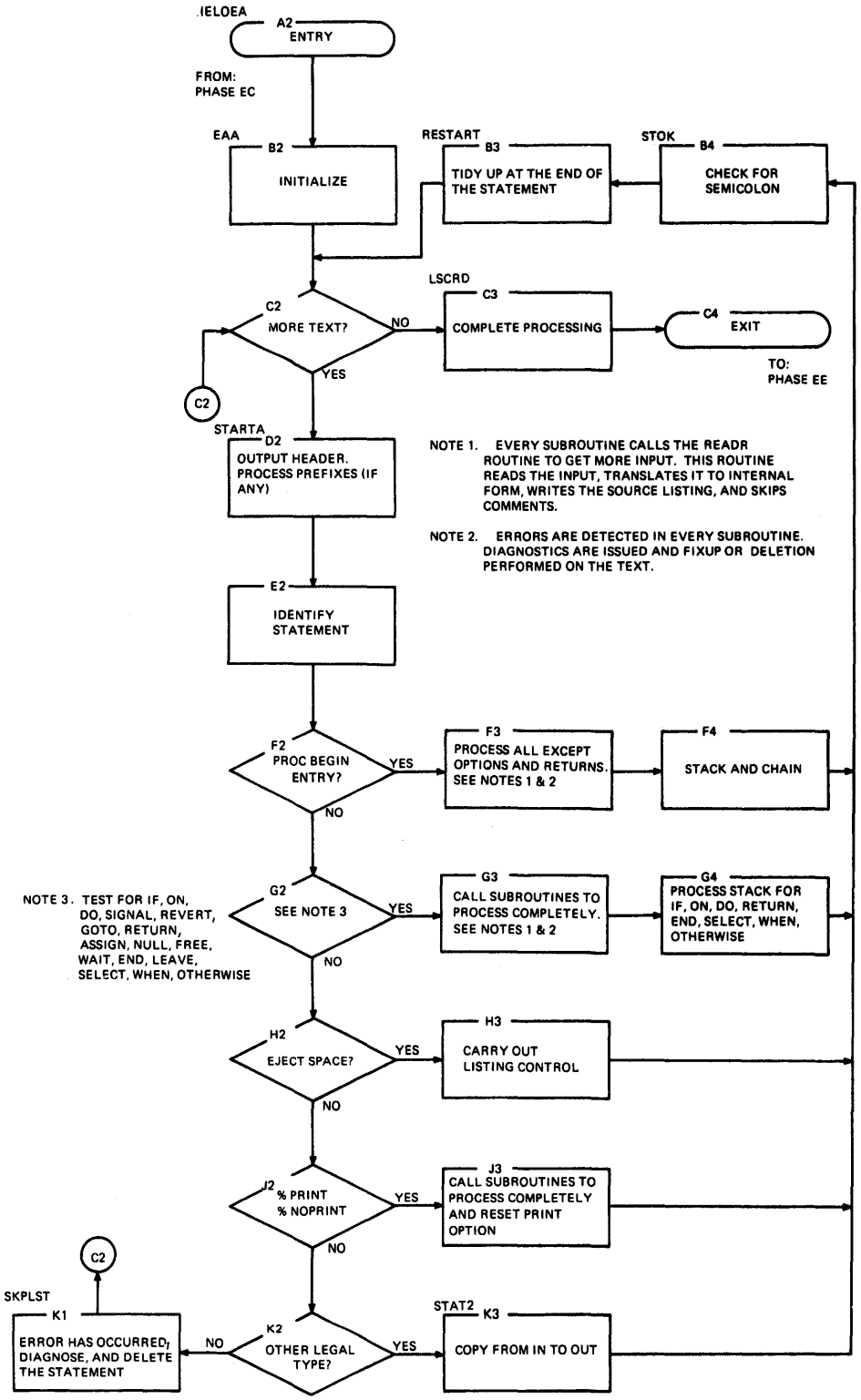


Chart 3.6. Syntax Analysis - Pass 1 (Phase EA)

Name	Type	Base registers	Function
IEL0EE	CSECT	R6	Entry point to Phase EE.
EEA	R		Initialize registers and storage.
SINGLE	S		Process and output an expression enclosed in parentheses.
EXP	S		Process an expression.
OPTER	S		Write out operator code and obtain the next non-blank character if the current input is an operator.
SQUID	S		Process a subscripted/qualified identifier.
IDENTR	S		Test input identifier for validity, and output it.
ACONST	S		Process an arithmetic constant.
DECINT	E		Entry point to ACONST for a decimal integer constant.
SCONSTR	S		Process a string constant.
SCONST	E		
INCR	S		Get the next non-blank character in the current record.
STOFLD	R		Nesting overflow routine.
SKIPT	S		Skip over invalid text in the input text stream.
PSQUID	S		Process parenthesized subscripted/qualified identifier.
XTRCEL	S		Label trace subroutine.
XINIT	CSECT	R9	)
XMESGR	CSECT	R9	)
XBREAK	CSECT	R9	) XROUT
XTXPG	CSECT	R9	)
XBRIC2	CSECT	R9	)
XSTG	CSECT	RA	Private storage for Phase EE.
EEE	CSECT	R7,R8, R9	Initial entry point to module EEE.
EEE	R		Initialize registers and text input/output.
SCNA	R		Main text scanning loop.
PLAB	R		PROC statement processor.
BGNLAB1	R		BEGIN statement processor.
READLAB	R		READ/WRITE/REWRITE/UNLOCK/DELETE statement processor.
LOCATLAB	R		LOCATE statement processor.
CALLAB	R		CALL statement processor.
EXPLST	R		Process a parenthesized list of expressions.
BADST3	R		Invalid statement routine.
ONEND	R		Insert a generated END statement for a single-statement on-unit.
ELAB	R		END statement processor.
CHKLAB	R		Final processor of CHECK/NOCHECK prefix options.
ONLAB	R		Final processor of ON statements.
ONBGNLAB	R		Process an ONB block.
COPYIO	R		Skip text or copy it from input to output.
OPTNSE	S		Options list processor.
OPTNS	E		
SKSTMT	S		Skip to the end of the current statement.
PROCOUT	S		Handle internal block for chaining in the dictionary text file.
NEWBLK	S		Handle chaining where new blocks are started.
STOK	R		Skip to the next semicolon and output it.
JMPTXT	S		Branch to a different location (RB) in the input text stream.
DELAYLAB	R		DELAY statement processor.
DSPLYLAB	E		DISPLAY statement processor.

Continued on next page

STREAMIO	R		Set switch and copy stream I/O into output for Phase EI.
OPENLAB	R		OPEN statement processor.
CLOSELAB	E		CLOSE statement processor.
TERMINAT	R		Wind up processing and exit to Phase EI.
EEC	CSECT	R5,R8,	Module containing tables and attribute processing routines.
LETTAB	T		Table for the XLET macro.
CSTRT	T		Branch table offsets for the COPYIO routine.
PICTAB	T		TRT table for picture characters.
ATTRTAB	T		TRT table for DCL attributes.
KYWD	S		Subroutine to replace a PL/I keyword by an internal character.
RIOOP	T		)
RDOP	T		)Record I/O keywords.
CALLID	T		)
OPENID	T		OPEN/CLOSE statement options.
DSPID	T		DISPLAY statement options.
ATTID	T		Attribute table.
DFTID	T		DFT statement options.
ENTID	T		PROC/BEGIN/ENTRY statement options.
OPTID	T		PROC/BEGIN options.
MISCID	T		Miscellaneous.
ENVID	T		ENVIRONMENT attribute options.
ENVIR	S		Process the ENVIRONMENT attribute.
DECLBASE	S		Process text containing a DCL, DFT, or ALLOC statement.
DECLN	S		Process a declaration.
ATTLST	R		Process an attribute list.
RADIXL	R		Process BINARY/DECIMAL/FIXED/REAL/COMPLEX attributes.
FLOATL	R		Process FLOAT attribute.
STRNGL	R		Process CHAR/BIT attribute.
ENVIRAT	R		Process ENV attribute.
ATTVLD	R		Process ALIGNED/UNAL/AUTO/STATIC/PTR/VARYING/CTL attributes.
ATTIVD	R		Illegal attribute routine.
FTTLST	R		Complete processing of a factored attribute list.
DIMENS	R		Process dimensions.
CHARBT	S		Process CHAR/BIT attribute.
REFOP	S		Process REFER attribute.
DCLOPTNS	R		Process OPTIONS list.
LIKEAT	R		Process LIKE attribute.
PICTUR	R		Process PICTURE attribute.
PMOVE	S		Move characters from START3 to output.
PREC	S		PRECISION routine.
FLOATAT	E		Entry point to PREC for a FLOAT variable.
DEFLT	R		Process DEFAULT statement.
GENER	R		Process GENERIC attribute.
RTRNS	S		Process ENTRY/RETURNS/AREA attributes.
BASE	R		Process BASED attribute.
SETIN	R		Process SET/IN options or ALLOCATE statement.
INITIAL	R		Process INITIAL attribute.
CALLOPT	S		Process CALL option in an INITIAL declaration.
POSTN	R		Process POSITION attribute.
LABATT	R		Process LABEL attribute.
DEFINED	R		Process DEFINED attribute.
ALLOCST	R		Process header for ALLOCATE statement.
LABOPT	R		Process labels on DCL/DFT/ALLOCATE statements.
EEDINT	S		Storage initialization subroutine.

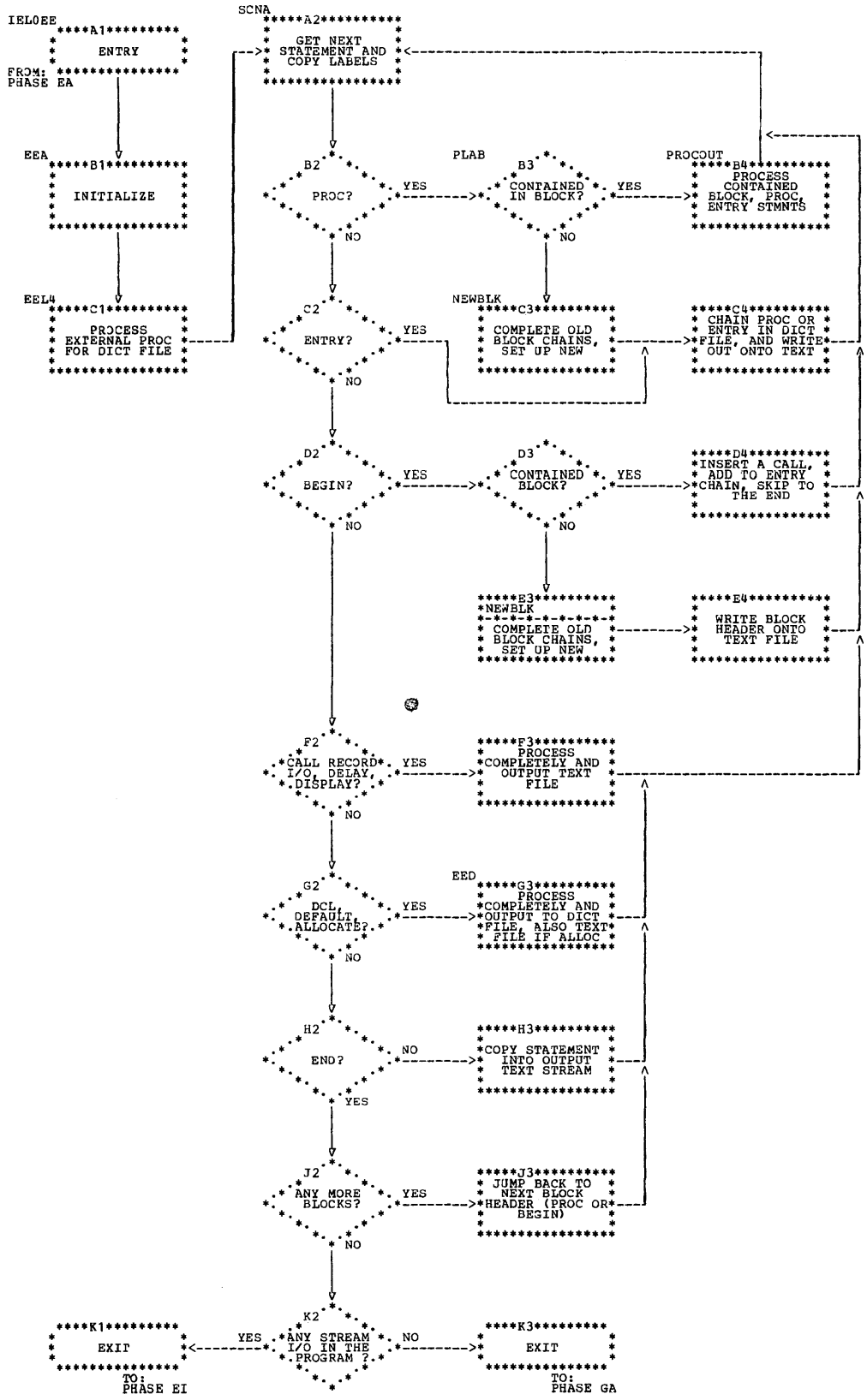


Chart 3.7. Syntax Analysis - Pass 2 (Phase EE)

Name	Type	Base registers	Function
IEL0EI	CSECT	R5,R6, R7	Entry point to Phase EI.
EIEI	R		Initialize registers, storage, and scan of text stream.
CLENS	T		Table of maximum lengths for arithmetic constants.
LETTAB	T		Table for the XLET macro.
CSTPT	T		Branch table offsets for the COPY/SKIP routine in EEE.
PICTAB	T		TRT tables for picture characters.
SINGLE	S		Process and output an expression enclosed in parentheses.
EXP	S		Process an expression.
OPTOR	S		Write out operator code and obtain next non-blank character if the current input is an operator.
SQUID	S		Process subscripted/qualified identifier.
IPENTR	S		Test input identifier for validity, and output it.
ACONST	S		Process an arithmetic constant.
SCONSTR	S		Process a string constant.
SCONST	E		
INCR	S		Get the next non-blank character in the input text stream.
BUMP	E		
STOFLO	R		Nesting overflow routine.
SKIPT	S		Skip over invalid text in the input text stream.
PSQUID	S		Process parenthesized subscripted/qualified identifier.
KYLID	S		Subroutine to replace a PL/I keyword by an internal character.
FMTID	T		Format items)
GPID	T		GET/PUT options) tables used by the KYWD subroutine.
MISCID	T		Miscellaneous)
SCNA	R		Main text scanning loop.
FORMATLB	R		Common processing routine for format lists, and
GETLAB	E		statement header processor.
PUTLAB	E		
DATLST	R		Data list processing routine.
DOSPEC	S		Process DO repeat specification in a data list.
FMATLST	S		Process format list.
EXPLST	S		Process an expression list enclosed in parentheses.
BADST3	R		Invalid statement routine.
STOK	R		Skip remainder of statement and output a semicolon.;
TEMINAT	R		Wind up Phase EI and exit to Phase GA.
XINIT	CSECT	R9	)
XMESCR	CSECT	RF	)
XBREAK	CSECT	RF	)XROUT
XTXPG	CSECT	R9	)
XBRIC1	CSECT	R9	)
XSTG	CSECT	RA	Private storage for Phase EI.

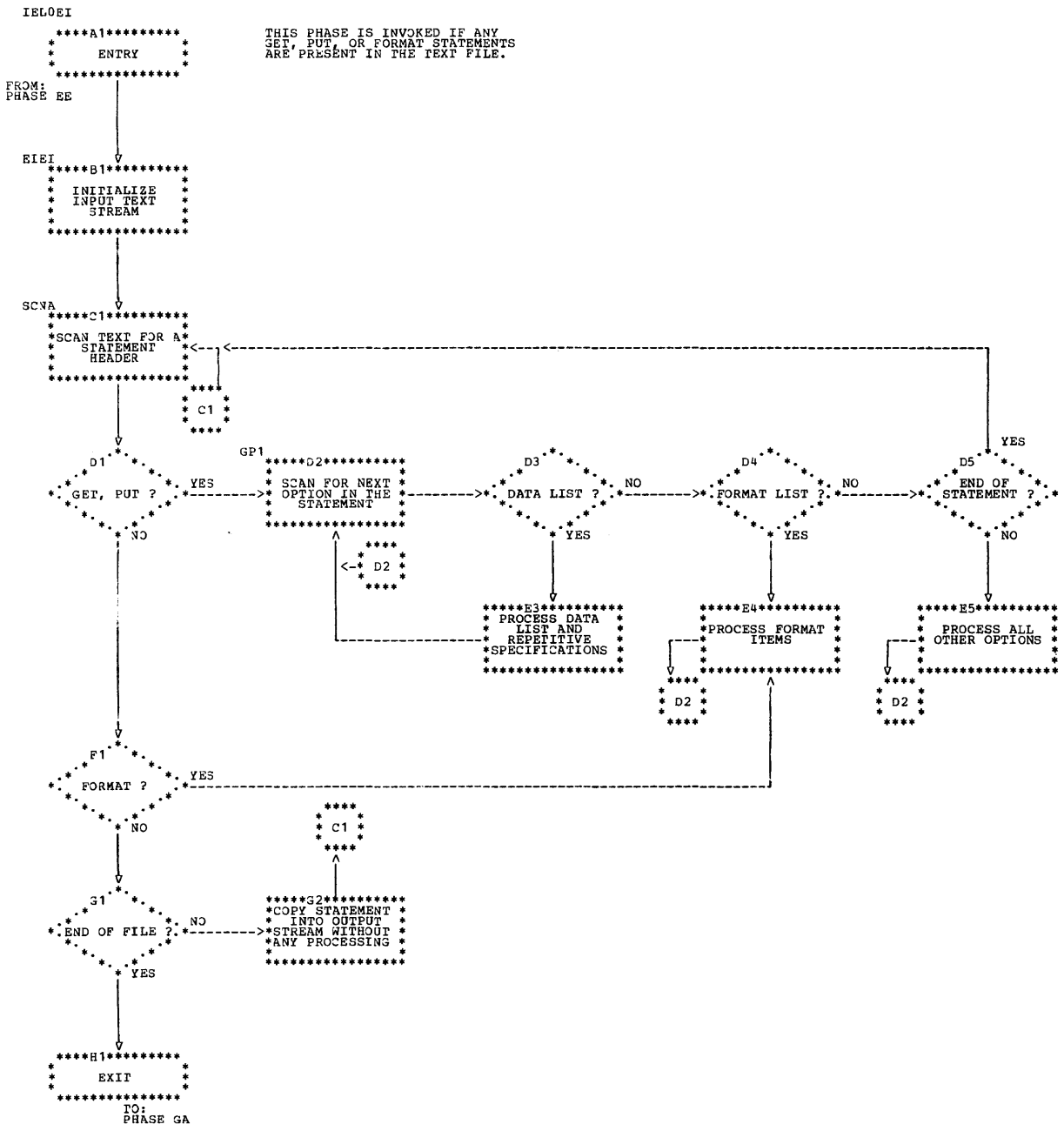


Chart 3.8. Syntax Analysis - Pass 3 (Phase EI)

Name	Type	Base registers	Function
IEL0GA	CSECT	R8,R9	Entry point to Phase GA.
GAA	R		Initialize registers, storage etc.
RESPCE	R		Reserve space in the general dictionary for block headers and labels.
GADCL	R		Select next item in the block header, DFT, PROC, ENTRY, or
NEWBLK	E		DCL statement chain.
DCLO2	E		
DCLSTAT	R		Initialize processing of a DCL statement.
DCLST1	E		
DCLA	R		Find a new declared item and process, ignoring its structure level. Scan the statement and 'collect' all declared attributes for the item. Make 'arithmetic' or 'coded' implications.
SKPATT	R		Skip the attribute pointed at by the input text pointer (R1).
DCL4	R		Find structure level of the next declared item.
DCL6	R		Search the default directory and 'collect' all specified default attributes.
DCL7	R		Determine whether the item is a major structure, minor structure, base element, or is not in a structure.
DCL8	R		Make a names dictionary entry and detect multiple declaration.
STRUCTR	R		Set up structuring information.
ATTIMPS	R		Generate variable, file and entry attribute implications.
DCL9	R		Make attribute selections using the attribute tree.
DCLVQ	R		Make a general or variables dictionary entry.
DCL102	R		Set the appropriate attributes bits.
GASZAL	R		Calculate size, alignment, precision and scale, or length.
GANXDCL	R		Select the next declaration.
GANXDO	E		
GANXDCL1	E		
EPIDSC	E		
DEFACTOR	S		Scan to the next ')' at the current parenthesis level.
MJSTRAL	S		Set the alignment of a major structure to the maximum alignment of its members.
LPSO	R		Deal with endpage and end-of-statement during LIKE processing.
LPS1	R		Expand a LIKE declaration for a structure.
NXTLIKE	R		Locate the next LIKE declaration in the LIKE directory.
NXL2	R		Access the next member in the current LIKE structure.
LPS10	R		Return to scan of like structure on completion of the expansion of a minor structure, if more exists.
GAPARM	R		Access the block header dictionary entry for a block, and initialize the processing of contextually declared parameters.
NXTPARM	R		Scan to the next parameter, and branch as required.
GAPM5	E		Access the next entry-point dictionary entry for this block and initialize the parameter scan.
GAPM6	E		Process the next parameter for this entry point.
NXTDSCR0	R		Set up descriptor list declarations.
NXTDSCR	E		
VRS00	R		Resolve explicitly declared names appearing in ENV options.
GAIMP	R		Generate sample implicit declaration entries for each different DEFAULT range.

Continued on next page

NXTIMP	E		
PRENTRY	R		Process entry point, making names dictionary entries for any formal parameters.
ENT10	R		Make a dictionary entry for the entry point,
ENTNXT	E		clear default attributes, and make a block header dictionary entry if there are no more entry points for the block.
DFSTAT	R		Process DFT statements, and build a GA-time default directory.
XINIT	CSECT	R9	)
XRFSEQ	CSECT	R9	)
XRFAB	CSECT	R9	) XROUT
XDIREC	CSECT	R9	)
XTXPG	CSECT	R9	)
XSRCH	CSECT	R9	)
XSTG	CSECT	RA	Private storage for Phase GA.
GAB	CSECT	R7	
RTN	T		Table of addresses to attribute-processing routines shown below.
STATR	S		STATIC
CLTR	S		CONTROLLED
SMSTR	S		SYSTG (system default storage)
SALR	S		SYSAL (system default alignment)
UNALR	S		UNALIGNED,
DIMSR	S		DIMENSIONS
PTRR	S		POINTER
BSDR	S		BASED
EVNTR	S		EVENT
TASKR	S		TASK
INITR	S		INITIAL
DEFR	S		DEFINED, iSUB definition
POSR	S		POSITION
OFFR	S		OFFSET
FIXR	S		FIXED
BINR	S		BINARY
CHARR	S		CHAR
BITR	S		BIT
AREAR	S		AREA
STRING1	R		)
STRGRET	R		)
CHARR3	R		)Common processing routines for data variables.
BITR3	R		)
AREA3	R		)
STVR	S		String length or area size (set in STRVAL).
REF2FL	S		Make overflow for text references of an attribute, if necessary.
VALR	S		VALUE (in DEFAULT statement)
VARR	S		VARYING
CPLXR	S		COMPLEX
PRC2R	S		Two-value precision
PRC1R	S		One-value precision
LABR	S		LABEL
PCNR	S		PICNUM
PCRR	S		PICHAR
LIKE	S		LIKE
PARMR	S		PARAMETER
RTNR	S		RETURNS

Continued on next page

FILR	S		BUFF, UNBUFF, EXCL, KEYED, STREAM, RECORD, BACK, SEQ, DIRECT, PRINT, INPUT, OUTP, UPDATE
FILER	S		FILE
ENTR	S		ENTRY
ENVR	S		ENV
SKPATT	S		Skip an attribute.
BIFR	S		BIF
IMBIF	S		Mark an entry as implicit built-in.
INTR	S		INTERNAL
EXTR	S		EXTERNAL
REDR	S		REDUCIBLE
GENR	S		GENERIC

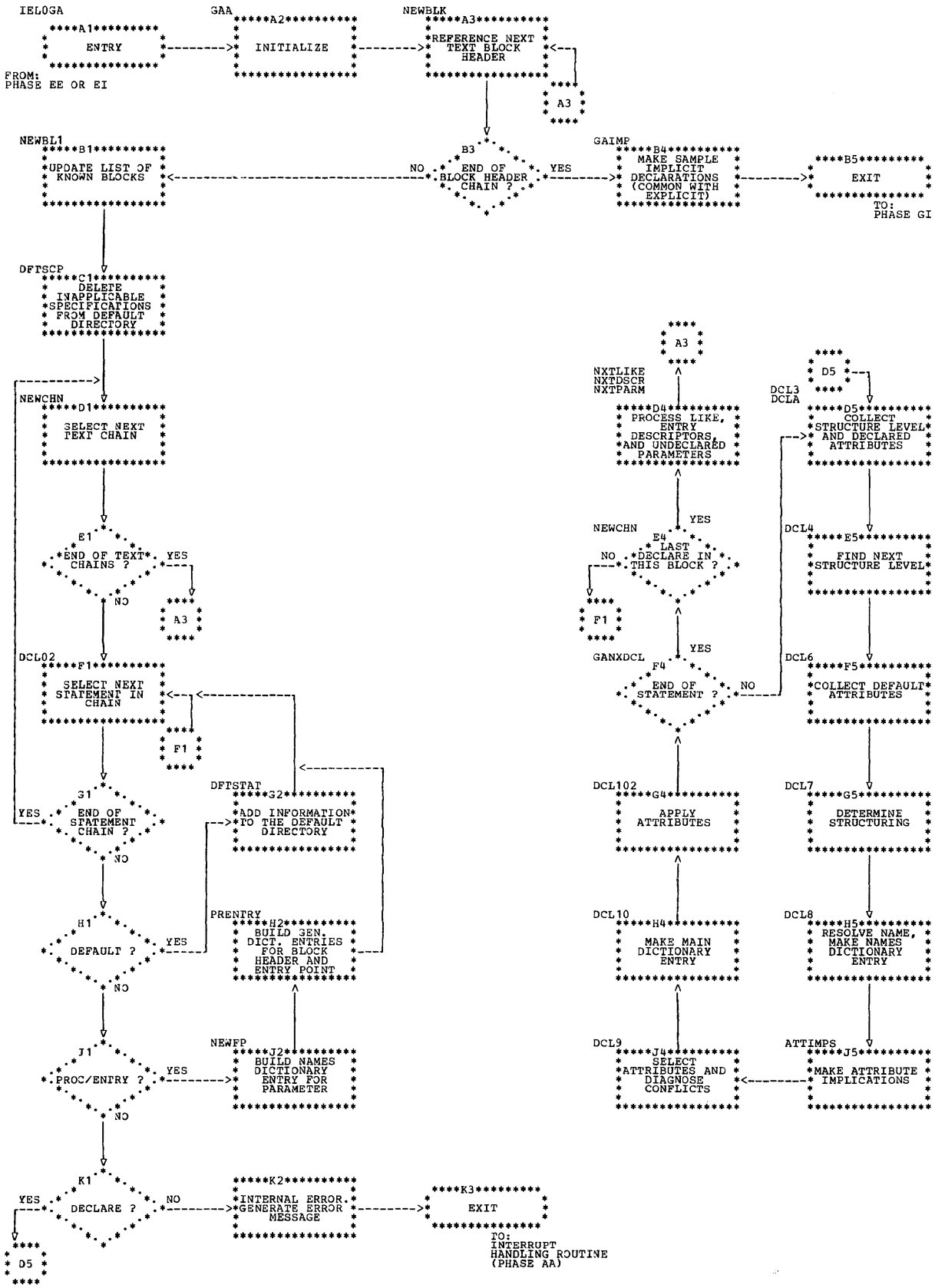


Chart 3.9. Explicit Declarations Phase (Phase GA)

Name	Type	Base registers	Function
IEL0GI	CSECT		Entry point to Phase GI.
GI	R		Initialize registers, storage, tables, and scan of text.
GITRTA	R		Dump text, text scan pointer (R1) to skip a byte.
GITRT	E		Test the text code byte pointed at by R1, and branch to the relevant processing routine.
CLEARTN	R		A semicolon has been found. Test for error and continue scan.
SLSNRTN	R		New statement has been found. Test the statement type and branch accordingly.
SL1	R		Process any labels on this statement, and make dictionary entries.
SCOPE2	R		A PROC block header has been found in the dictionary text stream. Update the list of known blocks (KNOLST), and skip to the next statement.
SL3A	R		Skip over a BEGIN, CALL, PROC, SIGNAL, or ONB statement in the dictionary text stream.
SL3	R		A PROC block header has been found in the main text stream.
SL4	E		Update the list of known blocks (KNOLST), and skip to the next statement.
SL5	R		Signal contextual BIF found, and branch to SL1.
SL7	R		Scan over the statement header and continue processing.
SL8	R		Check for ambiguous label reference.
SL9A	R		Branch to SL1 if ALLOCATE statement in main text stream.
SL10	R		Process a WAIT statement.
DFTRTN	S		Create a variables dictionary entry for a DFT variable.
NAMERTN	R		Check a contextual declaration and create a names dictionary entry.
SLVLRTN	R		Analyze the context of a second-level marker.
POINTRTN	R		Skip over a qualifying name.
LAPRTN	R		Keep track of the current parenthesis level and context
RPARTN	E		indicator CCTX.
ODDSRTN	R		Trap any relevant byte which may have been missed by SLVLRTN.
CONTEXT	S		Partially process a contextual declaration involving a file.
CONDRTN	R		Make a dictionary entry for a programmer ON-condition name.
ENV1	R		Scan a chain of ENV attribute dictionary entries, and make entries for any undeclared variables appearing in the format list.
GIEND	R		Test for completion of a dictionary text stream scan, and commence if it has not been done.
GIOUT	R		Wind up processing by this phase.
XINIT	CSECT	R9	)
XRFSEQ	CSECT	R9	)
XRFAB	CSECT	R9	)
XDIREC	CSECT	R9	)
XDSTAT	CSECT	R9	) XROUT
XPRNTN	CSECT	R9	)
XTXPG	CSECT	R9	)
XBRIC2	CSECT	R9	)
XSRCH	CSECT	R9	)
XSTG	CSECT	R4	Private storage for Phase GI.

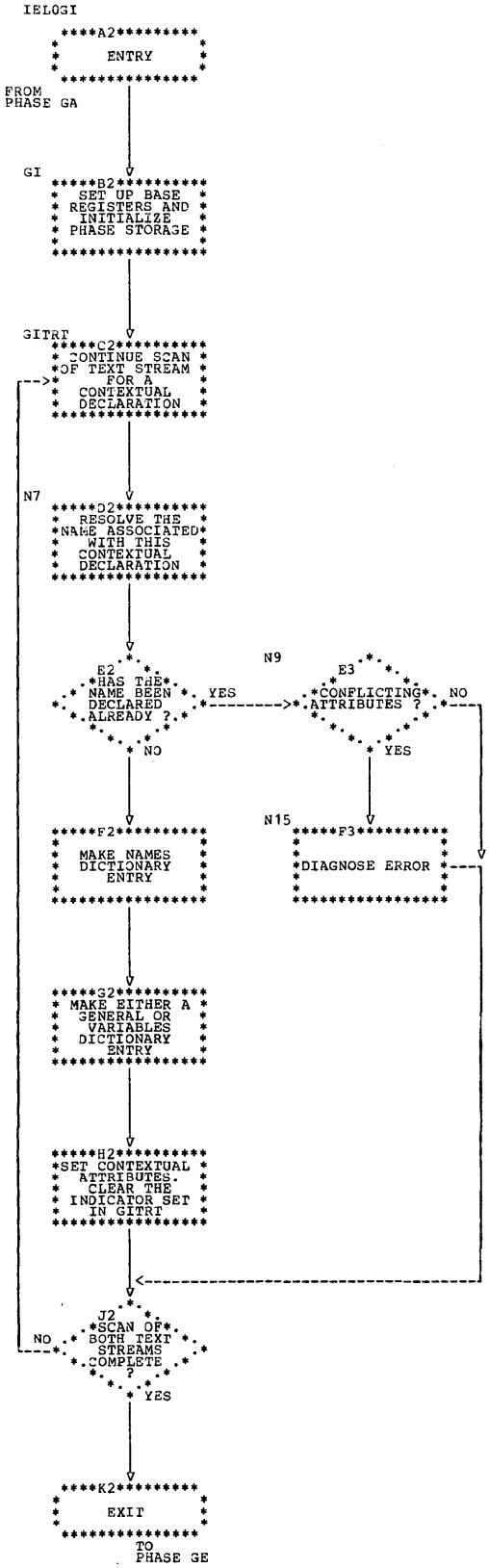


Chart 3.10. Contextual Declarations Phase (Phase GI)

Name	Type	Base registers	Function
IEL0GE	CSECT	R8,R9	Entry point to Phase GE.
GEX	R		Initialize registers, storage etc.
GEA	R		Initialize scan of DEFAULT entries.
GE1	R		Initialize scan of variables dictionary.
GE2	R		Get the next dictionary page and lock it into main storage.
GE3	R		Determine whether processing is required for an entry.
GE4A	R		Set up an aggregate table in workspace (YGTAB).
GE12	R		Attempt commoning for array or major structure, or generate a branch to a structure commoning routine.
GE16	R		Complete the processing of a dictionary entry, and scan to the next entry.
E801	R		E801 to LERR are error and diagnostic routines.
EXIT1	R		EXIT1 to EX2 are exit-testing routines.
GEND	R		Close Phase GE.
GEOUT	r		Pass control to Phase GM.
GE20	CSECT	R9	Carry out commoning, and, when necessary, uncommoning, of aggregate tables for structures.
GEQ	CSECT	R9	
GEQ	R		Carry out the uncommoning of declaration expression file statements for aggregates which fail to common.
GEJ	R		Copy an aggregate table from workspace into the general dictionary and maintain various tables of information on structuring, chaining etc.
GENRTN	CSECT	R9	Process GENERIC entry.
DEFCSECT	CSECT	R9	Process DEFINED entry.
DEFVAR	E		Entry point to DEFCSECT.
GE6	CSECT	R9	Process adjustable bounds, including creation and commoning of declaration expressions file statements.
XINIT	CSECT	R9	)
XMESGR	CSECT	RF	)
XRFSEQ	CSECT	R9	)
XRFAB	CSECT	R9	)XROUT
XDIREC	CSECT	R9	)
XBREAK	CSECT	RF	)
XTXPG	CSECT	R9	)
XSRCH	CSECT	R9	)
XSTG	CSECT	RA	Private storage for Phase GE.
GEB	CSECT	R8,R9	Second module of Phase GE.
BOUNDS	DSECT	RF	Array bound (8 bytes).
RGDSECT	DSECT	R6	Chain header (2 bytes).
LVLSTACK	DSECT	R7	Entry in structure information table (8 bytes).
FDTBENT	DSECT	R3	DEFAULT entry (4 bytes).

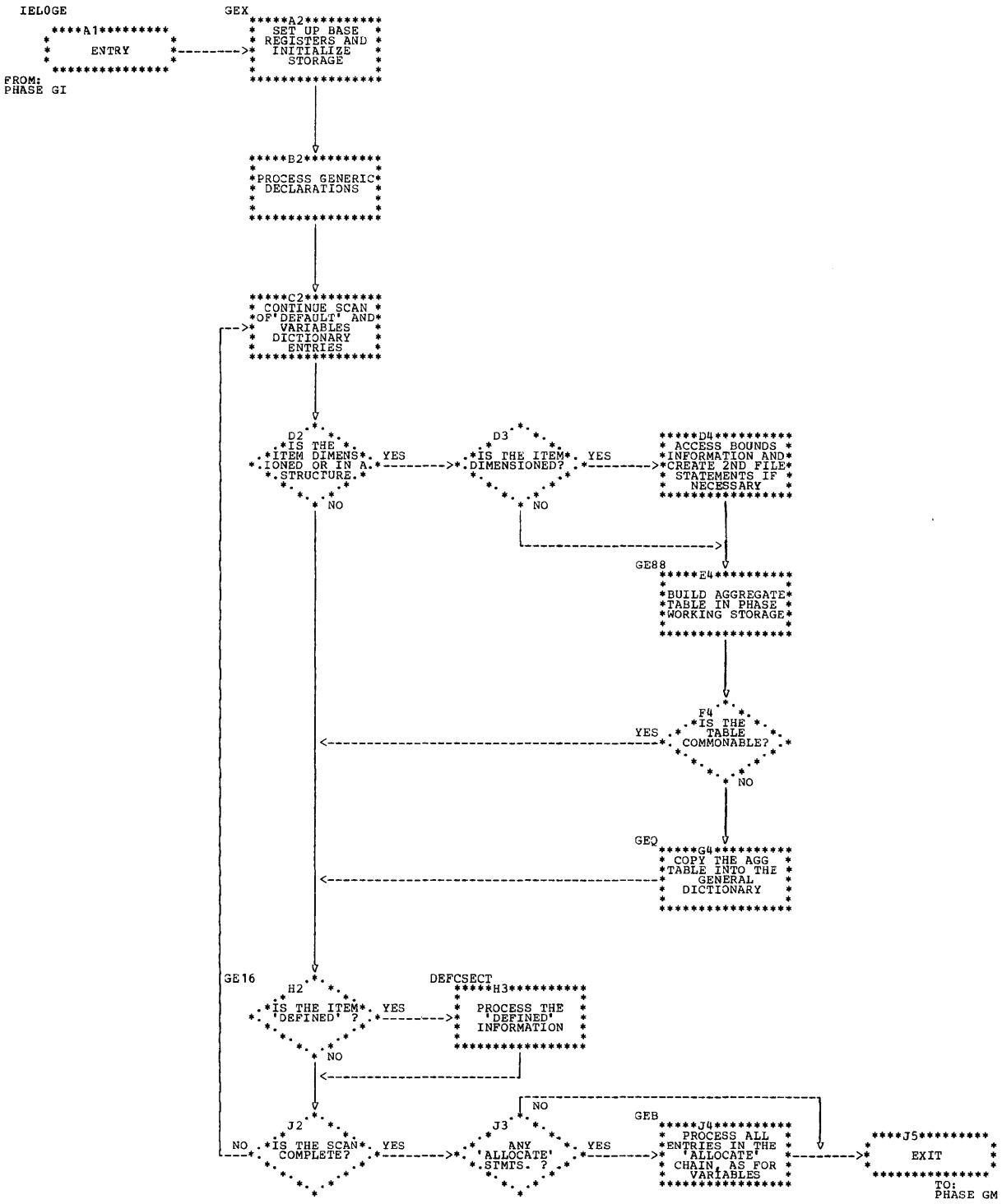


Chart 3.11. Declaration Expressions Phase (Phase GE)

Implicit Declaration (Phase GM)

Name	Type	Base registers	Function
IEL0GM	CSECT	R8,R9	Entry point to Phase GM.
GM	R		Initialize registers, storage, etc.
GMTRTA	R		Scan the text for the next implicit declaration, and pass
GMTRT	E		control to the relevant statement-body processing routine.
GMPMNS	R		Process prefix minus.
SYSNOL	R		
GMNSTD	R		Set bit 8 of GMFLG on.
GMEPROG	R		Output end-of-text marker. Initialize 2nd file scan if
GM2END	E		not already processed, or else exit from this phase to IA.
GM1BT	R		Output item to Type 1 text stream.
GMSL	R		Test statement type, and branch to relevant processing routine.
GMPROC	R		Initialize processing of PROC/BEGIN/ONB statement.
GMOPEN	R		Initialize processing of OPEN statement.
GMCHECK	R		Initialize processing of CHECK statement.
GMENTRY	R		Initialize processing of ENTRY statement.
GMASN	R		Initialize processing of ASSN statement.
GMGEN	R		Common label-processing routine.
GMGOTO	R		Initialize processing of GOTO statement.
GMEOPG	R		Process end-of-page marker.
GMSC	R		Process end of OPEN statement and make dictionary entry.
GM2SC	R		Prepare for 2nd file processing.
GMLPAR	R		Process parenthesis.
GMRPAR	E		
GMPRD	R		Process period.
GMPARD	R		Process argument.
GMNAM	R		Resolve a name.
GM810	E		Set type byte.
GM811	E		Create and output a 6-byte reference for the name.
GMSTRUC	R		Output descriptions of base elements, process structure member.
GMIMP	R		Make an implicit declaration.
GMASGN	R		Process question mark.
GASLVL	R		Process second level marker.
NOTINT	R		Process non-integer constant.
STRCH	R		Process string constant.
GMCOND	R		Process a condition name.
GMPIC	R		Process picture table.
GM2FILE	R		Scan second file and process next item of interest.
GM2PROC	R		Process a second file PROC statement.
LAIGEN	R		Generate label array initialization statements.
GMAGASSN	R		Process a second file AGASSN statement.
GMINIT	R		Process a second file INASSN statement.
GMSTRL	R		Process a second file STRL statement.
GMOFFS	R		Process a second file locator qualifier statement.
GMBPSED	E		
QULEXT	R		Place a locator qualifier expression in 2nd file output.
TXT2OUT	S		Output the start of a second file statement.
OUT2TST	S		Initialize the second file output stream and/or output a block header if necessary.

Continued on next page

XINIT	CSECT	R9	)
XRFSEQ	CSECT	R9	)
XRFAB	CSECT	R9	)
XDIREC	CSECT	R9	) XROUT
XBREAK	CSECT	RF	)
XTXPG	CSECT	R9	)
XSRCH	CSECT	R9	)
XSTG	CSECT	RA	Private storage for Phase GM.
ARGENT	DSECT	R3	6-byte entry in ARGSTK for an argument.
LBARINIT	DSECT	R2	13-byte label array initialization statement.
DFTBENT	DSECT	R2	4-byte entry in DFTAB for a DEFAULT specification.

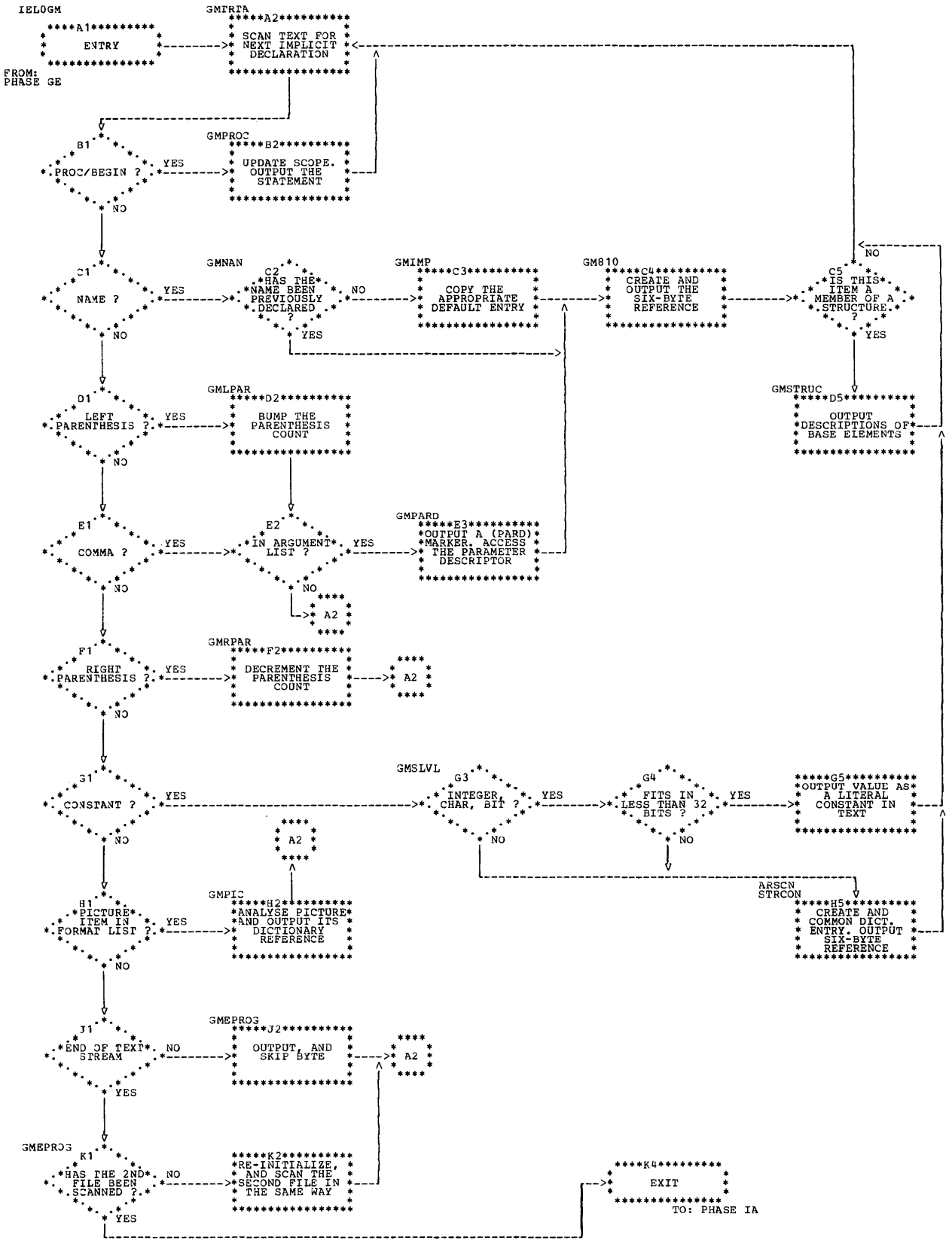


Chart 3.12. Implicit Declarations Phase (Phase GM)

Merge Declaration-expressions (Phase IA)

Name	Type	Base registers	Function
LOCSTK	DSECT	R3	Stack for implicit locator chain.
IEL0IA	CSECT	R5-R9	Entry point to Phase IA.
IA	R		Initialize registers and stacks. Call subroutine SCAN.
MT28	R		End-of-program routine.
BUMOUT	S		Call main text output routine, and increment input text pointer R1.
MTOUT	S		Main text output routine.
MT2	S		Process a statement header in the main text stream, and branch to the relevant processing routine.
MT25	S		Process a block-heading statement.
SCAN	S		Main processing routine. Scan the main text stream and declaration expressions file, processing and inserting as required.
SC1	R		Declaration expressions file statement processing routine.
SC2	R		Test statement header type and branch to MT2 if not 2nd file.
SC21	R		Branch to relevant 2nd file statement-processing routine.
SF1	R		Partially process a PROC statement.
SF13	R		Scan through all alignment chains, putting out MAP operators.
SF2	R		Process a locator expression.
SF3	R		Process an extent expression.
SF4	R		Process an INITIAL assignment.
SF47	S		Output fixed INITIAL assignments.
SF5	R		Process a MAP statement.
SF6	R		Process a defined addressing statement.
SF8	R		Complete the prologue code relating to a declaration expression by outputting the INITIAL assignments.
SF84	S		End of current prologue code has been reached. Reset input text pointer, and process the main text stream.
SC3	R		Operand analysis and locator-chain construction.
STACKA	R		Stack an AREA associated with an offset.
SC5	R		Defined variable routine.
SC51	S		Examine type of defining.
SC54	S		Process iSUB defined operand.
SC6	S		Process string overlay defined expressions.
SC65	S		Process based-addressing defined expressions.
SC66	S		Process POS (or DSUBS) expressions.
RF0	R		Examine and act on the results of a REFER scan of a statement.
RF1	S		Prescan action on finding a REFER item.
RF2	S		Prescan action on finding an operand which is not a REFER item.
RF3	S		REFER scan of text.
RF31	S		REFER item found.
RF7	S		REFER scan action on finding change of statement level.
RF9	S		All REFER items having been mapped, reprocess the whole statement, thus qualifying all the REFER items in the statement.
FU1	R		Analyze a user function, BIF, or pseudovariable.
FU2	R		Process the STRING BIF.
FU3	R		Process the UNSPEC BIF.
FU4	R		Process the DIM, HBOUND, and LBOUND array manipulation BIFs.
FU5	R		Process the REAL and IMAG BIFs.
FU9	R		Output a function and branch to operand processing routine.

Continued on next page

SKIPOVER	S		Skip over input text to an extent decided by the setting of SPECSW.
AL1	R		Process an ALLOCATE statement.
PSUD	R		General purpose routine to output up to 20 bytes of information.
IMP	R		Imply the locator for the SET option in an ALLOCATE statement.
IMPAR	R		Imply the area applicable for a given FREE statement/ALLOCATE statement.
ACC2	R		General routine for random second-file access.
SETOFF	R		Routine to generate assignments to set up any OFFSET expression.
MEM	S		Analyze an operand to see if it is a structure member. If so, access the current major structure entry.
OFSOUT	R		Construct and output OFFS pseudo text table.
RESCON	R		Construct RESDES pseudo text table.
GENMAP	S		Access a statement in a given alignment chain, update the chain slot, and output mapping operators.
GDAS	S		Generate locator/descriptor assignments.
HOWFA	S		Determine how far a REFER structure should be mapped.
PSEUDO	S		Output pseudo text tables (e.g., OFFS, RESDES).
OFAS	S		Generate an OFFS/ASSN pair to set up the bounds slot in the descriptor.
MAPO	S		Output a MAP pseudo text table.
RFAB	S		Access any dictionary entry.
TABS	S		Convert a given text reference to an absolute address, and fetch the appropriate page into main storage.
TB2	S		Routine used for second-file page accessing.
TABS3	S		Routine used for accessing output text pages.
TREF	S		Convert an absolute address in R1 to a text reference.
DELEAT	R		Delete any statement.
IAER1 to			
ERR4	R		Error routines.
MISCER	R		Error routines.
MISCER	R		General purpose error routine.
FREDI	R		Free one dictionary page to allow access for the error page.
XINIT	CSECT	R9	)
XBAKTX	CSECT	R9	)
XMESGR	CSECT	R9	)
XDISC	CSECT	R9	)
XRFAB	CSECT	R9	) XROUT
XDIREC	CSECT	R9	)
XBREAK	CSECT	R9	)
XTXPG	CSECT	R9	)
XBRIC3	CSECT	R9	)
XSTG	CSECT	RA	Private storage for Phase IA.

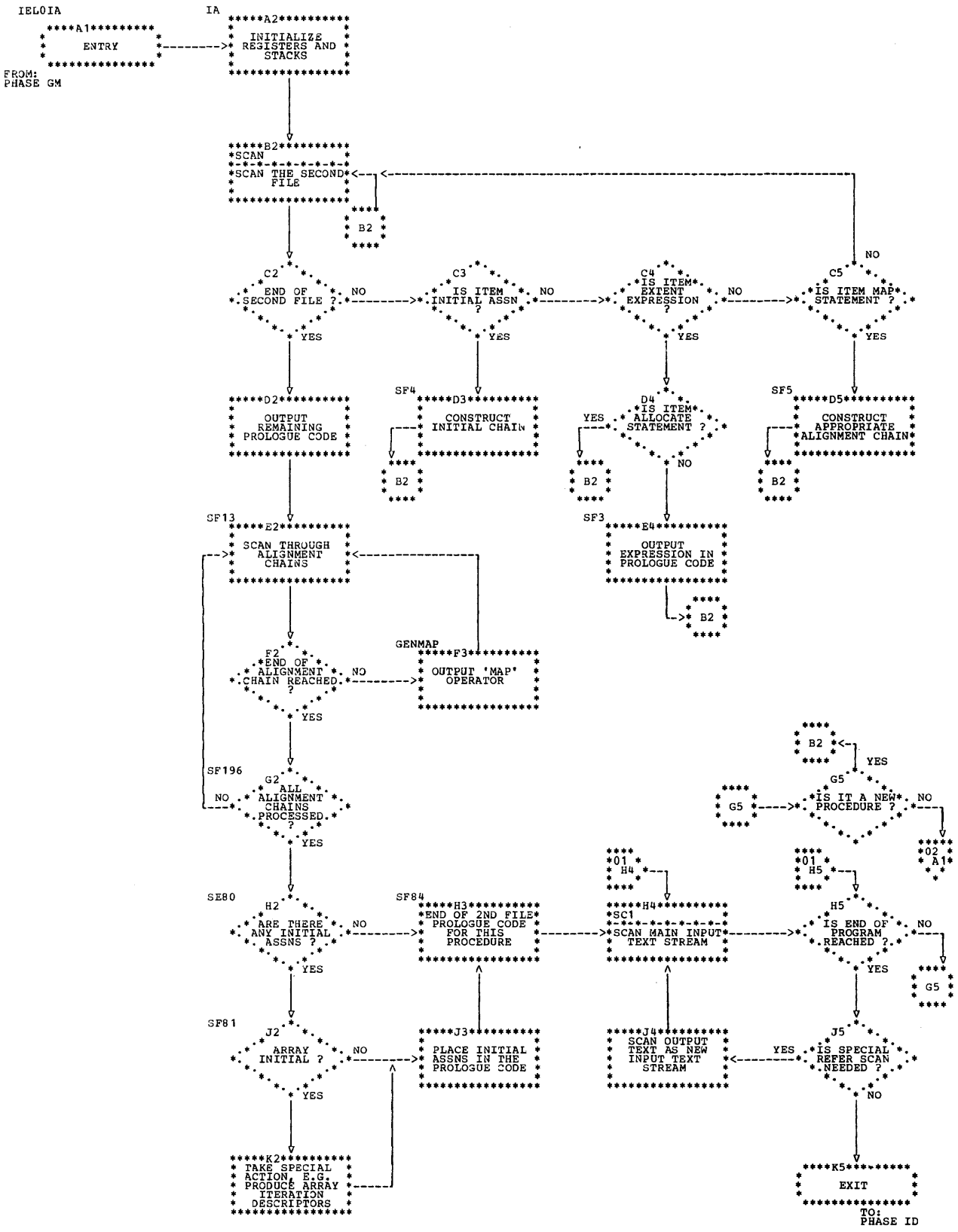


Chart 3.13. (Part 1 of 2). Merge Declaration Expressions Phase (Phase IA)

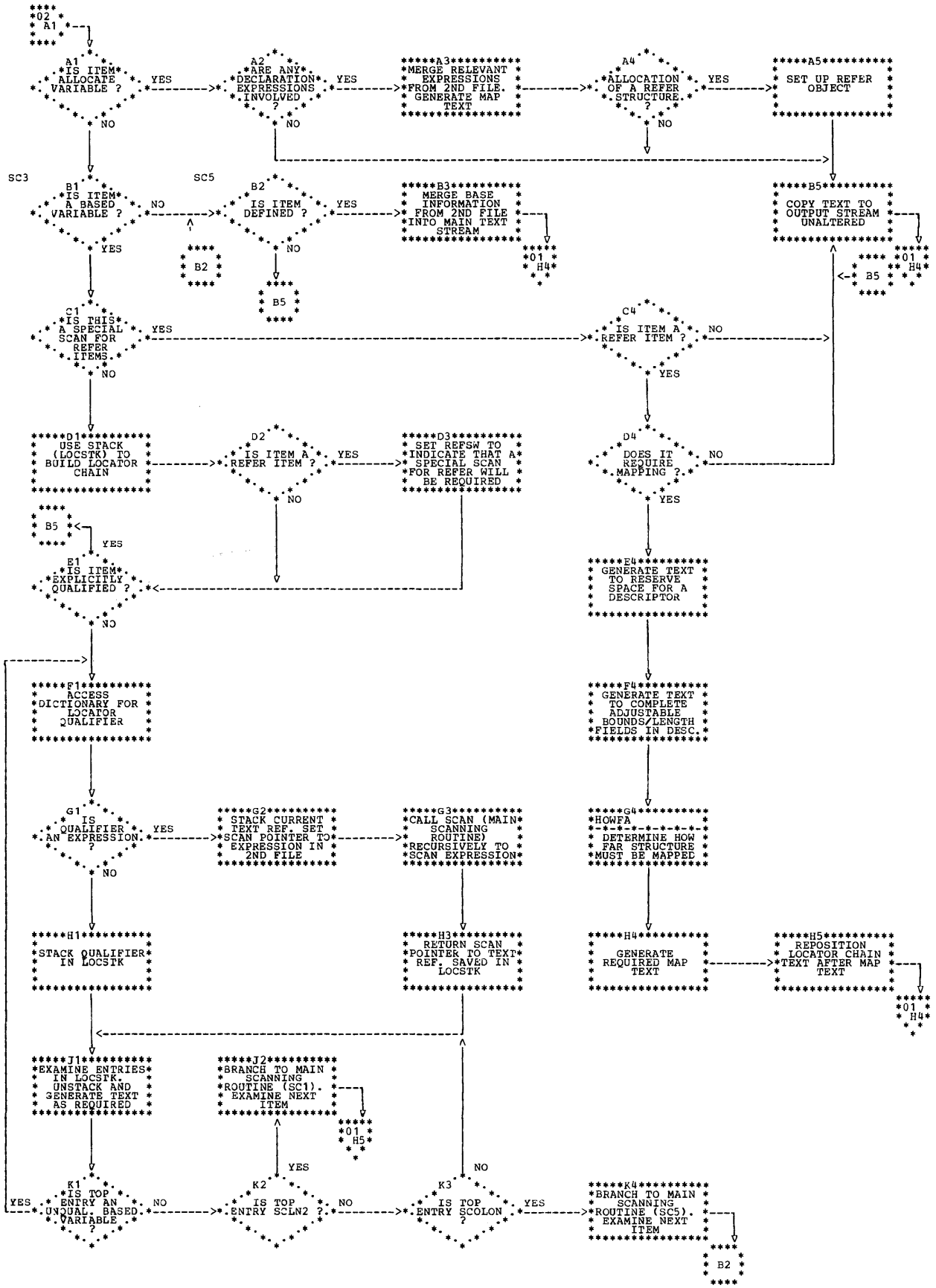


Chart 3.13. (Part 2 of 2). Merge Declaration Expressions Phase (Phase IA)

Matching of Data-aggregate Arguments (Phase ID)

Name	Type	Base registers registers	Function
DSTAK	DSECT	R3	Stack for nested-function entries.
IEL0ID	CSECT	R6-R9	Entry point to Phase ID.
ID	R		Initialize registers, storage, etc.
QSCAN	R		Main scanning routine. Scan the input text stream, and transfer text immediately to the output stream unless special action is needed.
QSUBS	R		If a subscript list has been found, check that the number of subscripts is correct and that all the subscripts are scalar.
ARGLIST	R		Start of a parameter/argument list found. Increment the stack pointer to the next level.
NEXTPA	R		Scan to the next parameter/argument. If a parameter is present, store information.
NEXTARG	R		Process the next argument.
OUTARG	R		Output an argument. At this entry point to the routine, the argument is output with slight changes, no temporary operand being required.
OUTARG5	E		Special entry point to OUTARG for when a structure reduced-descriptor has been created.
OUTARG8	E		Entry point to OUTARG to output the argument unaltered.
COMP1	R		The argument is a single (possibly subscripted) variable. Compare it with the parameter.
TEMP1	R		Create an aggregate temporary in text and dictionary.
ARGEND	R		Complete the outputting of an argument. Output a temporary operand if required.
ARGEND3	S		Test for the end of an argument list.
ARGEND5	S		If found, go back to the previous level in the stack and continue processing.
ABIF	R		BIF processing routine. At this entry point, process an array BIF.
ABIF2	S		Process ALL and ANY BIFs.
ABIF3	S		Process SUM and PROD BIFs.
ABIF5	S		Process the STRUCTURE BIF.
POLYCODE	R		Process the POLY BIF.
SCANEXP	R		Scan an expression (e.g., an argument or a subscript) to see if it is structured and/or dimensioned. Take the appropriate action if so.
KMDE	R		Make dimension entries if a dimensioned variable is found in an argument.
SKIPOVER	R		Skip over input text to an extent decided by the setting of SPECSW.
IDERR0 to IDERR19	R		Error routines.
ERTN	S		Produce an error message, and delete the incorrect statement.
PUTHOP	R		Scan and output text.
BRIC	R		Increment the input pointer and handle pages in the text stream.
ACCGD	S		Acess a general dictionary entry.
ACCVD	S		Access a variables dictionary entry.
LUKPOS	S		Skip over DSUBS/POSX expressions, if present.
CGNGHD	S		Re-access a statement header in the output text stream, and set bits in the PREFIX option byte.
SKSTRUD	S		Skip over a STRUD list, if present.
OUTSTRUD	S		Output a STRUD list.

Continued on next page

XBREAK	CSECT		)
XBRIC1	CSECT		)
XINIT	CSECT		) XROUT
XTXPG	CSECT		)
XDIREC	CSECT		)
XSTG	CSECT	RA	Private storage for Phase ID.
XTXEQU			)
ZTEXT			)
YDICT1			)
YDICT2			) Copybooks invoked.
XCOMM			)
XEQU			)
XBIF			)

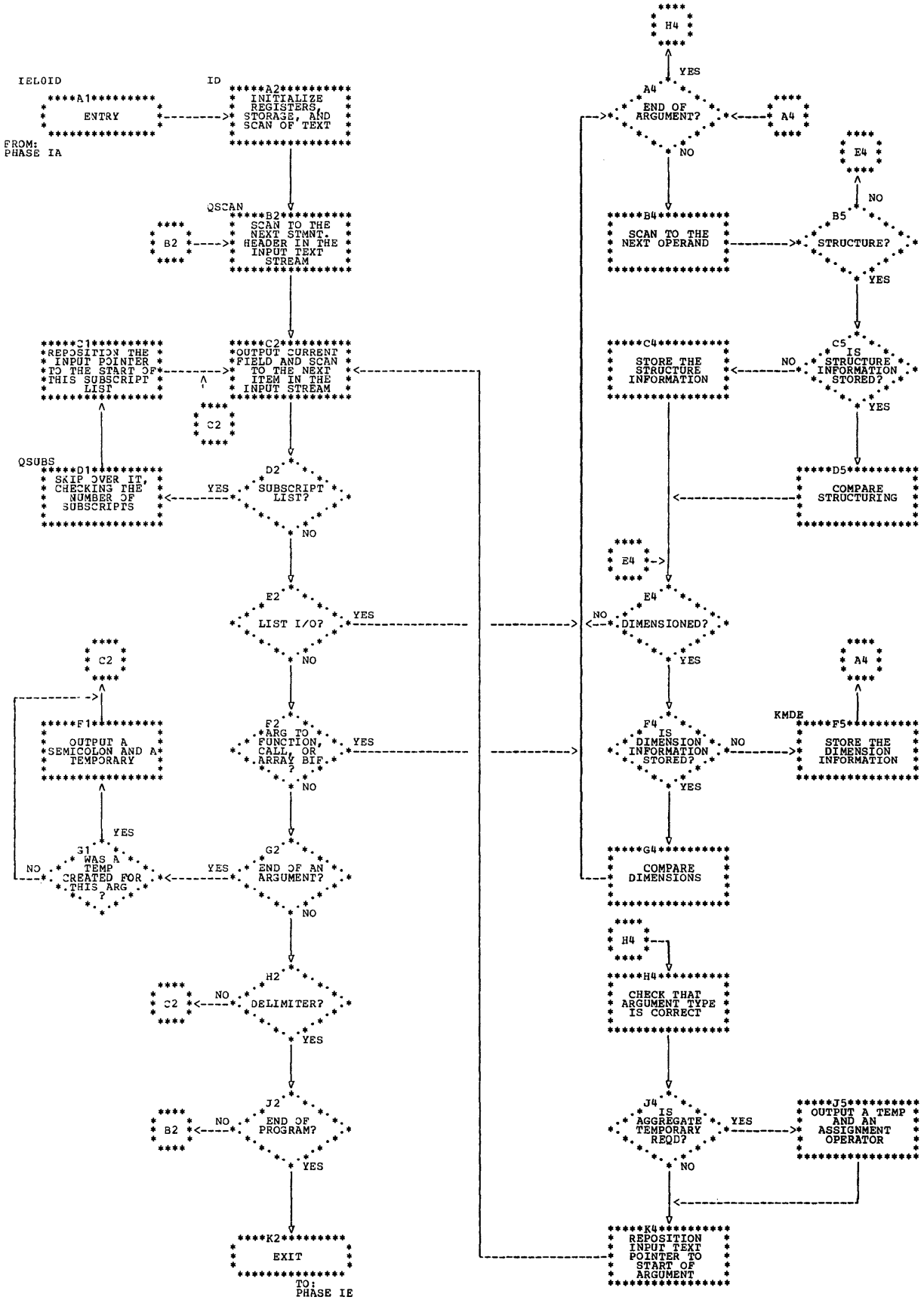


Chart 3.14. Aggregate Argument-matching Phase (Phase ID)

Name	Type	Base	Function
DSTAK	DSECT	R3	DSECT for all the stacks used.
MEMENT	DSECT	R9	Description of member entry in structure stack.
DIMENT	DSECT	R2	Description of dimension entry in structure stack.
BOUNDS	DSECT	RF	Description of bounds entry in aggregate table.
IEL0IE	CSECT	R5 to R8	Entry point to the main module, IE1.
TSTPROC	R		Analyze type of statement.
OUTHDR	R		Output complete statement if it is one of PROC, END, BEGIN, ON, or ENTRY. Otherwise output the statement header only.
TSTASS	R		Check for assignment and stream I/O statements.
NOTOFINT	R		Examine each operand, in turn, in all statements other than assignment and stream I/O statements.
NIDE	R		Check for illegal use of aggregates in OPEN statement.
NIOPEN	R		Check for illegal array in OPEN and GOTO statements.
FUNEST	R		Look for embedded aggregate assignments within the arguments generated by Phase ID.
BYNAME	R		Calls the BYNASN subroutine to process statements containing a structure assignment with the BY NAME option, and checks for any error conditions raised by the subroutine.
BYNERR	R		Deal with the error condition raised by the illegal use of the BYNAME option.
IOSET	R		Handle stream I/O statements.
FRST	R		Process and analyze assignment statements.
SCALPUT	R		Process and analyze element variable assignment statements and pseudovariable assignment statements.
STRUCASS	R		Process structure assignment statements.
OPRTR	R		Analyze second and subsequent operands in structure or element-variable assignment statements.
CHKSP	R		Analyze second and subsequent array operands in structure assignment statements.
ATRII	R		Analyze second and subsequent structure operands in structure assignment statements.
ZDIM1	R		Output structure assignment statements; modify the statement header if required.
ARAS	R		Process array assignment statements.
AAB	R		Text scan after the master operand in array assignment statements.
OUTII	R		Output array assignment statements; modify the statement header if required.
ARRII	R		Analyze second and subsequent array operands in array assignment statements.
IERR1	R		Error handling routine.
LUKPOS	R		Look for DSUBS/POSX lists in text.
PUTCHAIN	R		Called when outputting a pointer-chain in the expansion of a structure expression/assignment. Calls the routine PCHRTN to actually process the pointer-chains.
COPYSUB	R		Output SUBS lists.
RESET	R		Reset text pointer back to a given point within a page.
DISC	R		Build a table of all discardable input text pages.
REFSV	R		Convert an absolute address into a 5-byte text reference.
XSTG	CSECT	RA	Private storage for module IE1.
IE2	CSECT	R5,R6, R7	Second module of Phase IE. Contains all the sub-routines called from the main module, IE1.
BLDOUT	S		Build a 7-byte operand from the STRUDs and the major structure operand in text, in the 7-byte field IAREA, to be output.

Continued on next page

GENSUB	S		Generate subscripts in expansion of assignments for array and structure operands.
CHNGHD	S		Access the header of the statement currently being processed (in the output text stream) and modify it with the contents of the 1-byte slot IOP.
MDE	S		Make entries in the dimension stack.
MME1	S		Make first member entry in the structure stack and carry out other initializations for housekeeping, etc.
MME	S		Make the second and subsequent entries in the structure stack.
DLUP	S		Output DOTMP and ENDIT.
PUTHOP	S		Output and jump over text.
BRTN	S		Output and/or skip over a subscript/argument list.
ACCGD	S		Access the aggregate table entry for an array or a member of a structure.
SCANSUBS	S		Scan the subscript list, analyze it, and store the necessary information to be used for checking and outputting all labels starting with SUB.
STKRTN	S		Initialize stack pointers and bump the stack pointers up to the next higher level.
GDE	S		Output DOTMPs and ENTMPs.
STAKAR	S		Access the dictionary entry for an array operand. Make dimension entries in the stack if not subscripted or if asterisks present in SUBS list. Check number of subscripts.
SKSTRD	S		Skip over STRUDs in the text stream. On the first text scan, store the level increment and the dimensions of the major structures from the counters (LINC and MID respectively) into the first two bytes of the spare STRUD. Also count the number of base elements and store the count in the third byte of the spare STRUD. On the second and subsequent text scans load the three respective counters from the text.
OPSKP	S		Skip over a structure operand and DSUBS/POSX expressions if they are present in the text, and set the input pointer (R1) to point to the first STRUD. Used in the expansion of structure expressions only.
COMDED	S		Compare the DEDs in the STRUDs of two structures.
BYNASN	S		Handle BYNAME structure assignments.
ACCVD	S		Access variables dictionary. Used in BYNAME structure assignments only.
REINIT	S		Reinitialize the text (structure operands) in a nested function when the function is in a structure assignment.
PSCHRTH	S		Handler pointers, etc. in the text.
ERTN	S		Called when the phase finds an error in the PL/I source program. It deletes the statement from the text and passes a message to be processed by the error editor (Phase UA).
XBREAK	CSECT		)
XRFAB	CSECT		)
XTXPG	CSECT		)
XMESGR	CSECT		) XROUT
XDISC	CSECT		)
XRFSEQ	CSECT		)
XDIREC	CSECT		)
XSTG	DSECT	RA	Private storage for module IE2.

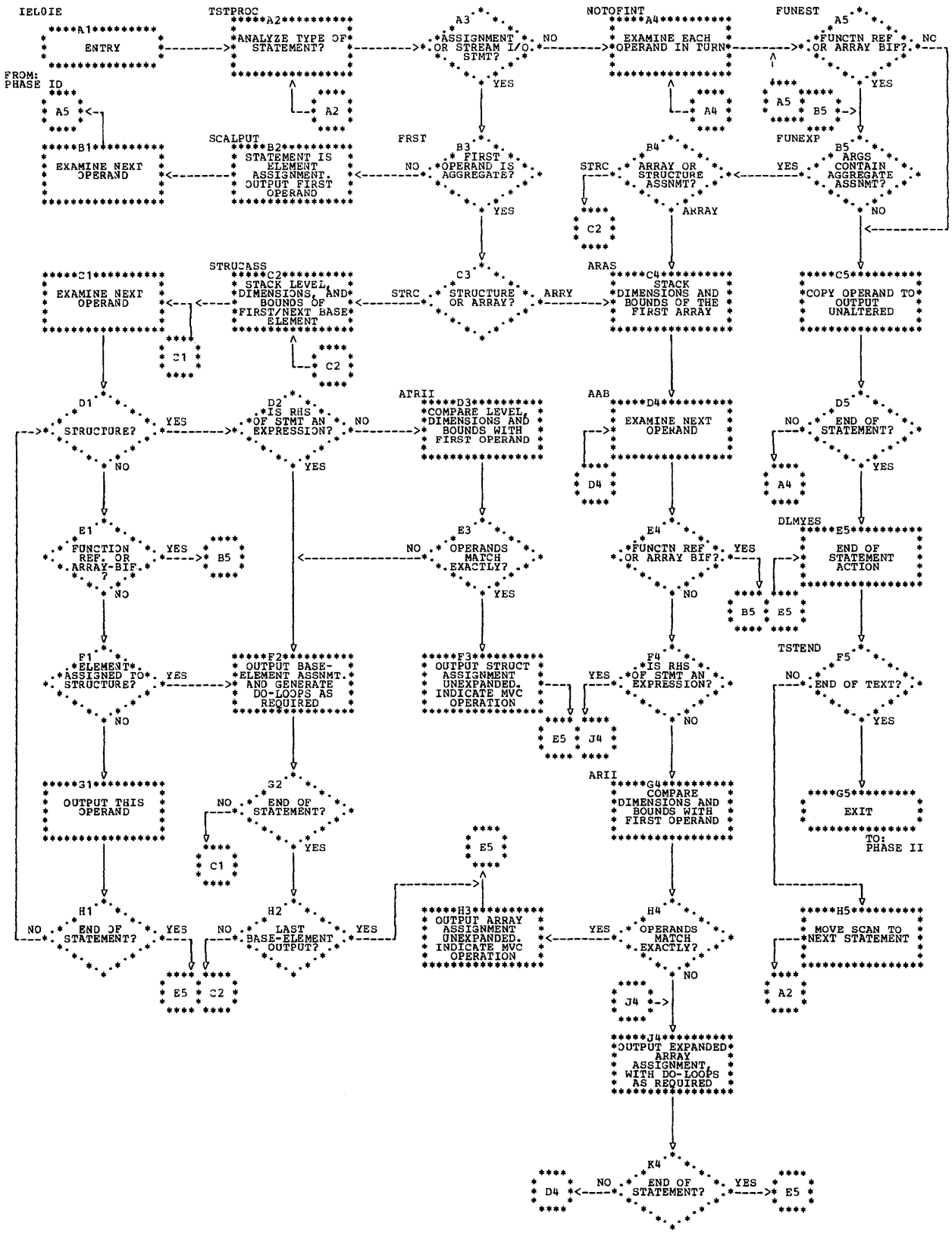


Chart 3.15. Aggregate-expression Phase (Phase IE)

Expression Analysis and Text Formatting (Phase II)

Name	Type	Base registers	Function
IEL0II	CSECT	R6,R7, R8,R9	Entry point to root module of Phase II.
II	R		Initialize registers, storage, and text streams.
IEL0II3	CSECT		Translate tables used in II.
TG1	R		Basic text scanning routine.
TG23	R		End-of-program routine. Transfer control to next phase.
TG24	R		Deal with pseudo-text-table input, created by Phase IA.
TG5	R		Generate a statement header table.
TG51	R		Generate PROC and BEGIN text tables.
PENDO	S		Output a PEND text table.
UNSTK	S		Delete an MSTACK entry.
CHIP	R		Input text stream handling routine.
TG6	R		Compare the text operator priority with that of the top operator in MSTACK, and take appropriate action.
TG8	R		Process MSTACK further, and continue scan.
TG89	S		Make stack entries for 'non-standard' operators.
BIFQ	S		Test for a BIF operand, and make relevant stack entry.
SUPP	S		Determine when output should be suppressed.
TGOUT	S		Output up to four text tables.
NULTAB	S		Output NULL text tables.
TG7	R		Process operators, take any special unstacking action required (as indicated by the following routines), and generate text tables.
TGS1	R		Special action for IF operator
TGS2	R		" " " ELSE "
TGS3	R		" " " THEN "
TGS4	R		" " " TO,BY,DOEQ "
TGS5	R		" " " ITDO "
TGS6	R		" " " WHILE "
TGS7	R		" " " ENDIT "
TG76	R		" " " assignments.
TGS8	R		" " " LPAR operator
TGS9	R		" " " DO "
TGSA	R		" " " SUBS1/SUBS "
TGSA3	R		" " " SBAR "
TGSB	R		" " " PREFIX "
TGSC	R		" " " DO TEMP "
TGSC1	S		" " " ENTMP "
TGS97	R		" " " ENDDO "
TGR	R		" " " I/O and system-interface stmts.
TGSD	R		" " " arguments.
TGCAFU	R		Special action for CALLS/function operator.
TGBIFA	R		" " " BIF arguments.
TGSIG	R		" " " SIGNAL, REVERT operator.
TGPTSF	S		" " " PTSAT "
TGSE	R		" " " GOTO, GOOB, MAP, TASKO, etc.
TGSE2	R		" " " FIT,FITE text tables.
TGPOS	R		" " " POS expressions.
TGSG	R		" " " WHEN(SELECT) operator.
TGSH	R		" " " SELECT operator.
TGSI	R		" " " OTHERWISE operator.
TGSJ	R		" " " END(SELECT) operator.
TGSK	R		" " " stacked SELECT operator.
TGGEN	R		" " " GENERIC arguments.
TGCF	R		" " " COMPLEX operator.
TGONX	R		" " " ON statements.
FITEO	S		Output FITE text tables.
OUSTAK	S		Stack-control subroutine.
NXVLABL	R		Generate compiler labels (GSLs).
GDC	S		Generate DO conversion.

GSNGEN	S		Output extra statement header.
INVF	R		Invoke O-ARG functions.
POPRTN	R		Pop the main stack.
NDXGEN	R		Generate NDX text tables.
TGARG	S		Unstack an argument.
TREF	S		Save the current text reference for a statement rescan.
PRITAB	T		Operator priority table.
TRTAB	T		Table for statement-type special action.
GENTAB	T		Generation action table.
RIOT	T		Translate table for record I/O statements.
EA	CSECT	R6,R8, R9	Expression analyzer module of Phase II.
EA	S		Expression analyzer subroutine.
EA1	R		Call EA and return to TG75 in root module, II.
EA2	R		Process assignments.
CVCHEK	S		Arithmetic operation checking routine.
EA4	S		Analyze information provided by subroutine CVCHEK.
CONEXP	S		Evaluate result for fixed exponentiation.
TEMPAN	S		Construct temporaries.
CNTAR	S		Generate intermediate temporary operands when assignments are made to string targets.
TRUDED	S		Construct a standard DED for a literal constant
LIEDED	S		Restore DED for a literal constant.
SCALP	S		Calculate scale and precision.
CONC	S		Convert arguments to the correct string type.
BIFE	S		Examine BIFs.
ACHE	R		Analyze arguments and generate conversions where necessary
FRED	R		Function result determination routine.
CRUD	S		Check a result for user compatibility.
SELECT	S		Perform a GENERIC selection.
BARD	S		Convert the usual data byte of a GENERIC argument to the 2-byte form used in the argument descriptor.
QSRCH	S		Determine the code byte corresponding to a Q-temp.
MATCH	R		Argument matching routine.
MAERR	R		Error message routine.
BIFERR	S		BIF error message routine.
MISCER	S		General purpose error message routine.
DELEAT	S		Perform "tidy-up" action after error situation detected.
DELR	R		Output DEL text tables.
GNPSV2	S		Output a pseudovariable text table for a non-string type pseudovariable (PSV2).
RFAB	S		Access any dictionary entry.
FUNTAB	T		Function table for BIFs.
FAT	T		Function access table. Used to determine the offset of any particular BIF.
XINIT	CSECT	R9	)
XMESGR	CSECT	RF	)
XBREAK	CSECT	RF	)
XTXPG	CSECT	R9	)
XBRIC1	CSECT	R9	) XROUT
XBAKTX	CSECT	R9	)
XRFAB	CSECT	R9	)
XDIREC	CSECT	R9	)
XSTG	CSECT	RA	Private storage for Phase II.

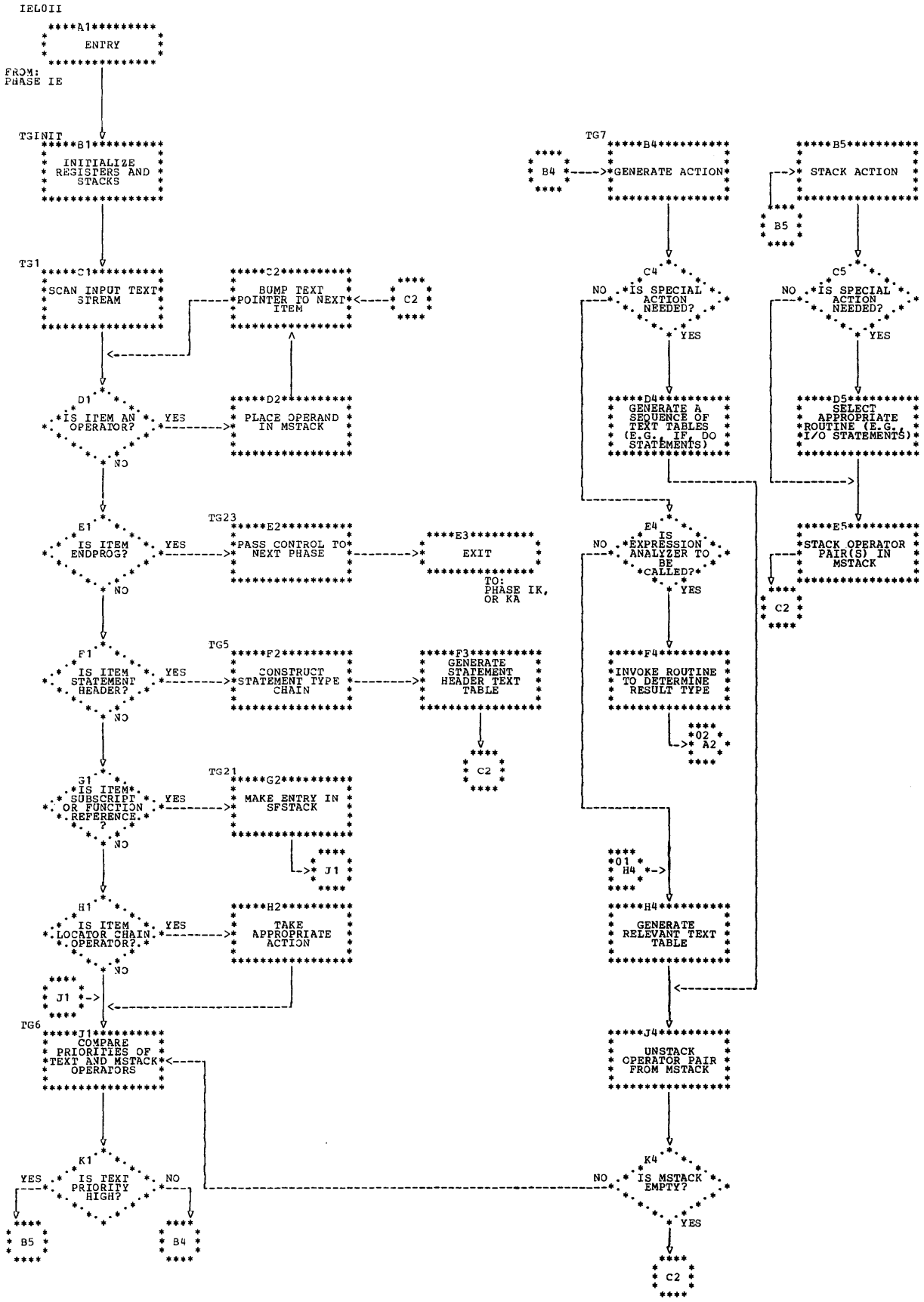


Chart 3.16. (Part 1 of 2). Expression Analysis and Text Translation (Phase II)

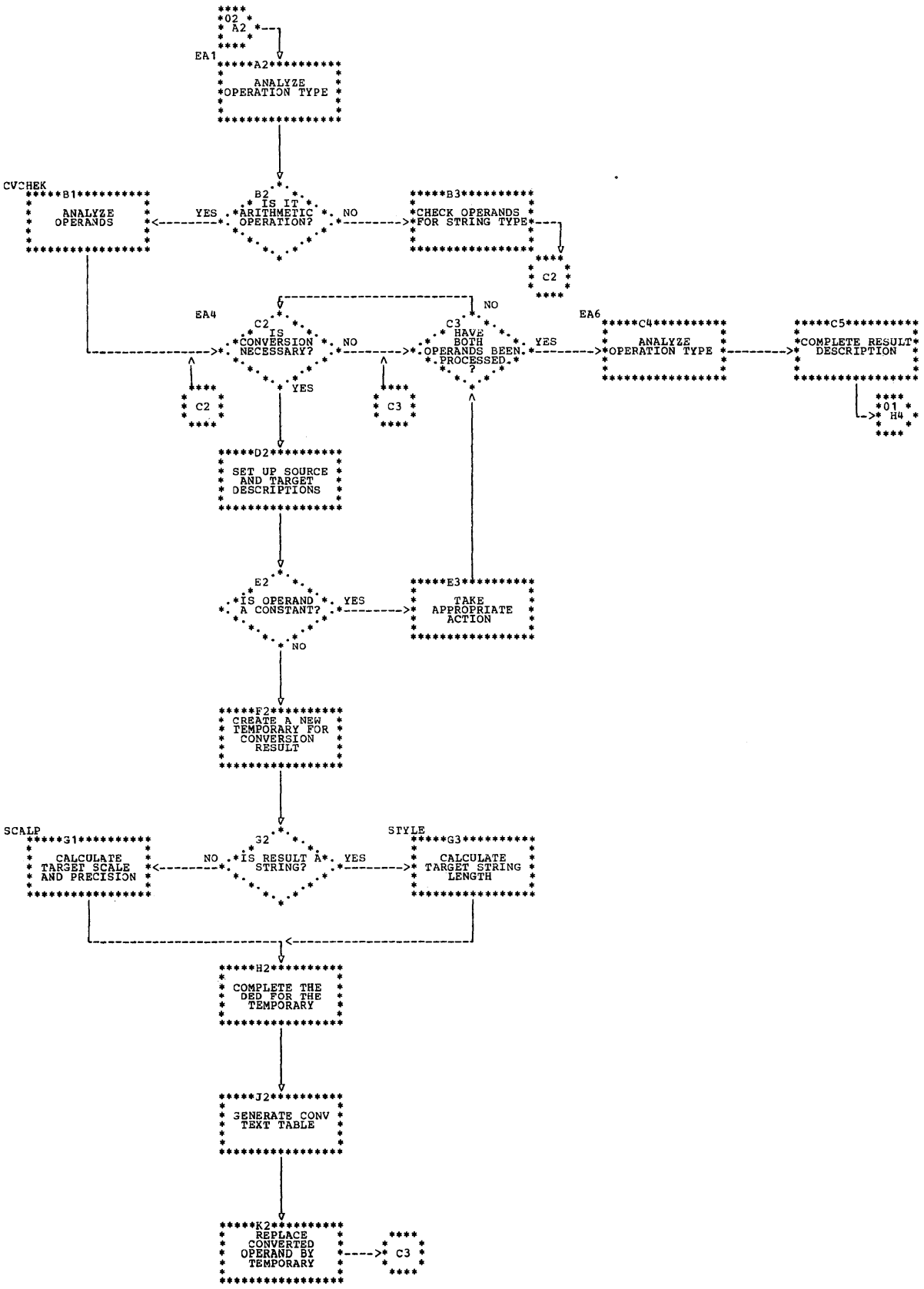


Chart 3.16. (Part 2 of 2). Expression Analaysis and Text Translation (Phase II)

Attributes and Cross-reference Listing (Phase IK)

Name	Type	Base registers	Function
			Continued on next page
IELOIK	CSECT	R9	Entry point to Phase IK.
IELOIK	R		Phase initialization.
IK05	R		Pick up the source text for scanning.
IK04	R		Pick up operands and see if they are variables.
IK13	R		Having identified a variable, go to its variables dictionary entry to access its name table reference.
IK21	R		Take the text reference of the next available cross-reference entry and insert it (a) in the previous cross-reference entry (if any), and (b) in the name table entry.
IK40	R		Copy names dictionary entries into spare text pages in preparation for sorting.
IK43B	S		Keep a count of the number of names dictionary entries copied.
SORT	R		Sort the names dictionary entries on the text pages into PL/I collating sequence. A SCHELL sort is used.
IK50	R		Initialize the printing of the information collected.
IK60	S		Access the variables in the order now given by the sorted text pages. Analyze each entry in turn.
IK90	R		At this point the status is as follows: 1. The identifier and its declaration number (if any) have been put out. 2. If the ATTRIBUTE option has been specified, the appropriate attributes have also been put out. This routine tests to see if cross-references are required. If not, the next item in the sorted list is accessed. If cross-references are required, subroutine PXR is called to output them.
PUTID	S		Pick up the identifier's declaration number (if any), convert it to external form. Also pick up the identifier itself and convert it to external form. Insert both items in the print line.
PUTAT	S		Accept an attribute and its length as parameters. Check for room in the current print line: if none, output that line and commence a new one.
PUTXR	S		Accept the address of a 2-byte cross-reference as a parameter. Convert it to external form and left adjust it. Check for room in the current print line: if none output that line and commence a new one.
QPXR	S		Print out the cross-references for each identifier.
PXRBIF	S		Identical to PXR except that dictionary references are used instead of text references.
PRNTCON	S		Deal with printing constants contained in parenthesis.
PRNTCON	E		Entry point to subroutine PRNTCON. Accept a parameter addressed by RB. The parameter is a binary number preceded by its length which can be one or two bytes. Output is of the form (n).
PRNTCON1	E		Entry point to subroutine PRNTCON. Back up the position in the line by one, insert a comma, the number itself, and a right parenthesis. Thus output is of the form (n,m).
			Continued on next page

XINIT	CSECT		)	
XTXPG	CSECT		)	
XBRIC2	CSECT		)	
XRFAB	CSECT		)	
XSEQRT	CSECT		)	XROUT
XPRNTN	CSECT		)	
XPRNT	CSECT		)	
XDISC	CSECT		)	
XDIREC	CSECT		)	
XSTG	CSECT	RA	Private storage for Phase IK.	

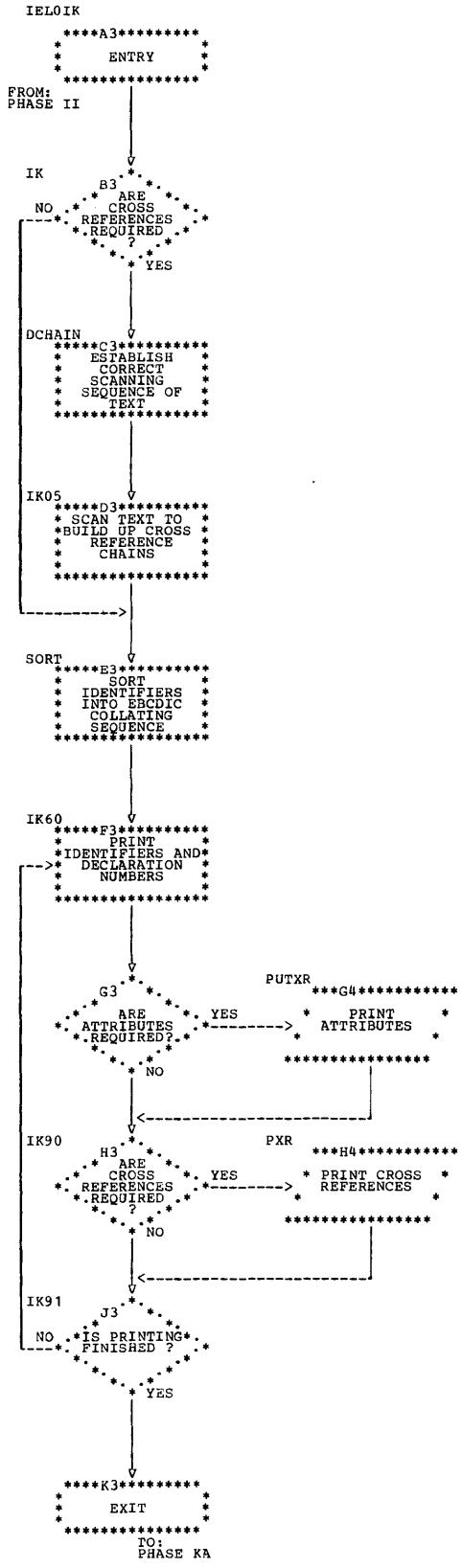


Chart 3.17. Attribute and Cross-reference Listing Phase (Phase II)

Name	Type	Base registers	Function
XTXEN	CSECT	R7	CSECT for routine XTXEN.
XTXEN	R		Invoked by the XTXEN macro when the text table required is on a new page. It obtains the page specified by the macro and checks that it is the right one.
IEL0KA	CSECT	R4,R5 R6,R7	Entry point for Phase KA.
KA1	R		Initialize registers, storage, and overflow space.
KA10	R		Overflow page allocation.
BOSS	R		Initialize text scanning.
BS1	S		Examine text table.
CH0	R		Analyse statement to see whether or not it should be placed in one of the statement type-chains.
CH1	S		Update XSYSCH field in XCOMM.
CH2	S		Update XDOCH field in XCOMM.
CH3	S		Branch to PROCESS routine.
CH4	S		Update XRIOCH field in XCOMM.
CH5	S		Update XSIOCH field in XCOMM.
CH6	S		Update XSUBCH field in XCOMM.
CH7	S		Update XBELCH field in XCOMM.
BS4	S		Examine text table for special action cases.
CHAINIF	R		Reset switches.
PROCESS	R		Process IF statements (including WHILE clauses).
IFMIN	R		Optimize MINUS text table.
QUERY	R		Call the subroutine CONVPROC.
COMPR	R		Process IF statements that have a single comparison operator.
LOGEX	R		Process IF statements containing logical expressions.
LOSCAN	R		Scan a logical expression backwards to the next text table.
LOVELY	S		Branch to the processing routine relevant to the text table found.
LOQUER	R		Text table is not COMP, AND, or OR. Process it and delete its stack entry.
LORAND	R		Process an AND or OR text table.
LONOT	R		Process a NOT text table.
LOCOMP	R		Process a COMP text table.
LOBUNG	R		Routine for recovery from abnormal conditions in the LOGEX routine.
LOBOUTT	R		Output a BCB table.
TEXTCH	R		Text housekeeping routine.
CONVPROC	S		Process a non-logical expression table whose result is used in a logical operation.
BZERO	R		If a constant cannot be generated, this routine converts the variable to a bit-string. A test is carried out to ensure that the variable is of a type that can be converted to bit, and the relevant diagnostic generated.
SETTEMP	R		Generate a CONVERT text table.
GOTOQ	R		Test for a THEN GO TO expression.
GOTOY	R		Optimize a THEN GO TO expression.
STRNG	R		This routine performs the following functions: 1. Look at every locator-assignment and if the source and target do not match change the ASSN table into an appropriate BIF table. 2. Set appropriate bits in XOPPHS1 in XCOMM for optional phase loading.

Continued on next page

			3. Look at each operand in turn and set the ALGN/UNAL bit in the data byte of all chameleon temporary operands, variables, and QTs. 4. Detect and resolve chameleon operands. 5. Detect all string-temps and resolve them as required.
IQSET	S		Bit setting for optional loading of Phase IQ (XOPPHS1, bit 1).
KKSET1	S		Bit setting for optional loading of Phase KK (XOPPHS1, bit 2).
OCSET1	S		Bit setting for optional loading of Phase OC (XOPPHS1, bit 3).
OXSETY	S		Bit setting for optional loading of Phase OX (XOPPHS1, bit 4).
	S		Check if the operand is a chameleon. If so, access the variables dictionary entry and fill the text DED from the dictionary.
UNAL1	S		Check if the operand is a data variable. Manipulate the ALGN/UNAL bit in the data byte as required.
S1	S		Replace references to temporaries by their evaluated form (including the string length if known at compile time).
CVAS	S		Convert any source to string.
S32	S		Process CONCAT text table.
S330	S		Process bit-string operations.
S35	S		Analyse and process some STRING BIFs.
FR31	S		Process SUBSTR and BOOL BIFs.
FR32	S		Process REPEAT BIF.
FR33	S		Process HIGH/LOW BIF.
FR34	S		Process TRANSLATE BIF.
FR35	S		Process BIT/CHAR BIF.
STRIMP	R		Construct string temporary operands.
STYLE	R		Calculate target-string length.
SCALP	S		Calculate scale and precision.
TRUDED	R		Provide a standard DED for literals.
ERTN	R		Error-message handling routine.
BIFERR	S		Generate BIF error messages.
MAERR	S		Generate error messages.
SINCH	R		Static initial error-checking routine.
EP	R		End of phase. Test the optional phase loading bits in XOPPHS1 in XCOMM and release control to the appropriate phase.
XINIT	CSECT	R9	)
XNXROUT1	CSECT	R9	)
XNXROUT2	CSECT	R9	)
XTXPG	CSECT	R9	)
XTCH	CSECT	R9	) XROUT
XNSRT	CSECT	R9	)
XTXEN	CSECT	R9	)
XTUNL	CSECT	R9	)
XLINK	CSECT	R9	)
XSTG	CSECT	RA	Private storage for Phase KA.
XEQU			)
ZTEX2A			)
ZTEX2B			)
XTXEQU			)
ZTEX2C			)Books invoked by a COPY statement.
XBIF			)
YDICT1			)

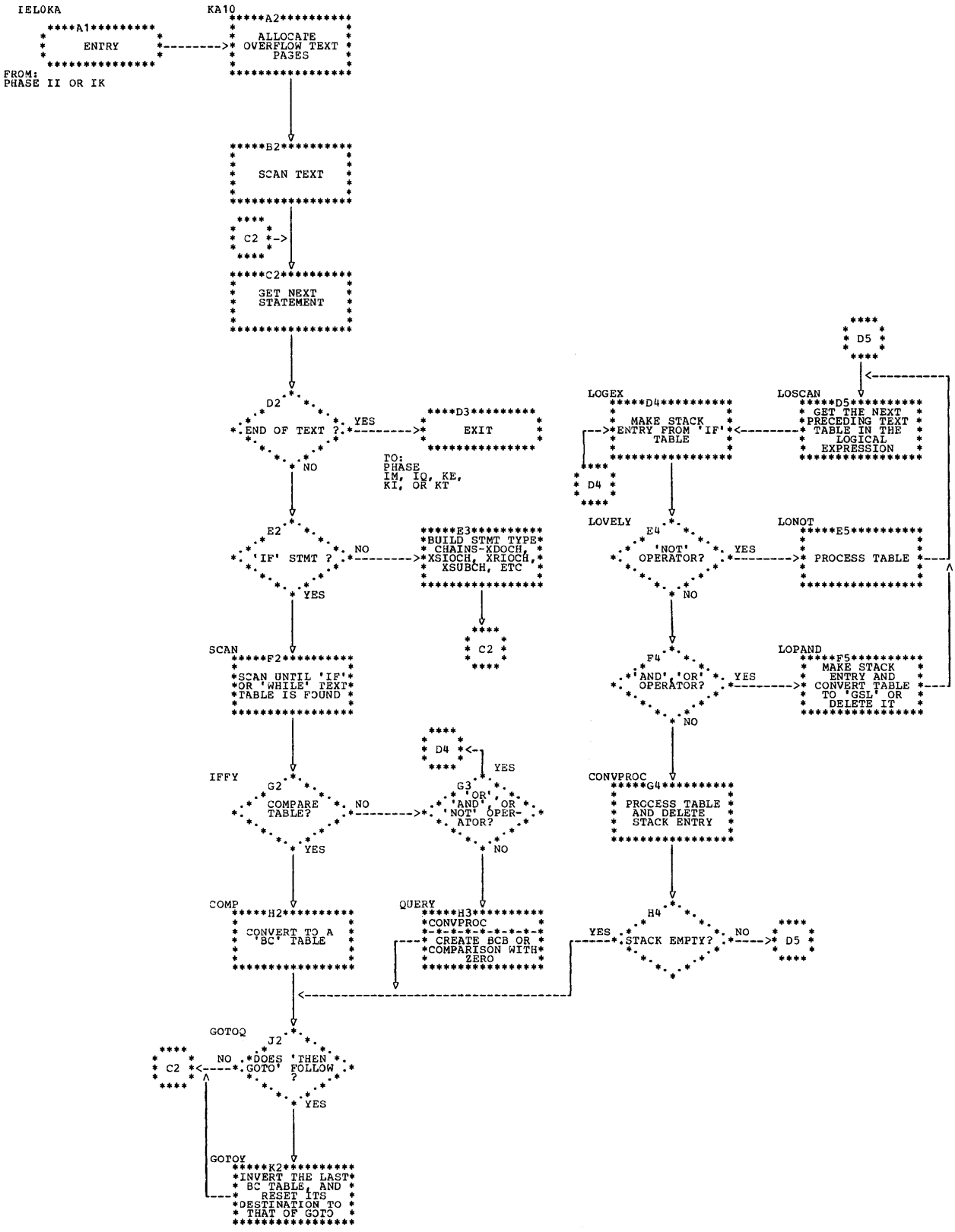


Chart 3.18. IF-statement Processing Phase (Phase KA)

Name	Type	Base registers	Function
IEL0IM	CSECT	R8	Entry point to Phase IM.
IM	R		Initialize registers, storage, scan of text, etc.
MCRAGE	S	R6,R7	Create an aggregate table entry for a structure, by copying an existing aggregate table, and flag it as COBOL, if required.
QTCOLV	S		Search a list of Q-temps. for a base reference, its associated pointer, and whether the Q-temp. is a scalar reference.
GTFIND	S		Search a list of Q-temps. for a given Q-temp.
QTENT	S		Allocate space in the list for a new Q-temp. entry.
PDNEXT	S		Return the next text table in the text stream, removing any dead Q-temps. from the list, and adding any new ones.
QTDELR	S		Delete a dead Q-temp. from the list of Q-temps.
PRCARG	S		Analyze the dictionary entries for the options on an entry point, and merge the options lists into a stack (ARGSTK).
PQQ000	S		'AND' the mapping options into ARGSTK for all parameters on an entry.
PZZ000	S		'AND' the mapping options into ARGSTK for a specified list of parameters on an entry.
REDFND	S	R8	Generate any additional text necessary to provide the interface with a COBOL file, for a record I/O statement.
LDUFND	S		Check that a COBOL option is not specified for a file referenced by a LOCATE, DELETE, or UNLOCK statement.
ARG000	R		Routine for handling calls and function references. It generates any additional text necessary to provide the interface on invocation of a FORTRAN or COBOL entry point.
MAGGAS	S	R6,R7	Create an aggregate assignment in text, for source and target aggregates having different mapping rules.
CXNSRT	S		Insert a new text table having the same operator and operands as the previous table.
MXNSRT	S		Insert a new text table.
CALLEP	S		Create a CALL text table for invocation of an entry point.
CALLGN	S		Create a CALL text table for invocation of a library routine.
ARGGEN	S		Create an ARG text table.
ALSTGN	S		Create an ALIST text table.
PLCBMP	S		Set MAPSAM on if a structure is mapped similarly in PL/I and COBOL.
RDSMAP	S		Create a RESDES text table.
MPPMAP	S		Create a MAP text table.
MAPCOD	S		Create an OFFSET and an ASSN table for aggregate extent assignment.
EXTMAP	S		Analyze an aggregate to determine whether it is adjustable, and if so invoke the appropriate routines for extent assignment.
PLFRMP	S		Set MAPSAM on if an array is mapped similarly in PL/I and FORTRAN.
COBFLT	S		Set COBFIL on if a file has the COBOL option.
MCRVDE	S		Create a variables dictionary entry by copying an existing entry and changing fields as required.
MCRFAR	S		Create an aggregate table entry for an array, and flag it as FORTRAN if required.
TXTSSS	S	R8	Scan the text stream for statements containing record I/O, calls, or function references with COBOL or FORTRAN options, and call the appropriate processing routines each time one is found.

Continued on next page

PRCFND	R		Examine all external entry points with FORTRAN or COBOL options, and invoke PRCFFF for each such entry point.
PRCFFF	S		Create a special procedure to contain the interface text for invocation of PL/I by a FORTRAN or COBOL procedure.
ARGCHK	S	R6,R7	Check the validity of arguments passed to a COBOL or FORTRAN entry point.
BRGCHK	S		Check the validity of arguments passed from a COBOL or FORTRAN procedure.
RETCHK	S		Check return values of function calls between FORTRAN and PL/I for validity.
MYTBFL	S		Ensures that aggregate temporary operand created in the encompassing procedure will be mapped during compilation.

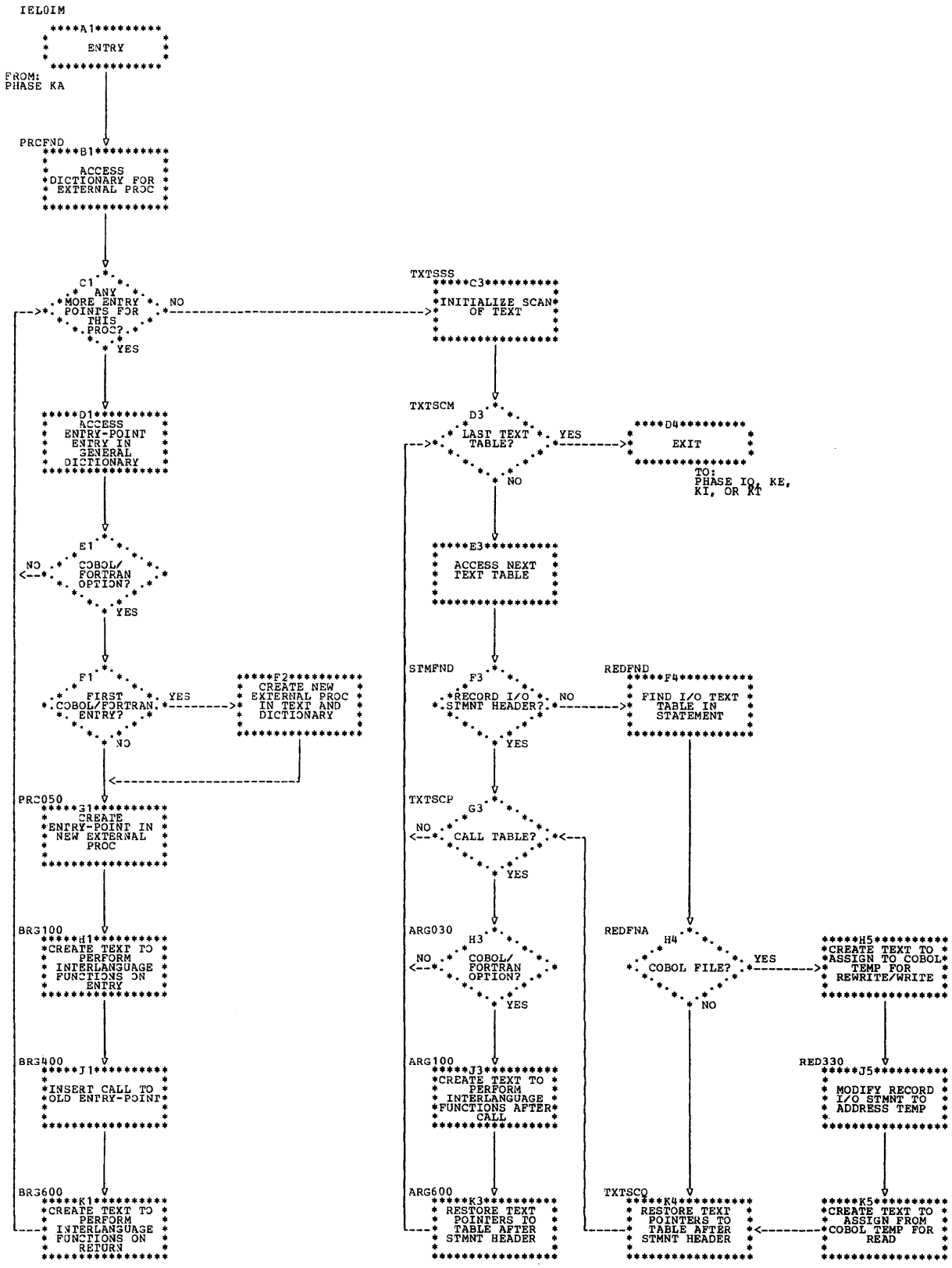


Chart 3.19. Interlanguage Communication Phase (Phase IM)

Name	Type	Base registers	Function
IEL0IQ	CSECT	R6,R7, R8	Entry point to Phase IQ.
SC0	R		Main text scan.
SC26	R		Process FREE text tables.
SC2	R		Process MAP text tables.
CALLEN	R		Generate code to calculate the length of a temporary array of adjustable strings.
GENASN	R		Generate assignment statement for the length of a string.
FINDTP	R		Find a temporary result in the result stack.
SETSZ	R		Determine the size and alignment of temporary aggregates.
STRFL	R		Calculate the length of the result of the STRING bif.
GENSDD	R		Called if a MAP text table refers to a structure that cannot be completely mapped at compile time. Uses the XCADD macro to construct a structure descriptor-descriptor for the variable and to insert it in an entry in the general dictionary.
IQDESC1	R		Set descriptor offsets in the aggregate table.
EXTEXP	R		Call the GENVM routine to create assignments from the old descriptor of a controlled variable to the temporary descriptor which is being used to map the variable.
GENLIB1	R		Generate library calling sequence tables.
GENMV1	R		Generate MOVE text tables for adjustable bounds and the virtual origin of adjustable arrays.
GENOFFS	R		Generate OFFSET text tables.
SC39	R		Process statement header tables.
SC25	R		Process RESDES text tables.
SC10	R		Process PEND text tables.
SC54	R		Process ACCUM text tables.
SC14	R		Process ALLOCATE text tables.
SC60	R		Process CONCAT text tables.
SC91	R		Process ASSN text tables.
SC3	R		End-of-program routine.
IQNSRTZ	R		Insert new text tables into main text stream.
LENBIF	S		Process LENGTH bif.
GENLIB	R		Routine to enter the GENLIB1 routine.
GENMV	R		Routine to enter the GENMV1 routine.
IEL0IQ2	CSECT	R6,R7	Entry point to second module of Phase IQ.
IQMAP	R		Complete the aggregate table entries in the general dictionary, as far as is possible at compile time, for those aggregates that have not been completely mapped.
XINIT	CSECT		)
XRFAB	CSECT		)
XTXPG	CSECT		)
XTUNL	CSECT		)
XRFSEQ	CSECT		) XROUT
XNSRT	CSECT		)
XNXROUT1	CSECT		)
XNXROUT2	CSECT		)
XDIREC	CSECT		)
XLINK	CSECT		)
XSTG	CSECT	RA	Private storage for Phase IQ.

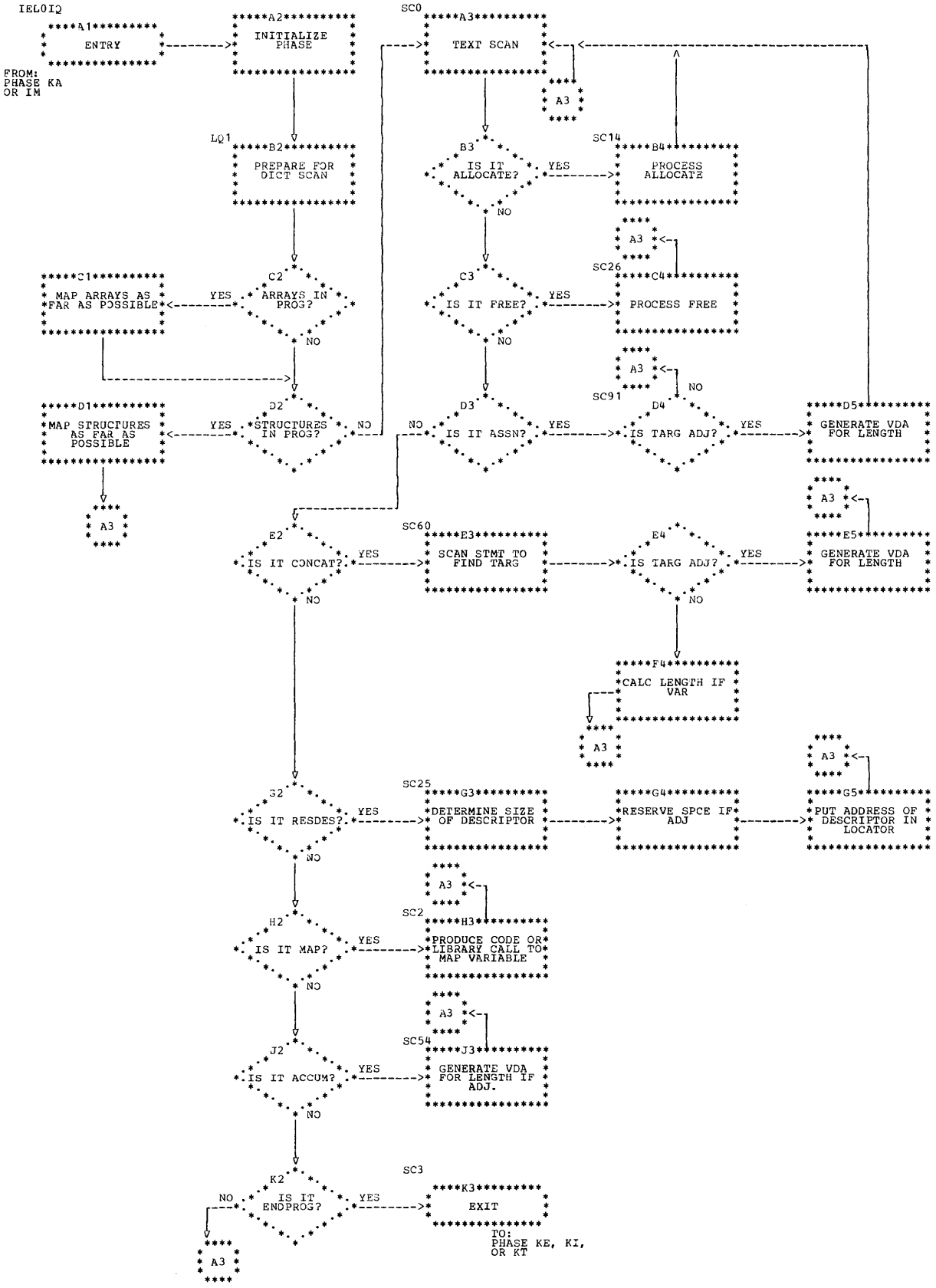


Chart 3.20. Aggregate and Structure Mapping Phase (Phase IA

Name	Type	Base registers	Function
IEL0KE	CSECT	R9,R6, R8	Root module of Phase KE.
NDXS	DSECT	R7	Contains the corresponding offset and temporary number for the index commoning in the subscript. For the iSUB defining routine it contains the index.
MULTS	DSECT	R2	Contains the calculated multipliers.
TAB	DSECT	R5	Contains information about the current array being processed.
DAZ1	E		)
DAZ2	E		) Entry points to module KE1.
START1	R		Point to the first statement header in the XSUBCH chain and subsequently at the preceding statement headers.
MVC1	R		Handle SOASSNs by generating the various sequences of code for array and structure assignments where they can be done by sequences of MVC instructions.
SUB1	S		Output the source offset table (OFFS 00).
SUB3	S		Output the target offset table (OFFS 01).
LENGSUB	S		Calculate the length, in bytes, of a structure for assignments done by moves.
LABA	R		Handle subscript assignments by processing the subscripts, calculating the multipliers and offsets, etc.
LABB	R		Calculate the required multipliers.
TCH20	R		Check the current multiplier and if it is not known at compile time, output the appropriate OFFS and SUBS(1) tables to address the multiplier in the object-time descriptor.
LOAC	R		Handle the constant subscripts.
CDE	R		Handle variable and temporary subscripts.
CJH	R		Output the current RS value.
EAC1	R		Output the current offset value (if any) by outputting on OFFS text table.
EAG	R		Process INDEX text tables.
SUBOP2	S		Place the appropriate code byte in the MULT text table.
OFFSUB	S		Place the appropriate code bytes in operand 1 of an OFFS text table.
SUBRGSUB	S		Check for occurrence of SUBSCRIPTRANGE condition.
PFBSUB	CSECT	R7	CSECT for a subroutine used to calculate, at compile time, subscripts of the forms A(I+1), A(I*1), A(I*2+1), A(I-1), and A(I*2-1).
DEF0	R		Process iSUB defining.
DEFSUB	CSECT	R9	CSECT for routine DEF0.
DOIT0	R		Process SEASSNs.
DOITSUB	CSECT	R9	CSECT for routine DOIT0.
DOITSUB1	S		Calculate the bounds of do-loops.
NEXT1	CSECT	R2	
NEXT3	CSECT	R2	
NSRT1	CSECT	R5	
NSRT3	CSECT	R5	
PTSATSUB	CSECT	R5	
XDIREC	CSECT		)
XTXPG	CSECT		)
XTUNL	CSECT		) XROUT
XLINK	CSECT		)
XSTG	CSECT		Private storage for module KE1.

Continued on next page

KE2	CSECT	R9,R6, R8	Non-root module of Phase KE.
AIDN	DSECT	R2	
INIT106	R		Handle array INITIAL assignments.
INIT0	S		Check for IASSN or AID text tables.
INIT81	R		Process array INITIAL assignments for interleaved arrays.
INIT100	R		Handle array INITIAL assignments for contiguous arrays.
MOVESUB	S		Generate the required number of MOVE text tables for the array assignment.
XRFAB	CSECT		)
XTUNL	CSECT		) XROUT
XMESGR	CSECT		)
XSTG	DSECT	RA	Private storage for module KE2.

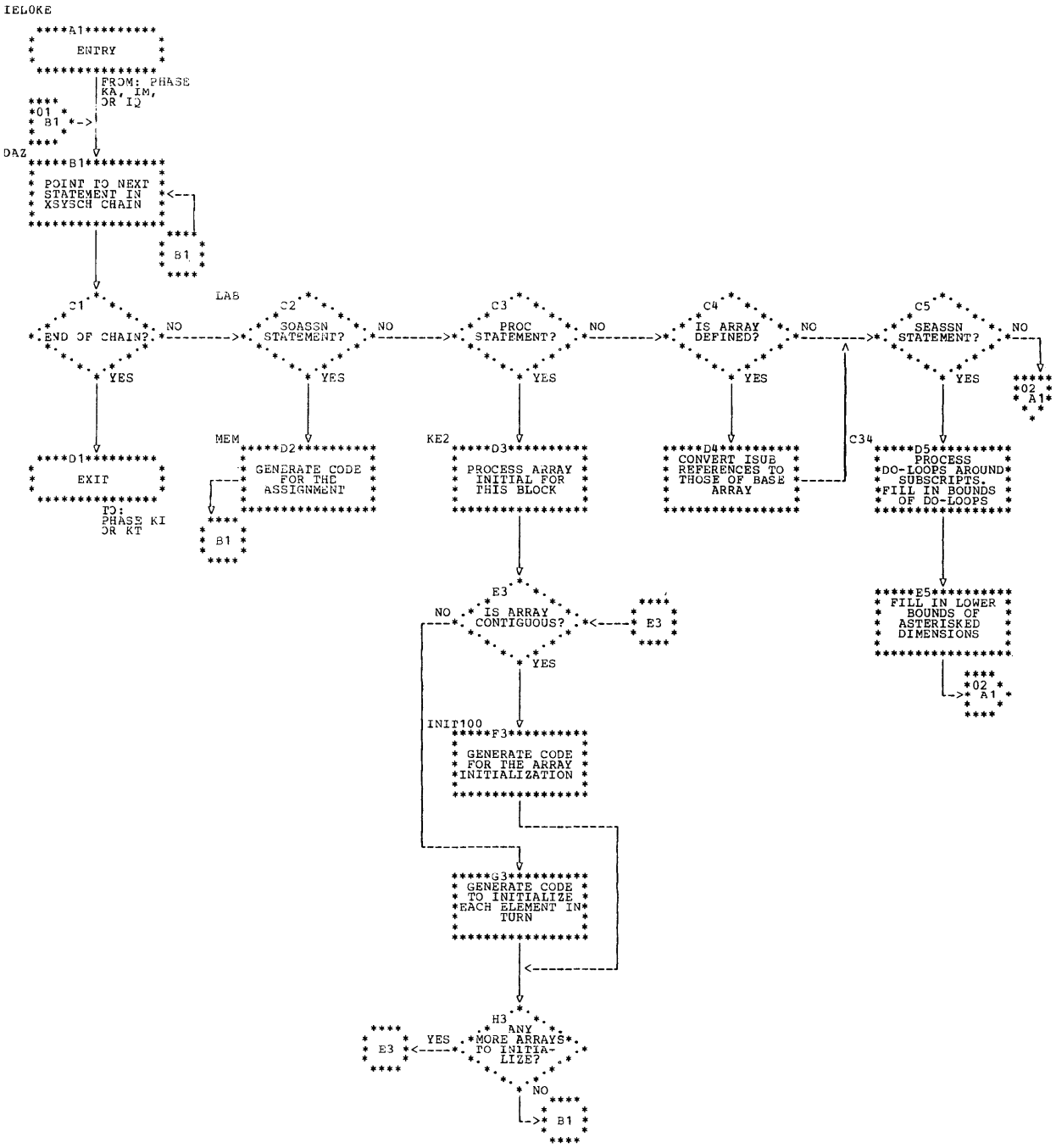


Chart 3.21. (Part 1 of 2). Subscript Processing Phase (Phase KE)

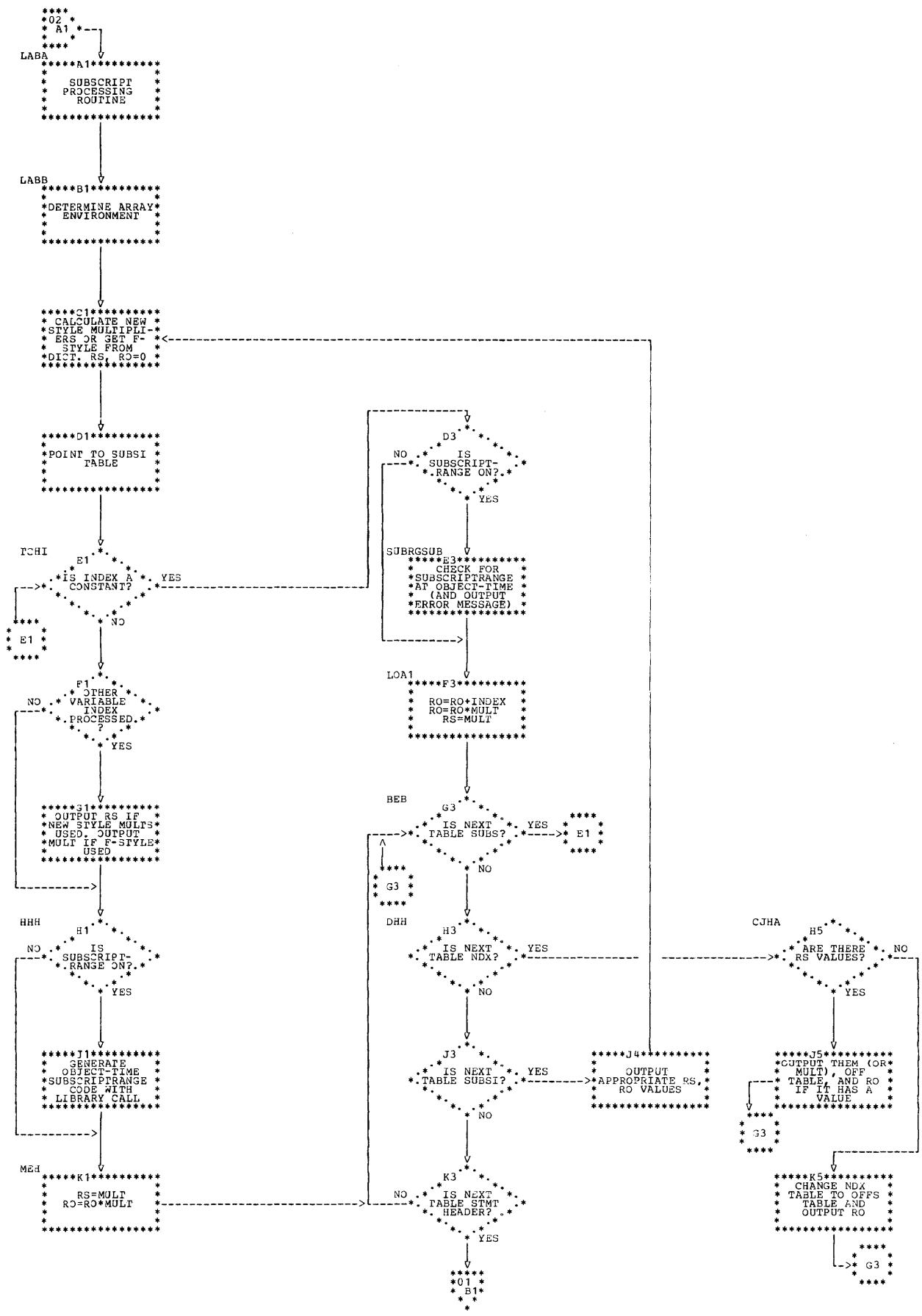


Chart 3.21. (Part 2 of 2). Subscript Processing Phase (Phase KE)

Name	Type	Base registers	Function
IEL0KI	CSECT	R6,R5	Entry point to Phase KI.
KI1	R		Initialize registers, storage, and scan of text.
CHAINDO	R		Scan along the statement chain to the next DO statement,
PROCESS	E		and initialize its processing.
SCAN1	R		Branch to the relevant processing routine, and process immediately if the current table is a CONV for a 'TO' or 'BY' variable.
SCANDO1	R		Process a DO1 table and determine whether BXLE is possible.
SCAN2	R		Scan to the associated DO2 table, and process it. Generate a SCI table if BXLE is not possible. Determine whether multiple specifications are present. Check that the loop specifications converge.
MULTIP	R		Generate loop head tables for multiple specifications, i.e., generate code to initialize the label variable temporary and to test the first specification initial value against the final value.
DOIT1	R		Save the base register and call DOIT10.
WHYLDO	R		Process a DOWHYL table without a control variable.
SINGLE	R		Single specification processing routine.
INCREM	R		Generate increment and test code at the end of a loop.
INBYVA	R		Generate alternative end-of-loop test code for a variable BY specification.
INCON	R		Generate increment code for an all-constant loop specification.
MULTI	S		Generate label variable assignments for multiple specifications after the first.
BYVARI	S		Generate label variable assignments for a variable BY specification.
DOIT10	S		Process array assignment ITDO tables by searching subscripts for multipliers with a common HCF.
DOESAV	R		Stack the text reference of a DO3 table, for use when loops nested.
KNEXTL4	S		Scan along the forward chain to the next table, using R4 as pointer. Unlock the old table, and lock the new table into main storage.
KNEXT4	S		Scan along the forward chain to the next table, using R4 as pointer.
KNEXTL1	S		Scan along the forward chain to the next table, using R1 as pointer.
ENDTAB	S		Insert GSN and ENDIT tables within iterative stream.
ENDTAB1	E		Input/output statements to provide the necessary format list matching information for Phase KQ.
GNOMORE	R		Free the current page and exit to Phase KK.
XINIT	CSECT	R9	)
XNXROUT1	CSECT	R9	)
XNXROUT2	CSECT	R9	)
XTXPG	CSECT	R9	) XROUT
XTCH	CSECT	R9	)
XNSRT	CSECT	R9	)
XTUNL	CSECT	R9	)
XLINK	CSECT	R9	)
XSTG	CSECT	RA	Private storage for Phase KI.

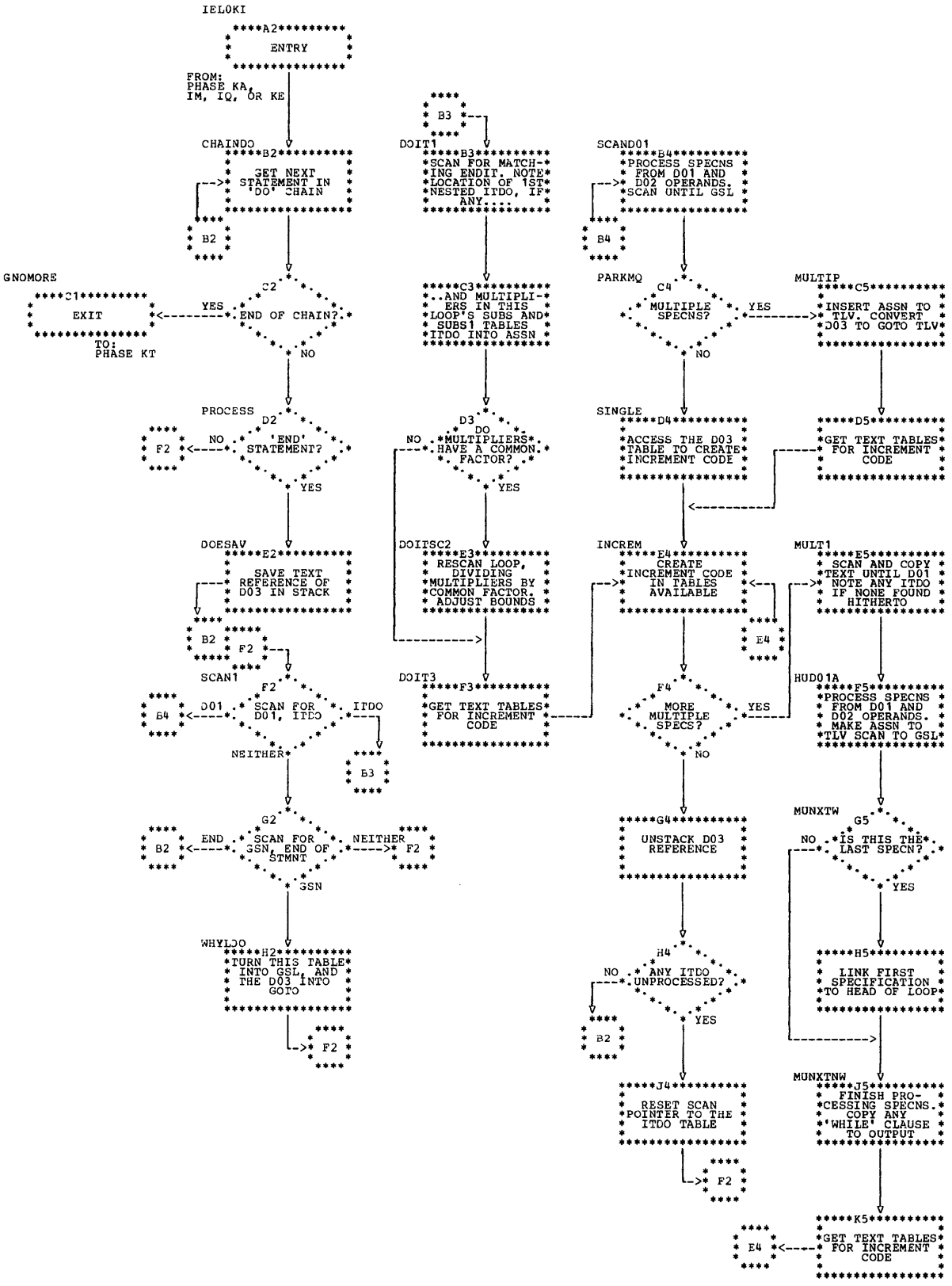


Chart 3.22. DO-statement Processing Phase (Phase KI)

Name	Type	Base registers	Function
IEL0KT	CSECT	R5,R8, R9	Root module of Phase KT. Processes PROCEDURE, ENTRY, BEGIN, RETURN, STOP, EXIT, DELAY, DISPLAY, SIGNAL, and WAIT statements.
START	R		Point to the first statement header in the XSYSCH chain, and subsequently at the preceding statement headers.
PROCA	R		PROCEDURE statement processor.
GSLSUB	S		Generate GSL text tables.
MOVESUB	S		Generate the appropriate number of MOVE text tables for moving the parameter(s) into the DSA.
RTRNSUB	S		Output the appropriate text table for the RETURN slot.
PENDSUB	S		Generate PEND text tables.
BEGINA	R		BEGIN ON-BEGIN statement processor.
RTRN1	R		RETURN statement processor.
STOPA	R		STOP and EXIT statement processor.
DELAY1	R		DELAY statement processor.
DISP	R		DISPLAY statement processor.
CONV04	S		Generate the appropriate text table to convert a given element to character.
DECFAC	R		Calculate the ceiling (N/3.32) for a given N. It is used to calculate decimal precision and scale in binary-to-decimal conversions.
SIGNALA	R		SIGNAL statement processor.
WAIT1	R		WAIT statement processor.
DELSUB	S		Delete illegal statements.
XSTG	CSECT		Private storage for module KT1.
XDIREC	CSECT		)
XTXPG	CSECT		)
XTUNL	CSECT		) XROUT
XLINK	CSECT		)
KT2	CSECT	R5	Non-root module of Phase KT. Only loaded if the CHECK prefix option is used in the program. Scans the entire text stream and carries out the required processing.
CA	R		Delete the CHECK/NOCHECK statement headers and move the PROC/BEGIN statement header and PROC/BEGIN statement in front of any CHECK/NOCHECK text tables.
CALL1	R		CALL statement processor.
ITDO1	R		Handle the CHECK condition for do-loop control variables.
DISP1	R		Handle CHECK condition for the REPLY option of a DISPLAY statement.
READ1	R		Handle CHECK condition for the identifiers appearing in the KEYTO and INTO identifier of the READ statement.
DATA1	R		Handle CHECK condition for the identifiers in the data list of an edit-directed or list-directed GET statement.
SIG1	R		Handle SIGNAL CHECK statements.
PUT1	R		Handle CHECK condition for the STRING option of the PUT statement.
VAR1	R		Scan the text looking for the actual variable being set in some form of assignment. Also look for arguments on function references.
CK1SUB	S		Scan the current stack of enabled CHECK parameters to look for CHECK.
CHKSUB	S		Generate the appropriate ALIST, ARG, and CALL text tables to signal CHECK.
CKSTRUC	S		Expand CHECK list containing structures to their base elements.

Continued on next page

ENDSUB	S		Point R1 at the last table in the statement.
ARRAY1	S		Stack the array reference together with its associated Q-temp. number.
ARRAY4	S		Find the array reference associated with its correspondingQ-temp. number.
XMESGR	CSECT		)
XRFAB	CSECT		)
XTUNL	CSECT		) XROUT
XTXPG	CSECT		)
XSTG	DSECT		Private storage for module KT2.

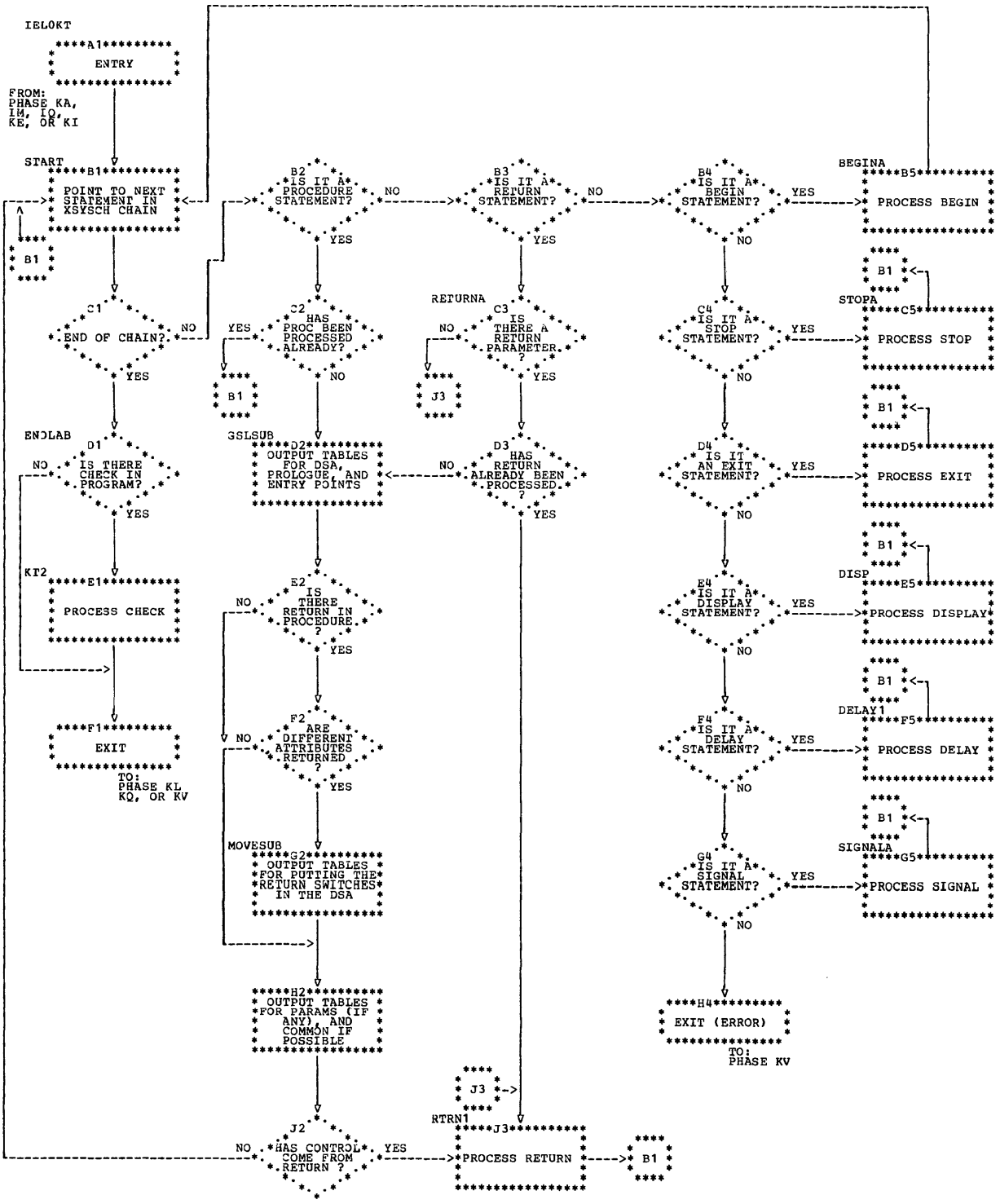


Chart 3.23. System Interface Processing Phase (Phase KT)

Name	Type	Base registers	Function
IEL0KL	CSECT	R6,R7, R8,R9	Entry point to Phase KL.
BEGKL	R		Initialize registers, storage, and scan of file entry chain.
DCLSET	R		Access the next entry in the file constant chain in the general dictionary
VMASK	R		Evaluate the 'valid statement' mask.
BUFSET	R		Check declared attributes, ENV options, and medium for DTF.
ENDCAL	R		Reserve space for a DTF entry, and branch to the routine for producing the required DTF dictionary entry.
MKENT	S		Obtain space for a DTF dictionary entry, and set up the first eleven bytes.
DTFCON	S		Store the first seven characters of the file name in the DTF entry, and set up bytes 6-7 to indicate which logical name (SYS---) the file has been given in the medium specification.
MKMOD	S		Make a names dictionary entry for the LIOCS module name for the DTF
ISCON	S		Subroutine used by DTF15 to set flag bytes.
MKDA	R		Make a DTFDA entry for a REGIONAL file.
MKDI	R		Make a DTFDI entry for SYSIPT, SYSLIST, or SYSPCH.
MKPR	R		Make a DTFPR entry for the printer.
MKCD	R		Make a DTF entry for the card reader or punch.
MKMT	R		Make a DTFMT entry for magnetic tape.
MKSDW	R		Make a DTFSD entry for a CONSECUTIVE UNBUFF disk file.
MKSD	R		Make a DTFSD entry for a CONSECUTIVE BUFF disk file.
MKIS	R		Make a DTFIS entry for an INDEXED file.
ENDDTF	R		Make an ENVB entry and insert in chain, if ENV declared.
ENDENV	R		Generate the FCB entry for this file.
NAMACC	R		Check for conflict between ENV options and declared attributes.
STHSCN	R		Scan the text chain for the next statement.
OPENPR	R		Process an OPEN statement.
CLSPR	R		Process a CLOSE statement.
GETDCL	S		Access the general dictionary entry for the file, if it exists.
HLFCNV	S		Convert an element to the required arithmetic form.
ALSTGN	S		Generate an ALIST text table.
ARGGEN	S		Generate an ARG text table.
CALLGN	S		Generate a CALL text table.
REFSAV	S		Convert an absolute address to a text reference.
CHKATS	S		Check declared attributes for validity and set up ATTWRD.
CHKENV	S		Process ENV attribute options.
RDNEXT	S		Scan to the next input text table.
RAHDST	S		Access the next input text table, locking in the current table.
RAHFIN	S		Reset the scan pointer on completion of a 'Read Ahead' operation.
STKTBL	S		Stack current input table for use by the WRTTBL subroutine.
WRTTBL	S		Access the next table to be output.
QTFIND	S		Locate a 6-byte qualified variable, given its Q-temp. reference.
CONVCH	S		Generate a text table to convert a given element to character.

Continued on next page

DECFAC	S		Calculate precision and scale in a BIN-to-DEC conversion.
ELTCHK	S		Test a 6-byte reference for convertability to arithmetic or string.
VMASK	R		Evaluate the 'valid statement' mask for a file.
ASMBUF	S		Check for a BUFFERS declaration.
	T		Tables, constants, and DTF macros.
XROUT	CSECTS	R9,RF	
XSTG	CSECT	RA	Private storage for Phase KL.

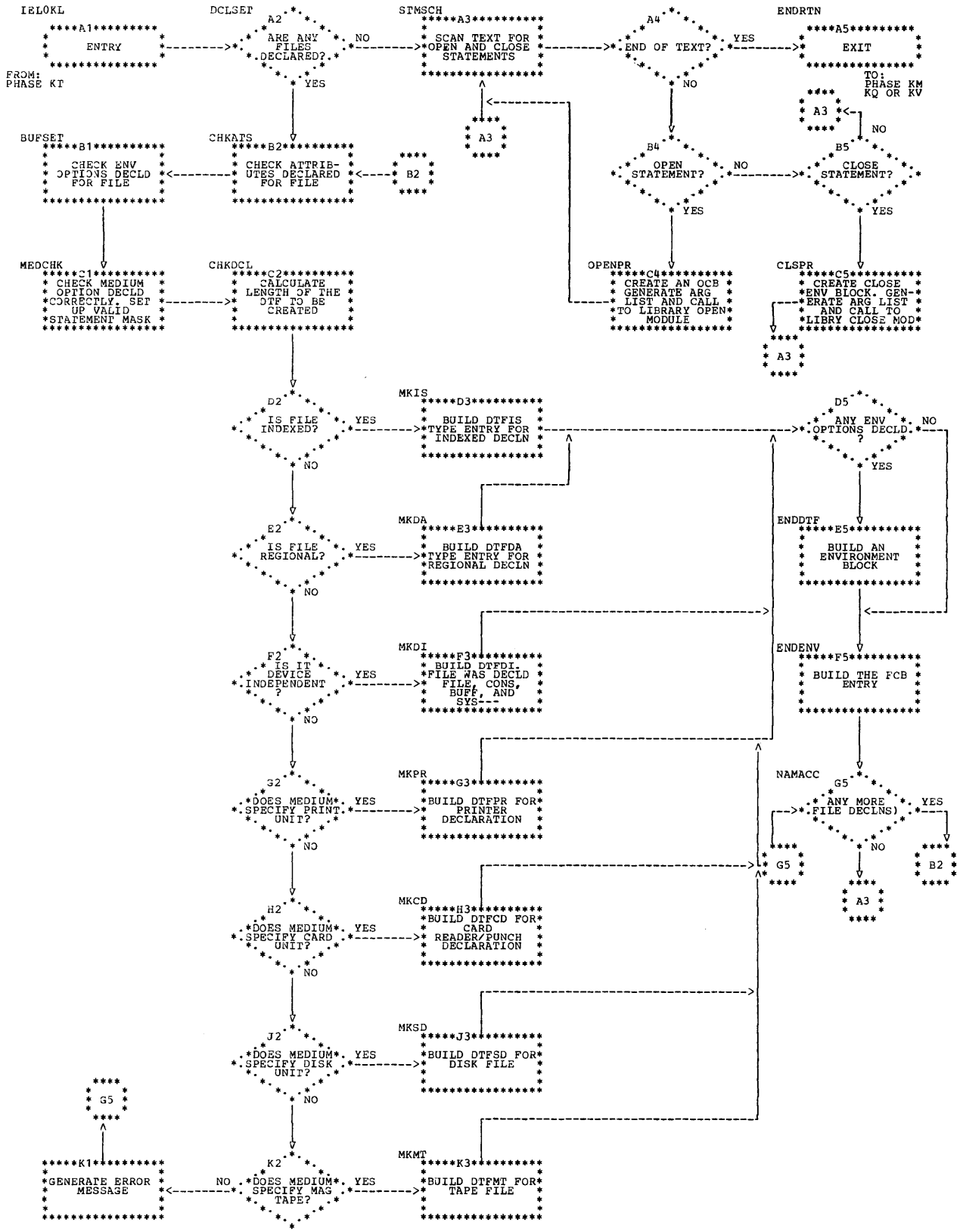


Chart 3.24. OPEN/CLOSE and File Declarations Phase (Phase KL)

Name	Type	Base registers	Function
IEL0KM	CSECT	R6-R9	Entry point for Phase KM. COPY-book definitions.
IBMDZFCB	DSECT		Execution-time (library) FCB.
IBMDZNVB	DSECT	R2	Execution-time (library) ENVB.
FILSC=0	R		Scans file-entry chain in general dictionary. Tests whether optimization of file open and close is feasible.
FILFIN	R		Checks size of file CSECT: if >=32K, no optimization; else modify FCB and DTF entries.
FILF10	R		Create buffer entry in DTF header.
SETUP	R		Start of a series of routines similar to subroutine in library modules IBMDOPA and IBMDOPB. Determine values for insertion in FCB and DTF.
LONGMV	S		Copies FCB and DTF entries from dictionary into XSTG, and copies modified entries back.
RFCBRT	S		Sets up an entry in the FCB-overflow entry to indicate an FCB entry that needs relocation.
RDTFRT	S		Indicates a field in the DTF that needs relocation.
STMSCN	R		Scan the text chain for record I/O statements.
RECIO	R		Process record I/O statements. Check the statements for validity against the FCB (if accessible) and generate a call to the library input/output interface module.
IGNRRT	R		Process the IGNORE option.
INFRRT	R		Examine the INTO and FROM options in a record I/O statement. Main task is to create a record descriptor (RD), and produce text tables to set this descriptor up at execution time.
IFQT	R		Process the INTO/FROM option in those cases where the record descriptor does not have a skeleton in static storage.
LTHFND	R		Attempt to find the length of a record variable. If this length is unknown at compile-time, then a study is made of those factors which are used in calculating the length at object-time. Such factors as are known at compile-time are saved, and flagged as known in LTHFLG.
NXTENT	R		Access the next variable dictionary entry (and hence the next item in a structure).
TRFTS	R		Determine whether the YTROFF field in the aggregate table dictionary entry (if any) is known for the given variable. Also set CBLKCD and DSCROFF to indicate whether descriptor fields are accessed via a locator descriptor or via a descriptor.
LIBLTH	R		Generate a call to the library module which calculates the length of a given structure.
SETRT	R		Process the SET option.
KEYRT	R		Examine the KEY/KEYTO/KEYFROM options in a record I/O statement. Create a key descriptor (KD) for the key, and leave a 6-byte reference to the descriptor in RCARG4, so that an ARG table for the key descriptor can be constructed later.
RLTHGN	R		Generate text tables to calculate the length of a record or key variable. If the variable is a structure, the length of the last base element is added.
MHCONS	R		These two routines calculate

Continued on next page

MHVAR	R		SUM(M*H) - (VO-AO) where M,H are multipliers and bounds of given array; AO = actual origin; VO = virtual origin. Thus, in effect, the routines calculate the offset of the last element of an array from its actual origin. Routine MHCONS calculates the part of this expression known at compile time; MHVAR generates text to calculate the rest of the expression.
PRECST	R		Determine whether a given constant needs halfword or fullword precision, and generate a 6-byte reference to the constant accordingly.
OFFSGN	R		Generate an OFFSET text table.
ASSTGN	R		Generate an ASSN text table.
TTGEN	R		Generate a text table.
EGEN	R		Calculate one of the factors used by routine RITHGN.
ASLDSC	R		Generate text to assign the contents of LTSTMP to the length field in a record or key descriptor.
MVADDR	R		Move the address of a variable into a record or key descriptor.
MVFLAG	R		Generate text to move the descriptor flag into a record or key descriptor.
EVNTRT	R		Process the EVENT option.
PGLSET	R		Determine whether the LINESIZE and/or PAGESIZE options on an OPEN statement conflict with attributes for the data set to be opened, and set the PGLNSW switch accordingly.
GETDCL	R		Access the FCB dictionary entry for a given data set.
QTFIND	R		Locate a 6-byte qualified variable, given a reference to its qualified temporary.
QTSQZE	R		Locate the NDPTAB entry which corresponds to a given QT, and 'squeeze' the entry out of the table.
TDSCIN	R		Set up a field TMPDSC with a 6-byte reference to a new temporary 8-byte record or key descriptor. Move the 6-byte descriptor reference into a field pointed at by R2 and ensure that the code byte in this field indicates a 'dead' temporary.
CONVCH	R		Generate a text table to convert a given element to character.
DECFAC	R		Calculate the ceiling (N/3.32) for a given N. Routine is used to calculate decimal precision and scale in binary-to-decimal conversions.
FLLCNV	R		Convert an element to FULLWORD BINARY(31,0), if necessary.
ELTCHK	R		Examine a 6-byte reference to an element, and decide whether it is a scalar and convertible to arithmetic or string. If not, signify an error.
ALSTGN	R		Generate an ALIST text table.
ARGGEN	R		Generate an ARG text table.
CALLGN	R		Generate CALL text tables.
GHOSTG	R		Generate a GHOST text table to indicate to the optimizer that a variable is being used or set.
REFSAV	R		Convert the address of a text table into a text reference.
GTXTBL	R		Convert a text reference to an absolute pointer.
SCRBST	R		Delete the current statement.
INMON	R		Check whether inline I/O code is possible, and if so, call routines to generate such code (routines RT0-RT22).
RDNEXT	R		Access the next input text table and update the input scan pointers CURTBL (which points to the new table) and CURPAG (which points to the start of the page containing CURTBL).
RAHDST	R		Access the next input text table, ensuring that the previous table is locked into main storage. Used at the start of a 'read-ahead' operation.
RAHFIN	R		Used when a 'read-ahead' operation is complete. It resets the input scanner to the value it had before the operation commenced.
STKTBL	R		Stack the current input table so that it may be used by the routine WRTTBL.

Continued on next page

WRTTBL	R		Access the next output text table.
RT0	R		Perform record checking for fixed format records.
RT1	R		Initialize an inline record input/output sequence and generate text to indicate to Phase QA the registers being used and set.
RT2	R		Generate code to load the address of the error routine generated by routine RT3.
RT3	R		See routine RT2, above.
RT4	R		Generate code to load the address of a library error routine and branch to that routine.
RT5	R		Generate ERRLAB EQU * The compiler label ERRLAB has been set up by a previous routine.
RT6	R		Generate the clear-up code of an inline record I/O statement, and indicate that registers 1 and 2 are now free.
RT7	R		Generate code to assign the buffer address to the pointer which is tc be set.
RT8	R		Generate text to move the record variable from/to the buffer.
RT9	R		Generate code to load the address of the ABNORMAL PUT RETURN label.
RT10	R		Generate PUTRN EQU * This label is the ABNORMAL PUT RETURN label.
RT11	R		Generate text to indicate to Phase QA that register 8 is now free for use.
RT12	R		Generate code to deblock FB-format records on input.
RT13	R		Generate code to load the address of the ABNORMAL LOCATE RETURN label.
RT14	R		Perform record checking for areas and varying strings (with SCALARVARYING option) on (non-locate) output statements.
RT15	R		Generate code to check for a record variable being too long.
RT16	R		Generate text to move data (whose length is in LTSTMP) between buffer and record variable. The text generated produces a call to the general move routine.
RT17	R		Perform compile-time record checking for U-format records, when possible.
RT18	R		Generate text to indicate to Phase QA that register 10 is set.
RT19	R		Generate text to indicate to Phase QA that register 10 is free.
RT20	R		Generate text to perform record checking for input statements of U-format data sets.
RT21	R		Generate text to assign the record variable length to field FREL in the FCB.
RT22	R		Generate text to pick up the length of the last U-format record in the buffer.
RCBTAB	T		Contains flags indicating the type of each record I/O statement.
TMTAB	T		Each entry contains the immediate and offset bytes of a TEST UNDER MASK instruction which tests flags in the FCB for statement validity.
XMESGR	CSECT		)
XRFAB	CSECT		)
XRFSEQ	CSECT		) XROUT
XTUNL	CSECT		)
XSTG	CSECT		Private storage for Phase KM.

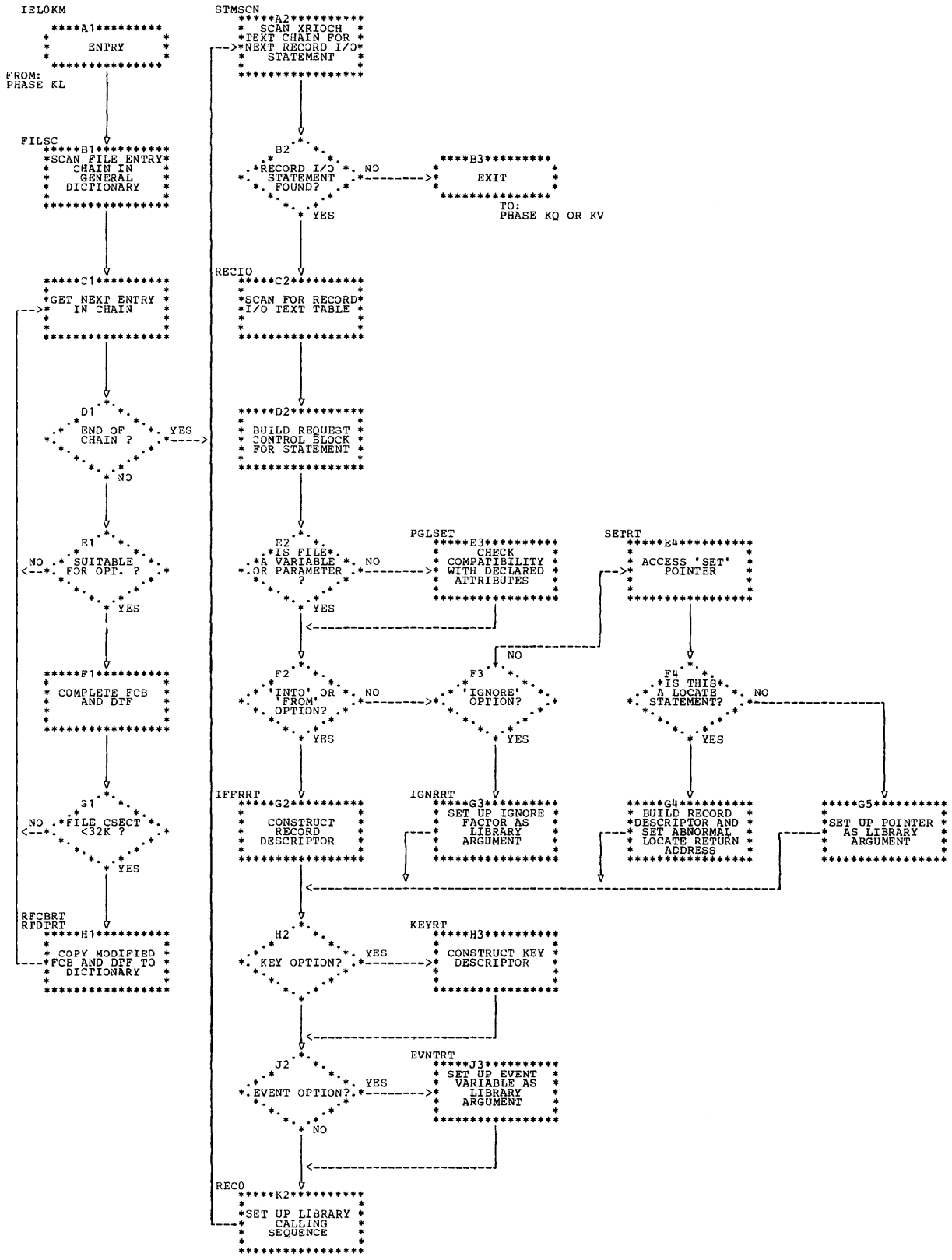


Chart 3.25. Record I/O Statement Processing Phase (Phase KM)

Name	Type	Base registers	Function
IEL0KQ	CSECT	R5,R6, R8,R9	Entry point to Phase KQ.
KQ0000	R		Initialize registers, storage, and scan of text.
KQSS00	R		Scan to the next statement in the stream input/output chain.
KQGP00	R		Scan a GET/PUT text table, and prepare the table for optimization if found.
KQST00	R		Check the syntax of a GET/PUT STRING statement.
KQSF00	R		Check the syntax of a GET/PUT FILE statement.
KQIC00	R		Convert a GET/PUT FILE text table from input to output.
KQSA00	R		Generate string arguments for an initial library call for a GET/PUT STRING statement.
KLSC00	R		Text table scan for a LIST-directed input/output statement.
KLSE00	R		Process a DATAE table in a LIST directed input/output statement.
KLAE00	R		Process an array data list element in a LIST directed input/output statement.
KESC00	R		Start the processing of an EDIT directed input/output statement.
KEDT00	R		Process an EDIT directed input/output statement in NO OPT mode.
KDSC00	R		Text table scan for a DATA directed input/output statement.
KDNL00	R		Process a DATA directed input/output statement without a data list.
KDIL00	R		Process a DATA directed GET statement with a data list.
KDOL00	R		Process a DATA directed PUT statement with a data list.
KQFM00	R		Process a FORMAT statement.
KQSE00	R		Complete the processing of a statement and continue scan.
KQSR00	R		Add compiler-generated subroutines for stream input/output to the start of the text stream.
QCSRGT	T		Subroutine generation table.
KQLC00	R		Set XLIBSTR to show library conversion routines required at execution time.
KQND00	R		Exit to the next phase.
SGNIT0	S		Get the next input text table.
SGNOT0	S		Get the next output text table space.
SREMB0	S		Store the address of a current text position, and lock it.
SFREE0	S		Free a text position previously locked in core by SREMB0.
SRECL0	S		Retrieve a previous text position, and save the current position.
SRETN0	S		Retrieve a previous text position.
SCURR0	S		Return to the current text position after processing by SRECL0.
SALST0	S		Create an ALIST text table.
SARGT0	S		Create an ARG text table.
SCALT0	S		Create a CALL text table.
SCDLE0	S		Check a data list element for validity.
SILCR0	S		Identify required library conversion routines.
SLMEP0	S		Set a library module entry point bit on in XLIBSTR.
SOFFT0	S		Create an OFFS text table referencing the SIOCB.
SLADT0	S		Create an LADDR text table.
SGENT0	S		Create a GEN text table.

Continued on next page

SAQTL0	S		Add an element to the Q-temp. list.
SSQTL0	S		Search the Q-temp. list to find the real operand referenced.
SCVFB0	S		Convert a text table operand to fixed binary.
SELSB0	S		Produce a text table to generate code to locate the SIOCB.
SELFC0	S		Produce a text table to generate code to locate the start of a format list.
SELDD0	S		Produce text tables to generate code to locate a data item and its DED, and store their addresses in the SIOCB.
SEBAL0	S		Produce a text table to generate code to branch and link between a data item and its format list.
SEFED0	S		Construct a FED for a format item, construct a text table operand referencing the FED, and identify library conversion routines required for the FED.
SEFDD0	S		Construct a dictionary entry for a FED, if required, and a text table operand (6-byte reference) referencing the FED.
SELFI0	S		Produce a text table to generate code to locate the current format item.
SECFF0	S		Produce text tables to generate code to call a library routine for processing a format item.
SECSR0	S		Produce text tables to generate code to call a compiler-generated subroutine.
SECLC0	S		Produce a text table to generate code to call a library routine for an EDIT-directed input/output conversion.
SEGOT0	S		Produce a text table to generate code to branch to a compiler-generated label.
SEGSL0	S		Produce a text table to generate a compiler label.
SEFHT0	S		Process a format item.
SEFIT0	S		Process a format iteration start (FIT) text table.
SEFIE0	S		Process a format iteration end (FITE) text table.
SEKDN0	S		Create a KONST text table in the output text stream.
XINIT	CSECT	R9	)
XRFAB	CSECT	R9	)
XDIREC	CSECT	R9	)
XNEXT	CSECT	R9	) XROUT
XTXPG	CSECT	R9	)
XICH	CSECT	R9	)
XNSRT	CSECT	R9	)
XLINK	CSECT	R9	)
XSIG	CSECT	RA	Private storage for Phase KQ.

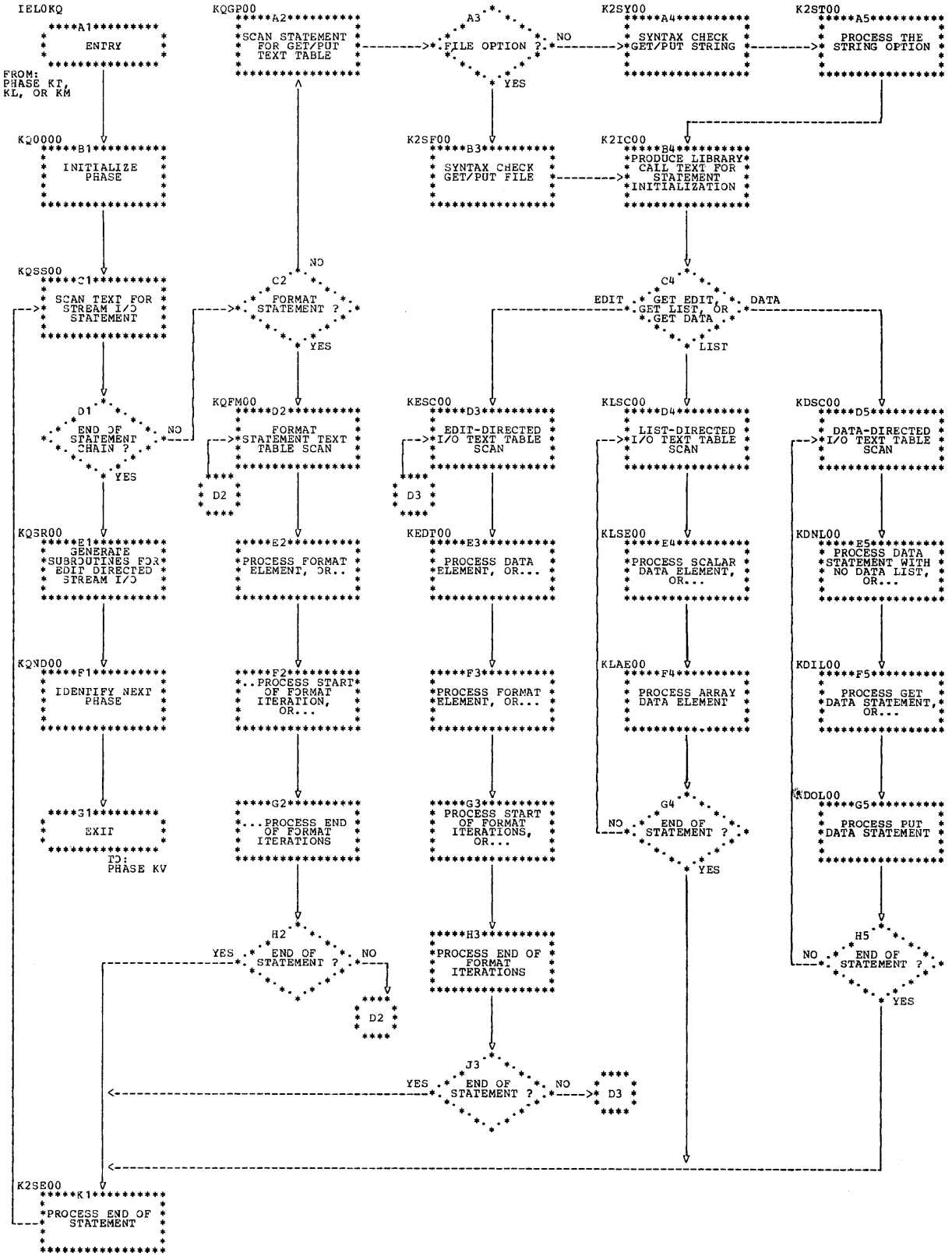


Chart 3.26. Stream I/O Statement Processing Phase (Phase KQ)

Special-case Processing (Phase KV)

Name	Type	Base registers	Function
IEL0KV	CSECT		Entry point to Phase KV.
KV1	R		Initialize registers, storage, etc.
SCANT	R		Scan the input text stream, processing and outputting special-case tables including the following types: DINC,NULL,BC,BCB,GOTO,PROCEDURE,BEGIN,END,CALL,FNCT,ALIST, statement header, end-of-program.
SCANGLAB	S		Process generated statement labels, eliminating those which are redundant.
LCCKON	S		Scan text to the next non-NULL text table, deleting any NULL tables found.
MPO	S		Process constant multipliers and exponents.
LABENT	S		Find or make a label directory entry.
DESTN	S		Process the label operands of a conditional branch or GOTO text table.
XINIT	CSECT	R9	)
XNXROUT1	CSECT	R9	)
XNXROUT2	CSECT	R9	)
XIXPG	CSECT	R9	) XROUT
XNSRT	CSECT	R9	)
XTUNL	CSECT	R9	)
XLINK	CSECT	R9	)
XSTG	CSECT	RA	Private storage for Phase KV.

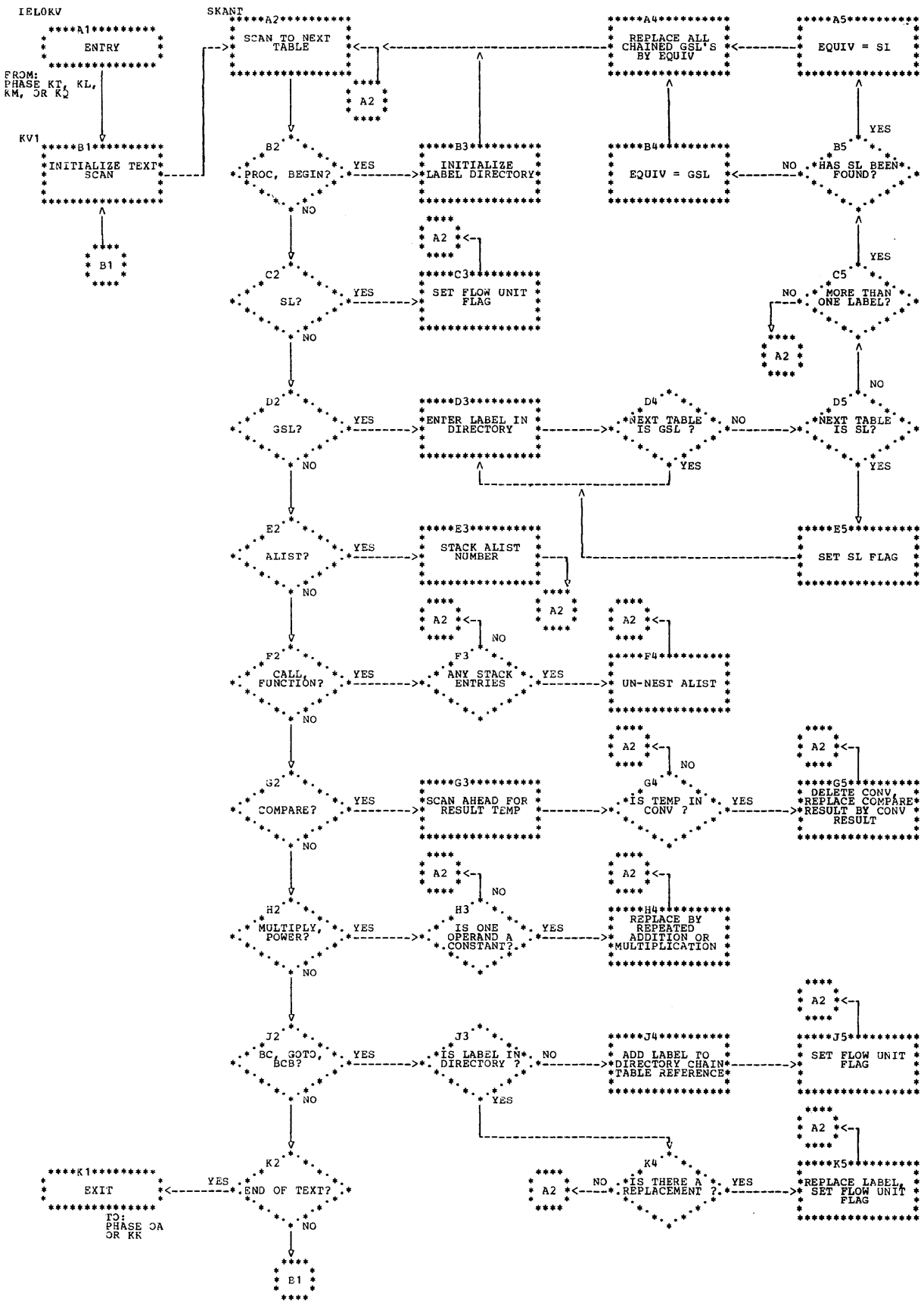


Chart 3.27. Special-case Processing Phase (Phase KV)

Extraction of Alias and Call Information (Phase OA)

Name	Type	Base registers	Function
IEL0OA	CSECT		Entry point to Phase OA.
OA	R		Initialize registers, storage, text scan, etc.
TSCAN1	R		Scan to the next text table, and branch to the required processing routine.
ENDTX1	R		End of text-scan routine.
DEF10	R		Scan to the next entry in the variables dictionary.
TRNS1	R		Process the secondary transfer table)
TRNS2	R		Process the primary transfer table)Bit vector
TRNS3	R		Process the CALL table)manipulation
PTRASN1	R		Process a pointer assignment.
LABENT	R		Process a label or entry assignment.
ARGR1	R		Argument/parameter matching routine.
CALLR1	R		Process a CALL statement.
ASSNR1	R		Preliminary processing of assignment statements.
SETUPVL	R		Set up value lists.
DEF20	R		Process a defined variable.
XINIT	CSECT	R9	)
XRFAB	CSECT	R9	)
XRFSEQ	CSECT	R9	)
XDIREC	CSECT	R9	) XROUT
XDSTAT	CSECT	R9	)
XTXPG	CSECT	R9	)
XNXROUT	CSECT	R9	)
XSTG	CSECT	RA	Private storage for Phase OA.

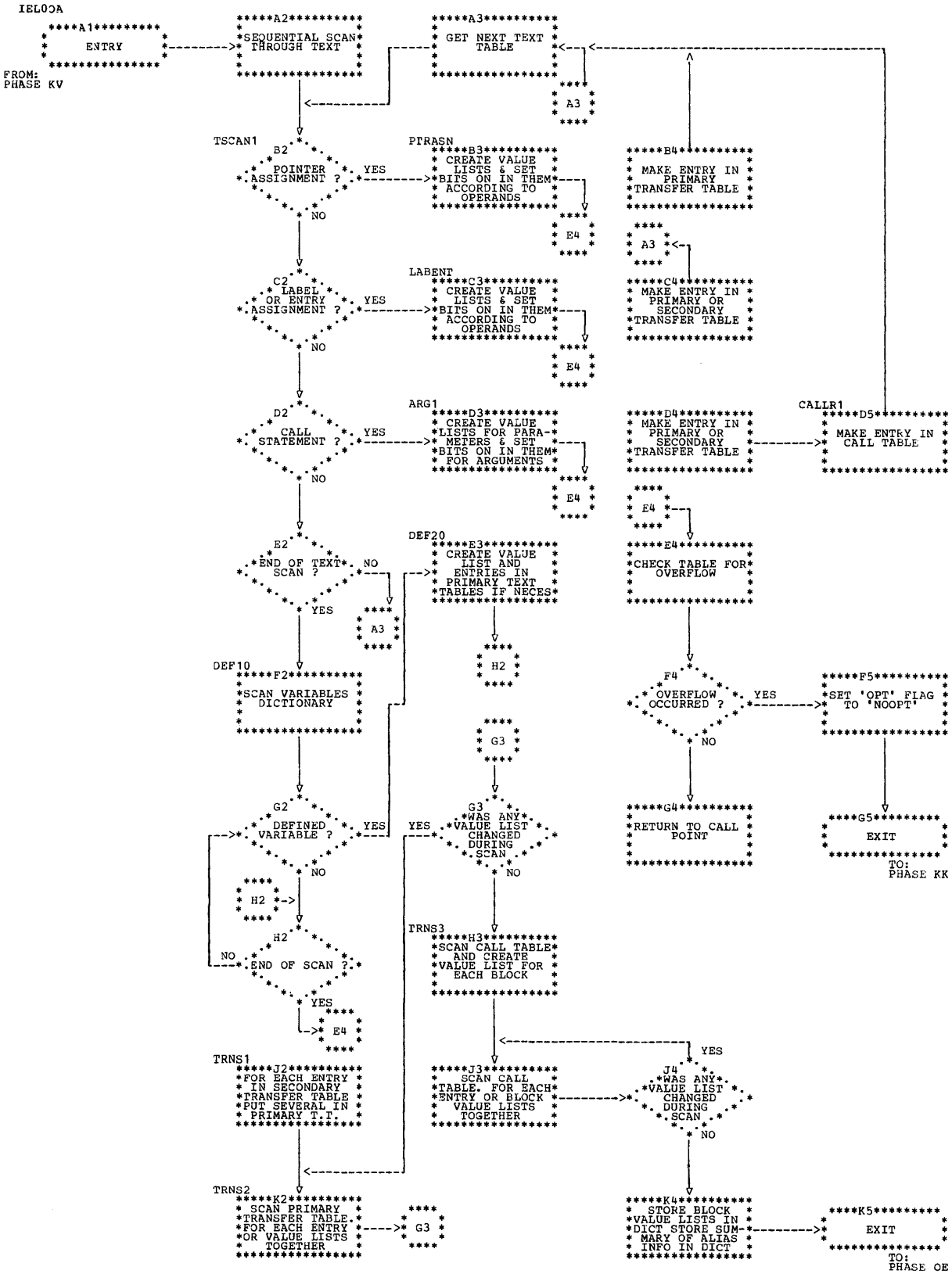


Chart 3.28. Extraction of Alias and Call Information (Phase OA)

Name	Type	Base registers	Function
IEL0OE	CSECT		Entry point to Phase OE.
VAR01	R		Scan variables dictionary for controlled variables.
OE	R		Initialize registers, storage, text scan, etc.
SCAN1	R		Scan to the next relevant text table, and branch to the required processing routine.
ARGR1	R		Process ARG text tables.
CALLR1	R		Process CALL text tables. Extract branching and flowpath information and store in flow-unit header.
STHD1	R		Process a statement header table.
FLOW1	R		Insert a flow-unit header into the text stream.
BLOCK1	R		Store variable usage information in a block optimization dictionary entry when text scan reaches end of block.
SLAB1	R		Extract statement label information.
BRCH2	R		Extract branching and flowpath information, and store it in the flow-unit header.
ENDTX1	R		End of text-scan routine.
CB1	R		Bit-vector manipulation routine, used to complete the block optimization entries in the general dictionary, and store information on variables used in on-units.
SETSUB	R		Extract information about variables set in current flow-unit.
BENSUB	R		Extract information about variables used in current flow-unit.
TLINK	R		Chain together two text tables.
BLKSUB	R		End-of-block routine.
NPSUB	R		Routine to handle the case of a flow-unit which extends over more than one page.
XINIT	CSECT	R9	)
XRFAB	CSECT	R9	)
XDIREC	CSECT	R9	)
XBREAK	CSECT	RF	) XROUT
XNXROUT	CSECT	R9	)
XTXPG	CSECT	R9	)
XRFSEQ	CSECT	R9	)
XSTG	CSECT	RA	Private storage for Phase OE.

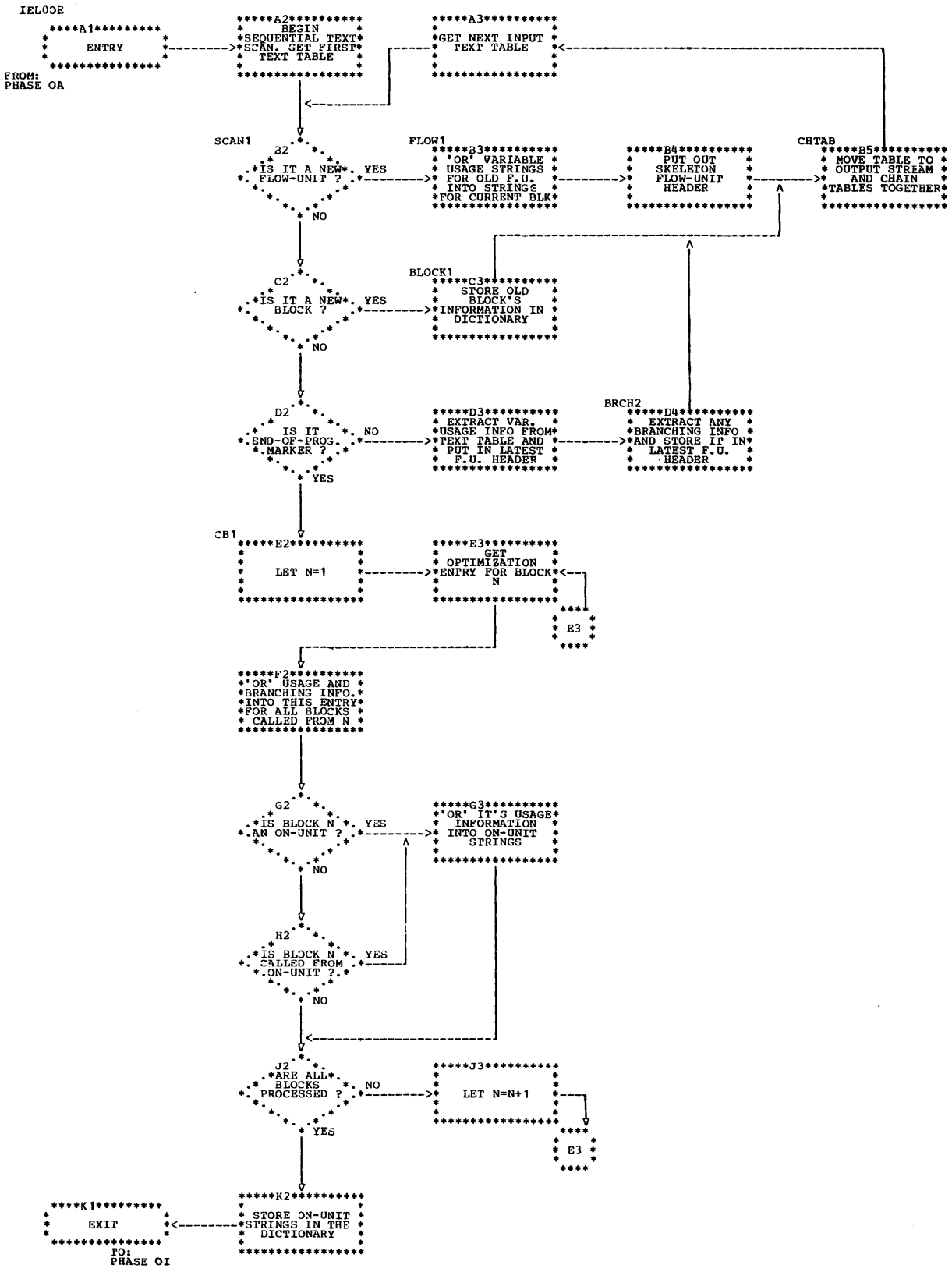


Chart 3.29. Extraction of Variable Usage Flowchart Information (Phase OE)

Flow Analysis (Phase OI)

Name	Type	Base registers	Function
IEL0OI	CSECT		Entry point to Phase OI.
OI	R		Initialize registers, storage, scan of text, etc.
INSFCT	R		Scan through the text stream inserting provisional forward connectors.
BLVSUB	R		Extract value list information for a label variable.
GENFCT	R		Generate a forward connector list.
FCTOBC	R		Generate a back connector list.
CALCLN	R		Order flow-units and assign level numbers and some back dominators.
CALCBD	R		Calculate back dominators for all flow-units.
FLODEN	R		Establish loop entry points and back targets.
FLOODA	R		Identify and order loops, and reorder flow-units.
BUSYEX	R		Generate busy-on-exit information for each flow-unit.
LOODAT	R		Scan the text stream for a second time, modifying flow-unit headers, and collecting loop data information in the general dictionary.
XINIT	CSECT	R9	)
XRFSEQ	CSECT	R9	) XROUT
XRFAB	CSECT	R9	)
XTXPG	CSECT	R9	)
XSTG	CSECT	RA	Private storage for Phase OI.

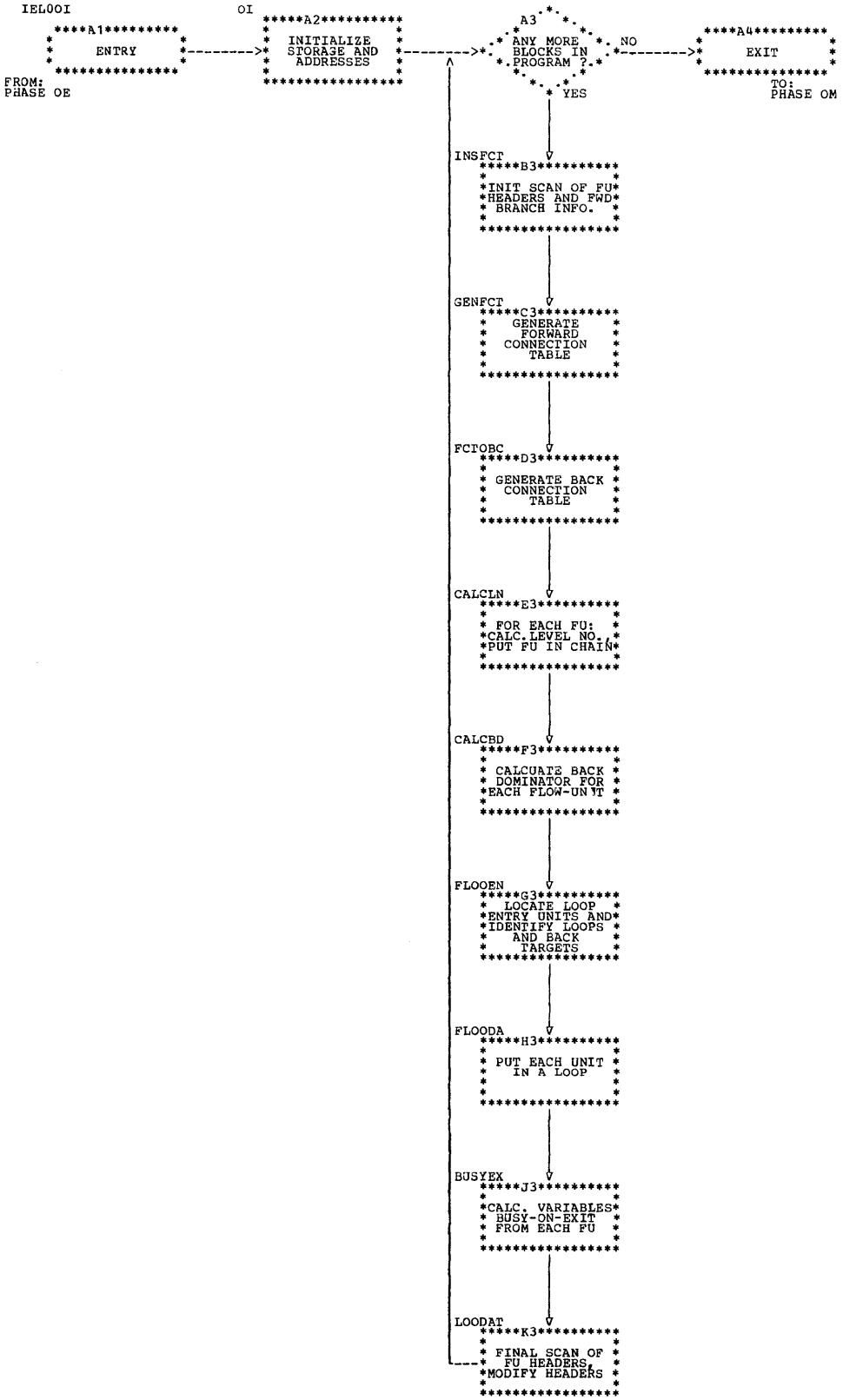


Chart 3.30. Flow Analysis Phase (Phase OI)

Text Optimization (Phase OM)

Name	Type	Base registers	Function
IEL0OM	CSECT		Entry point to Phase OM.
OM	R		Initialize registers, storage, text scan etc.
COMEXE	R		Scan through the text stream, eliminating common expressions and performing other optimization.
BAKMOV	R		Back movement of invariant-expressions routine.
STREDU	R		Strength reduction routine.
CLENBT	R		Tidies up back targets.
SBS000	R		Collect special casing necessary for dependent subscript lists.
GENTEM	S		Generate a new temporary operand.
HASHSB	S		Process hash operands in a text table.
SEMGT	S		Maintain a list of movable global temporary operands.
TEMGT	R		Test whether a global temporary operand is in the list created by SEMGT.
SINERT	R		Process special PLUS tables for strength reduction.
DEFALT	R		Calculate default scale and precision for the result of an expression.
DELSUB	S		Delete a table from the text stream.
REPSUB	S		Replace a temporary operand in text.
INSUB1	S		Insert a table into the text stream.
INSUB2	S		Process independent subscript lists.
FSM000	CSECT		
XINIT	CSECT	R9	)
XRFAB	CSECT	R9	)
XNXROUT1	CSECT	R9	)
XNXROUT2	CSECT	R9	) XROUT
XTXPG	CSECT	R9	)
XNSRT	CSECT	R9	)
XTUNL	CSECT	R9	)
XSTG	CSECT	RA	Private storage for Phase OM.

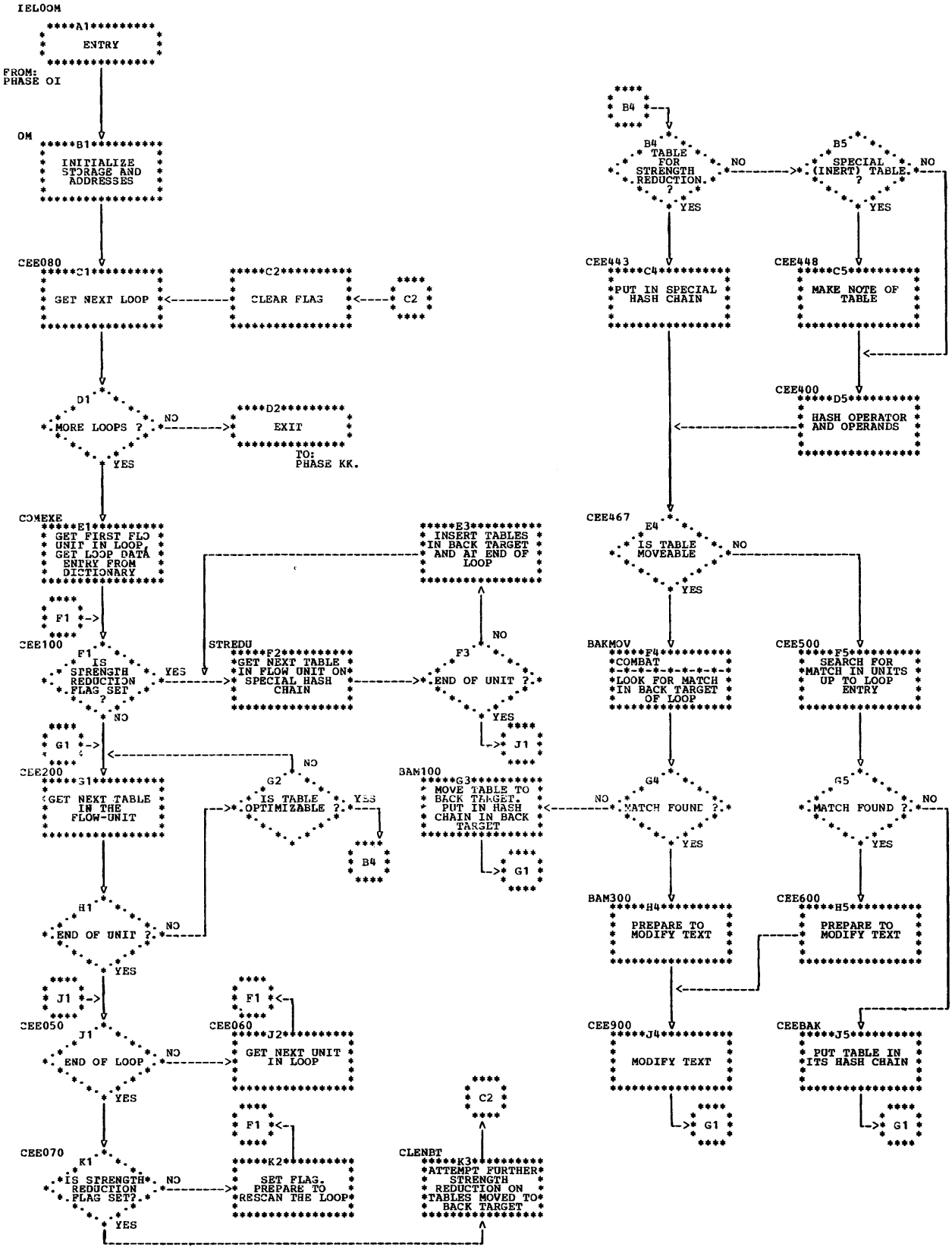


Chart 3.31. Text Optimization Phase (Phase OM)

Built-in Function Processing (Phase KK)

Name	Type	Base registers	Function
IELOKK	CSECT	R6,R7 R8,R9, R5	Entry point to Phase KK.
KK1	R		Initialize registers, storage, and text scan.
BIFSCN	R		Scan to the next text table. If NDX, OFFS, or PTSAT table, then carry out QT processing on IOPND3 and continue the scan. If BIF, PSV, or PSV2 table, then branch to BIFFND. If POWER or ARG table, then branch to relevant routine. Otherwise continue the scan or exit to the next phase.
BIFFND	R		Identify the Bif and branch to its processing routine:
CEILRTN	R		Process CEIL.
CONVRT	R		Process BINARY,DECIMAL,FIXED,FLOAT,PRECISION.
MAXRTN	R		Process MAX,MIN.
MODRTN	R		Process MOD.
ABSRTN	R		Process ABS.
RNDRTN	R		Process ROUND.
SGNRTN	R		Process SIGN.
TRCRTN	R		Process TRUNC.
SPADIM	E		Entry point to ARYBDS from SUM,PROD,ALL, or ANY.
ARYBDS	S		Process HBOUND,LBOUND, and DIM.
SPAART	R		Process SUM, PROD, ALL, ANY. Call SPADIM if Bif to be evaluated inline.
MATH	R		Generate library calling sequence for a mathematical function.
GNRCRT	R		Process Bif argument in an ARG table.
POLYRT	R		Process POLY.
EMPTRT	R		Process EMPTY.
NULLRT	R		Process NULL.
ADDRTN	R		Process ADDR.
FCBRTN	R		Process LINEND,COUNT.
CNJGRT	R		Process CONJG.
REALRT	R		Process REAL.
IMAGRT	R		Process IMAG.
CPLXRT	R		Process COMPLEX.
STATRT	R		Process STATUS.
CMPLRT	R		Process COMPLETION.
TIMERT	R		Process TIME,DATE.
ONCDRT	R		Process ONCODE.
ONLCRT	R		Process ONLOC.
CONDRT	R		Process DATAFIELD,ONFILE,ONKEY,ONSOURCE,ONCHAR,ONCOUNT.
POWERT	R		Process exponentiation.
CRBN1	S		Create a 'nearly 1' binary constant.
BIFEND	S		Clear up after a Bif has been processed, and return.
ALSTGN	S		Generate an ALIST text table.
ARGGEN	S		Generate an ARG text table.
CALLGN	S		Generate a CALL text table.
REFSAV	S		Convert a text table address into a text reference.
SETBAS	S		Determine the arithmetic type of an operand, and set bits.
SETFL	E		Entry point to SETBAS for an operand known to be FLOAT.
SETEPT	S		Calculate the value of EPTNDX.
DEDCRE	S		Create a DED for a 6-byte reference.
FLCONV	S		Compare two float arguments and convert if necessary.
FLCTRG	E		Introduce an intermediate result temporary if the target is not of the correct type.

Continued on next page

ASNINT	S		Generate text to assign an intermediate result temp. to a target.
PLISTG	S		Generate an ALIST table and ARG tables corresponding to
PLISTA	E		6-byte references in OPNDT3.
RDNEXT	S		Scan to the next input text table.
RAHDST	S		Access the next input text table, locking in the current table.
RAHFIN	S		Reset the scan pointer on completion of a 'read ahead' operation.
STKTBL	S		Stack current input table for use by the WRTTBL subroutine.
WRTTBL	S		Access the next table to be output.
GENTXT	S		Generate a text table and write it into the text stream.
TMPSET	S		Process a 6-byte operand contained in the TEMPTB table.
CNSSET	S		Create CONSTB and general dictionary entries for a constant.
QTSET	S		Create a TEMPTB entry.
CRE10	S		Set up a decimal constant 10**N.
CLRFRC	S		Set up constants and QTs to clear the fraction part of a FIXED DEC variable.
CONPREC	S		Determine the storage requirements for a binary constant.
NEWTRT	S		Create new Temp number and move it into the TEMPTB entry.
QTFIND	S		Access the QTTBL entry for a given QT.
QTSQZE	S		Access the QTTBL entry for a given QT, and remove it.
QTDTX	S		Remove from QTTBL any entries corresponding to dead QT operands in a given text table.
LIBSET	S		Set a library module inclusion bit in XLIBSTR.
	T		Tables, transfer vectors, etc. for the phase.
XINIT	CSECT	R9	)
XRFSEQ	CSECT	R9	)
XRFAB	CSECT	R9	)
XNEXT	CSECT	R9	) XROUT
XTXPG	CSECT	R9	)
XNSRT	CSECT	R9	)
XTUNL	CSECT	R9	)
XSTG	CSECT	RA	Private storage for Phase KK.

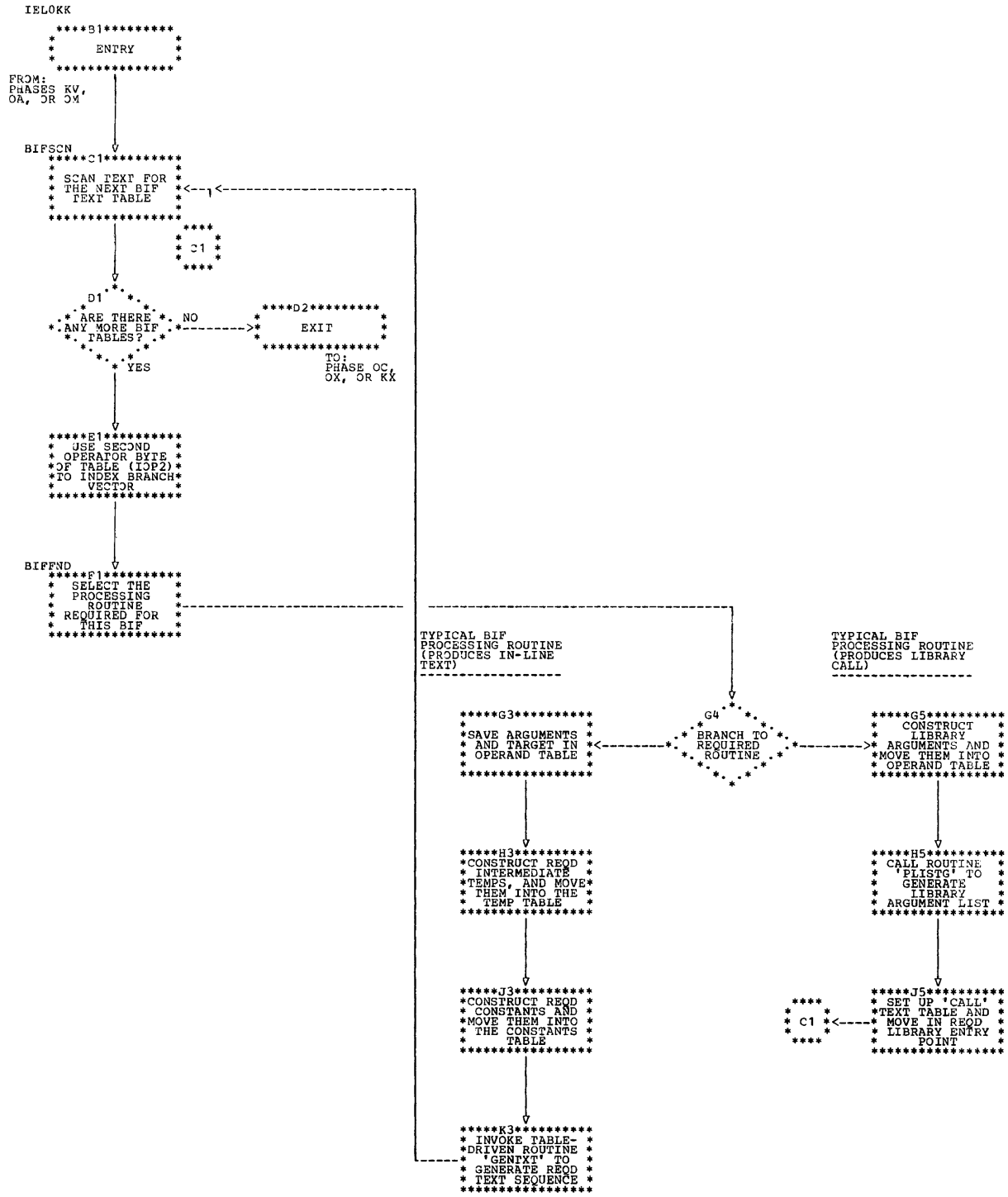


Chart 3.32. Built-in Function Processing Phase (Phase KK)

String Handling Operations - Part 1 (Phase OC)

Name	Type	Base registers	Function
IEL0OC	CSECT	R6,R7	Entry point to root module, OC1, of Phase OC.
OC1	R		Initialize registers, storage, etc.
SCANEXT	R		Scan text table in input text stream.
SCANTEST	S		Check type of text table and branch to appropriate processing routine.
SCANBIFS	S		BIF text table found. Check type of BIF and branch accordingly.
BOOL0	S		Branch to routine BOOLF in module OC2.
TRANSL0	S		Branch to routine TRANSLF in module OC2.
SCANCOM	S		Branch to routine SCANCOMP in module OC2.
SCANCAT	S		CONCAT text table found. Branch to routine CONCAT0.
SCANASSN	S		ASSN or CONV text table found. Branch to routine SCANASS0.
HASHLAB	S		HASH text table found. Get next input text page.
SCANNDX	S		NDX or OFFS text table found. Branch to subroutine QTENT.
SCANPTS	S		PTSAT text table found. Branch to subroutine QTENT.
SCANLAB	S		SL text table found. Save block level.
SCANSNO	S		SN text table found.
OCPIC	R		Process PICTURE assignments and conversions.
SCANASS0	S		Process ASSN or CONV text table.
GENVDA	S		Contains three entry points, as follows:
GENVDA1	E		Generate a GETVDA table using the length of the variable in the source.
GENVDA2	E		Generate a GETVDA table, together with LOAD and COMPARE tables, to get a VDA for the shorter of the maximum length of the source and the current length of the target, and assign the source string to the temporary after obtaining it.
GENVDA3	E		As for GENVDA2, except that no assignment is generated, and the current length of the target, not the maximum, is used in the comparison for the length of the VDA to be obtained.
ASSNSX00	R		Padding routine. Generate OFFSET and MOVE tables for padding CHAR or BIT strings of up to 1536 bytes.
ASSNOP0	R		Generate tables for padding spare bits in a fixed-length bit-string target.
BSSNSXP0	R		Entered when there is less than 8 bytes of padding in a fixed-length bit-string. Generate different types of MOVE tables, depending the number of padding bytes needed.
QTENT	S		Allocate a new space in the list of Q-temps. for a new entry, and test for overflow of the stack.
QTFIND	S		Search the qualified temporary list for a given temporary.
OLAPQ	R		Test to see if two operands may overlap.
GENLIB	S		Generate library calling sequence tables.
CONCAT0	S		Process CONCAT text tables.
GETOPND	S		Get the next operand in a CONCAT text table.
SAVEREF1	S		Save the reference to an operand which contains a temporary created by the CONCAT0 routine.
CLEARSTK	S		Remove a temporary from the stack.
CURLEN	S		Set up the current length of a string in operand 1.
MAXLEN	R		Set up IOPND2 with the maximum length of a fixed or adjustable string.
SCANNED	S		End-of-program routine.
SAVESOR1	R		Used by routine SCANASS0 to save the location of the last table containing the source.
SAVEATR1	R		Used by routine SCANASS0 to save the location of the last table containing the target.

Continued on next page

OC2	CSECT	R6,R9	Entry point to non-root module, OC2.
OC1SR	S		Call subroutines contained in the root module, OC1.
EXOP	R		Exchange operands.
CLRBIT	R		Generate MOVE(0D) text tables to clear the odd bits in the top part of a byte used in string operation.
BOOLF	S		Process BOOL BIF text table.
TRANSLF	S		Process TRANSLATE BIF text table.
IR29	S		Create library routine call.
SCANCOMP	S		Determine the type of the result of a comparison operation, and act accordingly.
GENBC	S		Generate a BC text table using the original first and second operands from a COMP text table.
ALOL	S		Scan the three operands of a string operation and set a switch according to the types of operands.
FILL	S		Generate MOVE(05) or MOVE(06) text tables to fill string operands.
ZTRAN1	T		Internal code-to-EBCDIC translation.
XNXROUT	CSECT		)
XTXPG	CSECT		)
XRFAB	CSECT		)
XNSRT	CSECT		) XROUT
XRFSEQ	CSECT		)
XDIREC	CSECT		)
XLINK	CSECT		)
XMESGR	CSECT		)
XSTG	CSECT	RA	Private storage for Phase OC.

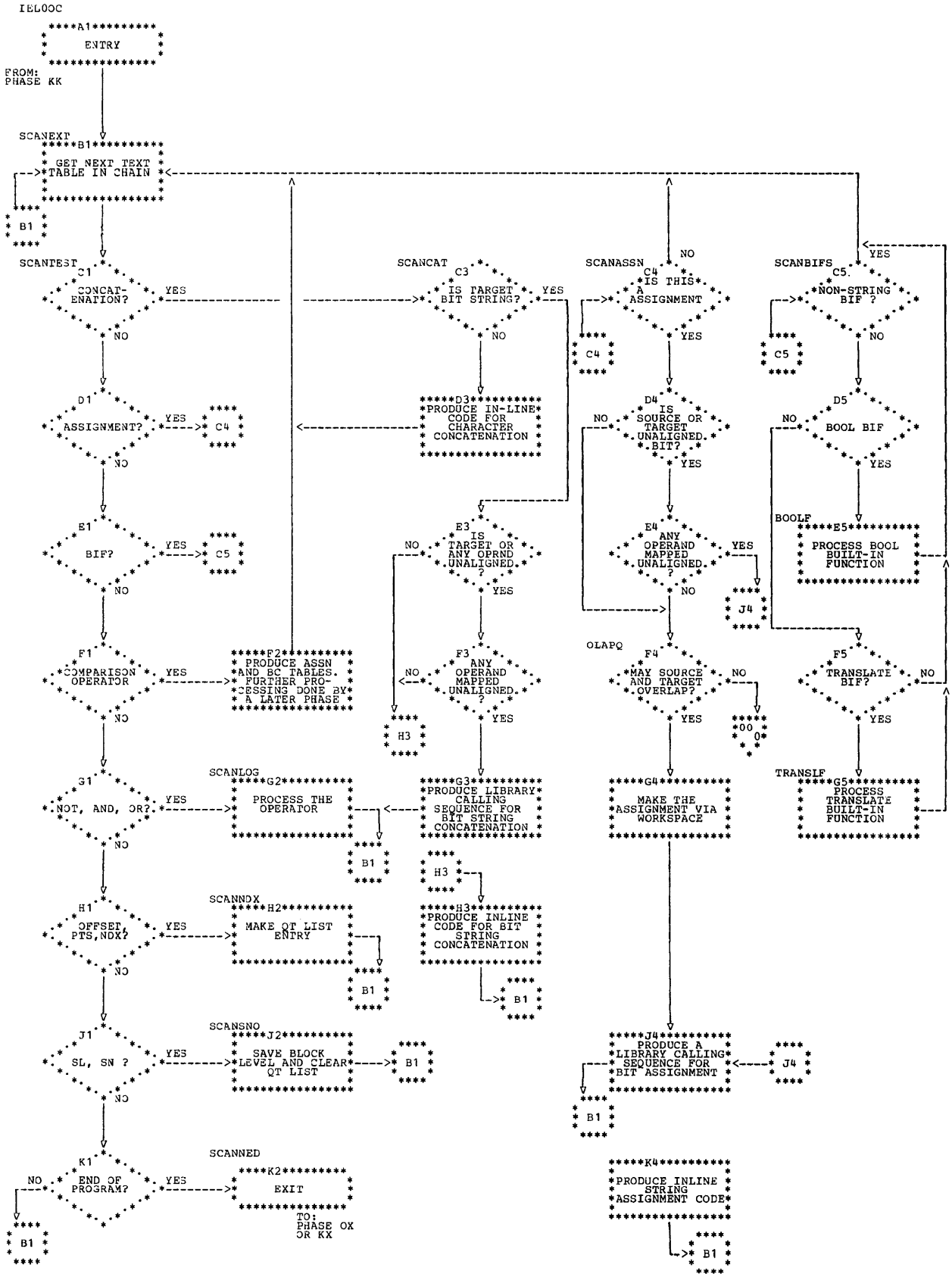


Chart 3.33. String Handling Operations - Part 1 (Phase OC)

String Handling Operations - Part 2 (Phase OX)

Name	Type	Base registers	Function
IEL0OX	CSECT	R6	Entry point to root module, OX1, of Phase OX.
OX1	R		Initialize registers, storage, etc.
SCANEXT	R		Scan text table in input text stream.
SCANTEST	R		Check type of text table and branch to appropriate processing routine.
SCANBIFS	R		BIF text table found. Check type of BIF and branch accordingly.
REPEAT00	R		Branch to routine REPEATF.
LENGTH0	R		Branch to subroutine LENBIF.
VERIFY0	R		Branch to routine VRFNCT.
HIGH0	R		Branch to routine HIGHLOW.
LOW0	R		Branch to routine HIGHLOW.
SCANMOVE	R		Branch to subroutine GENCONS.
SCANNED	R		Exit from phase when end-of-program marker encountered.
SCANNED0	R		Test for compiler-generated subroutines. Branch to routine OCSUB.
STRNGF0	R		Branch to routine STRNGF00.
SCANCOMP	R		Determine the type of the result of a comparison operation, and act accordingly.
GENCAS	R		Generate assignments from constants to variables for the comparison of labels, files, and entries.
STRNGF00	R		Process STRING BIF text table.
REPEATF	R		Process REPEAT BIF text table.
OCSUB	R		Insert the code in GEN text tables for compiler-generated subroutines.
GTSTART	T		Subroutine generation table.
ALOL	S		Scan the three operands of a string operation and set a byte switch according to the types of operands.
HIGHLOW	R		Process HIGH BIF and LOW BIF text tables.
HGOTO	R		Generate GOTO 05 text tables for LTR instructions.
GETVDA	R		Generate GETVDA text tables.
H15	S		Process CONV text tables.
VRFNCT	R		Process VERIFY BIF text tables.
FILL	S		Generate MOVE 05 or MOVE 06 text tables to fill string operands, depending upon the length of the operand in 'target'.
HASHLAB	R		HASH text table found. Get next input text page.
SCANNDX	R		NDX or OFFSET text table found. Branch to subroutine QTENT.
SCANPTS	R		PTSAT text table found. Branch to subroutine QTENT.
SCANLAB	R		SL text table found. Save block level.
SCANSNO	R		SN text table found.
QTENT	S		Allocate a new space in the list of Q-temps. for a new entry, and test for overflow of the stack.
QTFIND	S		Search the Q-temp. list for a given temporary.
LENBIF	S		Process LENGTH BIF text table.
OXNSRT	S		Clear Q-temps. which have been processed.
SCANQT	R		Scan text tables for dead Q-temps. and remove them from the Q-temp. list when found.
OLAPQ	R		Test to see if two operands may overlap. The result of the test is returned in the condition code setting.
CLBITS	S		Generate an OFFSET text table followed by a MOVE 0D table to clear the top bits of an aligned bit string prior to a comparison.
INDEX00	R		Process INDEX BIF text table.

Continued on next page

GENCONS	S		Examine all string operation text tables for cases of adjustable temporary strings. If such a string is found, a KONST table is generated in front of the string operation table.
OC1NSRT	S		Obtain the next text table from the input stream.
GENLIB	S		Generate library calling sequence tables.
TRTAB	T		
ZTRAN1	T		Internal code to EBCDIC translation.
IEL0OX	CSECT	R6,	Entry point to non-root module, OX2.
OX2	R		Initialization routine.
CP001	R		Test if text table is of interesting type. For SINIT and IASSN tables of complex variables, rescan the statement to find the actual initial values.
TMPST1	R		Set up 6-byte operand for temporary operand. Temporary is created as REAL and LOCAL. If COMPLEX is required, change data type and code on return.
TABAD	T		Table of addresses of text table sequences.
XMESGR	CSECT		)
XTXPG	CSECT		)
XNXROUT2	CSECT		)
XNXROUT1	CSECT		)
XRFSEQ	CSECT		) XROUT
XDIREC	CSECT		)
XINIT	CSECT		)
XRFAB	CSECT		)
XLINK	CSECT		)
XSTG	CSECT	RA	Private storage for Phase OX.

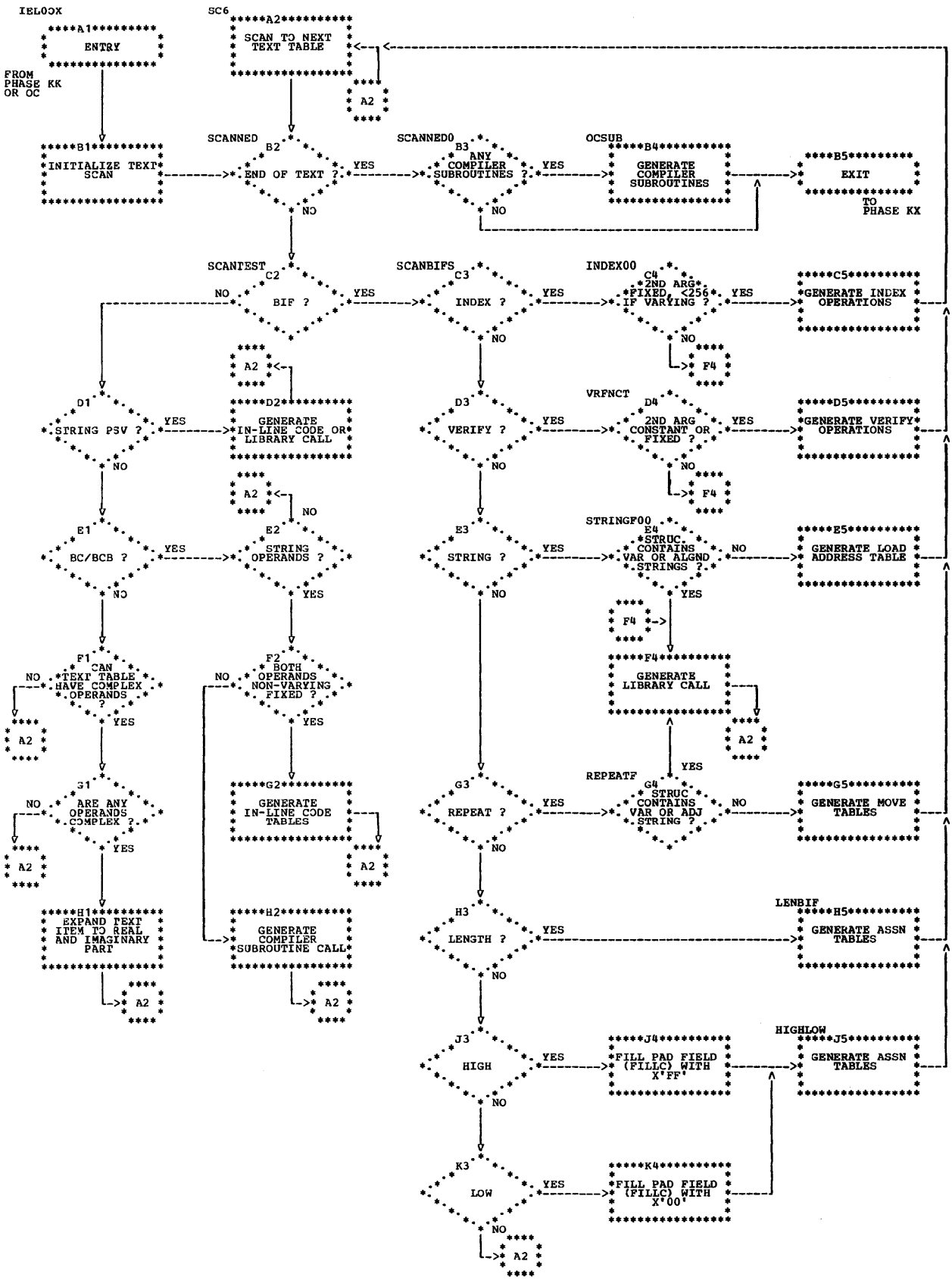


Chart 3.34. String Handling Operations - Part 2 (Phase OX)

Name	Type	Base registers	Function
IEL0KX	CSECT	R5,R6	Entry point to Phase KX.
KXA	R		Initialize registers, storage, scan text, etc.
SCAN	R		Scan the input text stream to the next relevant table, identify it, and branch to its processing routine.
ASSIGN	R		Process an ASSN table.
SUBTRACT	R		Process a MINUS table.
ADDITION	R		Process a PLUS table.
MULTIPLY	R		Process a MULT table.
DIVISION	R		Process a DIVIDE table.
GOTOLAB	R		Process a GOTO table.
DINCLAB	R		Process a DINC table.
SUBSLAB	R		Process a SUBS table.
FUHLAB	R		Process a FLWUNT table.
STATHEAD	R		Process a statement header table.
HASLAB	R		Process a HASH table.
TERMINAT	R		Process an ENDPRG table.
SETFLTP	R		Process a float precision expression.
CONVZBOT	S		Produce text tables to generate inline code for FIXED BIN to BIT conversion.
CONVDBOT	S		Produce text tables to generate inline code for FIXED DEC to BIT conversion.
CONVFBOT	S		Produce text tables to generate inline code for FLOAT to BIT conversion.
CONVBXOT	S		Produce text tables to generate inline code for BIT to FIXED BIN conversion.
CONVBDOT	S		Produce text tables to generate inline code for BIT to FIXED DEC conversion.
CONVBFOT	S		Produce text tables to generate inline code for BIT to FLOAT conversion.
CONVCBOT	S		Produce text tables to generate inline code for CHAR to BIT conversion.
CONVBCOT	S		Produce text tables to generate inline code for BIT to CHAR conversion.
CBAOTS	S		Check a BIT source for inline conversion suitability.
CABOTS	S		Check a BIT target for inline conversion suitability.
CCAOTS	S		Check a CHAR source for inline conversion suitability.
CACOTS	S		Check a CHAR target for inline conversion suitability.
CONVEX	S		Expand one input conversion to two fundamental types.
TXTIN1	S		Insert a new table into the text stream.
TXTIN2	E		
STPATT	S		Load an edit pattern into the dictionary.
OFFSHOV1	S		Insert OFFS and MOVE tables into the text stream after the current text table.
OFFSHOV2	E		
XINIT	CSECT	R9	)
XMESG	CSECT	RF	)
XRFSEQ	CSECT	R9	)
XRFAB	CSECT	R9	)
XDIREC	CSECT	R9	) XROUT
XNXROUT1	CSECT	R9	)
XNXROUT2	CSECT	R9	)
XTXPG	CSECT	R9	)
XNSRT	CSECT	R9	)
XTUNL	CSECT	R9	)
XLINK	CSECT	R9	)

Continued on next page

XSTG	CSECT	RA	Private storage for Phase KX. This contains a storage region (COMSTG) common to all four modules of the phase, as well as local automatic and explicit storage for each of the modules.
KXC	CSECT	R7,R8	Second module of Phase KX.
CONVERT	R		Test the source attributes, and branch to the relevant routine. These routines (CONVB to CONVX) contain the library call processors, and branch to the special case routines for inline code where applicable.
CONVB	R		Handle conversion from a BIT source.
CONVC	R		Handle conversion from a CHAR source.
CONVD	R		Handle conversion from a FIXED DEC source.
CONVE	R		Handle conversion from a FLOAT DEC PIC source.
CONVF	R		Handle conversion from a FLOAT source.
CONVP	R		Handle conversion from a FIXED DEC PIC source.
CONVQ	R		Handle conversion from a CHAR PIC source.
CONVX	R		Handle conversion from a FIXED BIN source.
CONVXFOT	S		Produce text tables to generate inline code for FIXED BIN to FLOAT conversion.
CONVFXOT	S		Produce text tables to generate inline code for FLOAT to FIXED BIN conversion.
CONVDFOT	S		Produce text tables to generate inline code for FIXED DEC to FLOAT conversion.
CONVFDOT	S		Produce text tables to generate inline code for FLOAT to FIXED DEC conversion.
PREASSN	S		Insert an ASSN table into text in front of a CONV table, to generate the correct float precision.
KXE	CSECT	R7,R8	Third module of Phase KX.
CONVXDOT	S		Produce text tables to generate inline code for BIN to DEC conversion.
CONVDXOT	S		Produce text tables to generate inline code for DEC to BIN conversion.
CONVCDOT	S		Produce text tables to generate inline code for CHAR to FIXED DEC conversion.
CONVCFOT	S		Produce text tables to generate inline code for CHAR to FLOAT conversion.
CONVCXOT	S		Produce text tables to generate inline code for CHAR to FIXED BIN conversion.
CONVDCOT	S		Produce text tables to generate inline code for DEC to CHAR conversion.
CONVXCOT	S		Produce text tables to generate inline code for BIN to CHAR conversion.
CONVFCOT	S		Produce text tables to generate inline code for FLOAT to CHAR conversion.
KXG	CSECT	R7,R8	Fourth module of Phase KX.
CONVDPOT	S		Produce text tables to generate inline code for DEC to FIXED DEC PIC conversion.
CDPOT04	S		As in CONVDPOT, for FIXED DEC PIC Type 1.
CDPOT40	S		As in CONVDPOT, for FIXED DEC PIC Type 2.
CDPOT50	S		As in CONVDPOT, for FIXED DEC PIC Type 3.
CONVXPOT	S		Produce text tables to generate inline code for BIN to FIXED DEC PIC conversion.
CONVFPOT	S		Produce text tables to generate inline for FLOAT to FIXED DEC PIC conversion.
CONVPDOT	S		Produce text tables to generate inline code for FIXED DEC PIC to DEC conversion.
CONVPXOT	S		Produce text tables to generate inline code for FIXED DEC PIC to BIN conversion.
CONVPFOT	S		Produce text tables to generate inline code for FIXED DEC PIC to FLOAT conversion.

Continued on next page

CONVPPOT	S		Produce text tables to generate inline code for FIXED DEC PIC to FIXED DEC PIC conversion (via FIXED DEC).
PICTRANS	S		Convert a picture specification in the dictionary to the form required by the library at execution time.
PCHECK	S		Scan a picture specification for inline code suitability and classification.
MVIS	S		Insert OFFS and MOVE tables in the text stream, to generate MVI instructions for picture punctuation.

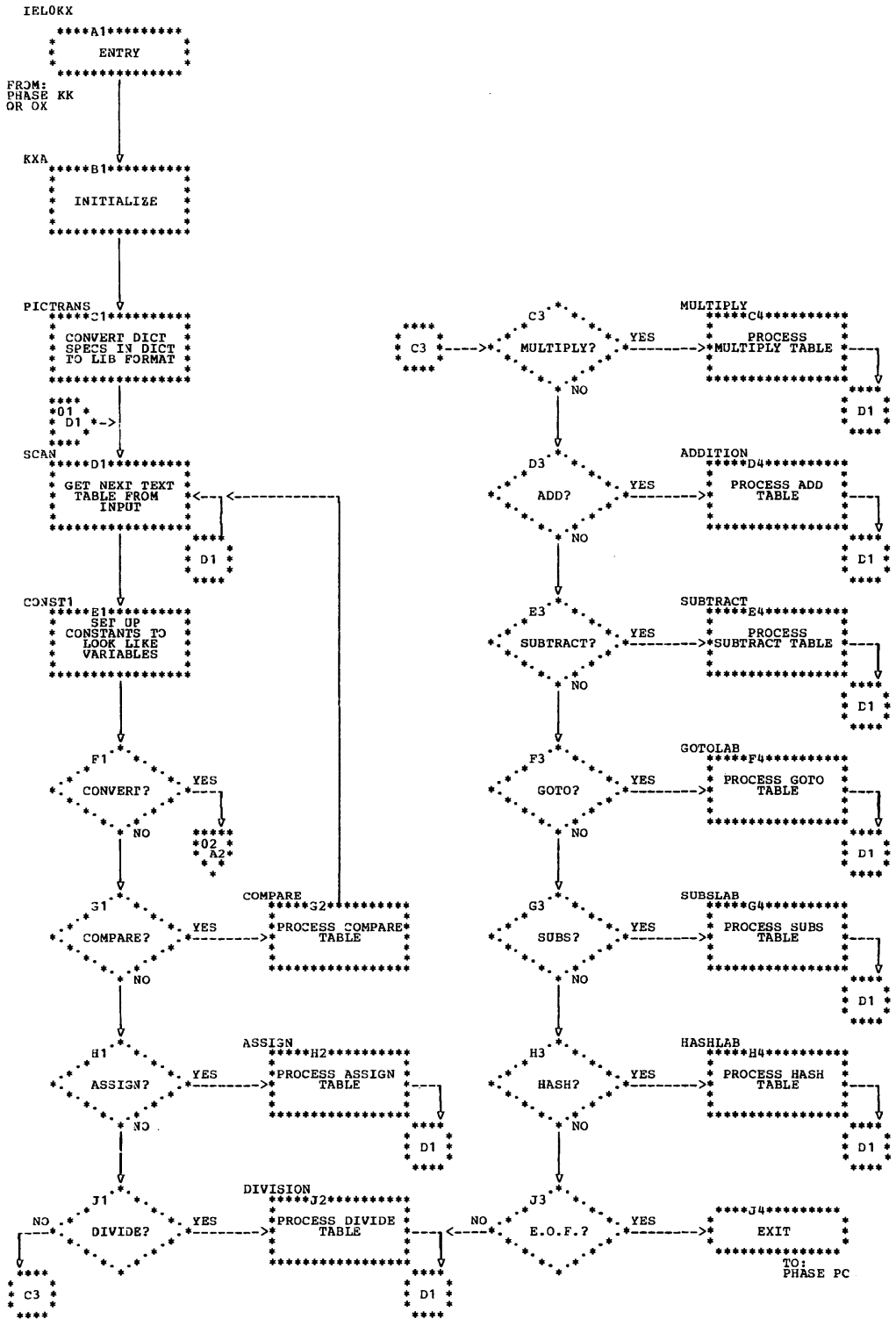


Chart 3.35. (Part 1 of 2). Arithmetic Operations and Conversions Phase (Phase KX)

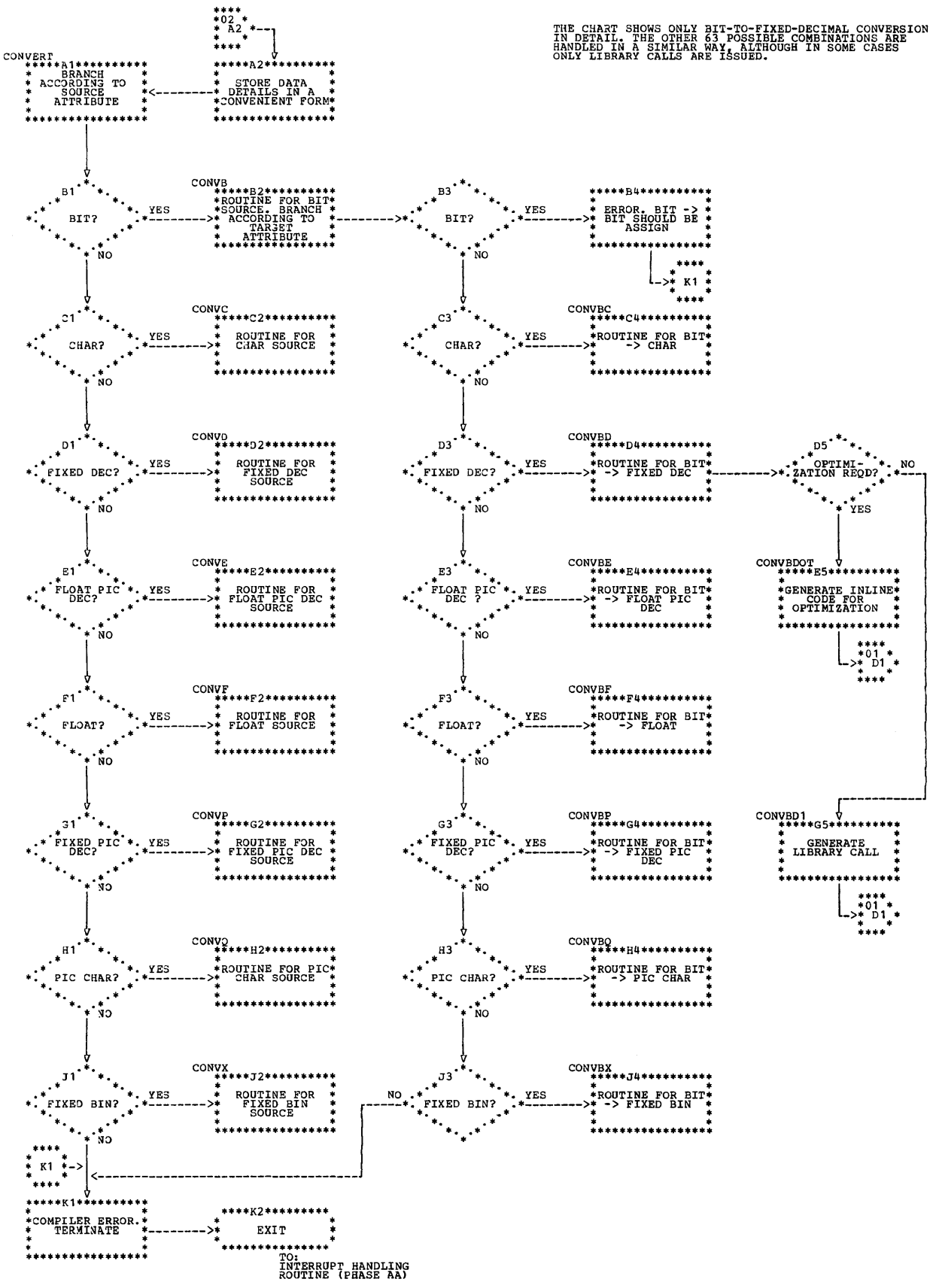


Chart 3.35. (Part 2 of 2). Arithmetic Operations and Conversions Phase (Phase KX)

Name	Type	Base registers	Function
ZDSECT	DSECT		Fields in Type 2 text tables.
PCPDSC	DSECT	R3	Format of entries in the pseudo constants pool, block field, QT table, AID/ENDAID table, and temporary replacement stack.
IEL0PC	CSECT	R6-R9	Entry point to Phase PC.
PC1000	R		Phase initialization.
PC2000	R		Scan the text tables of the input text stream and pass control to the appropriate processing routine.
PC2200	R		Start of block processor.
PC2300	R		Process symbol tables for all known variables.
PC2400	R		Process symbol table for a given identifier.
PC2500	R		Test if the text table indicates that an object-time DED is required (other than for a symbol table).
PC2600	R		FED construction processor.
PC2700	R		Target of constant conversion detects for Phase PA.
PC2800	R		Static initial processing routine (for Phase PA). Determine size of storage required for static initial variables, offset within structure storage, etc., for use by Phase PA in the mapping of static initial storage.
PC2900	R		Q-temp. table entry creation routine.
PC2910	R		Q-temp. table entry annihilation routine.
PC2920	R		KONST text table processing routine.
PC2930	R		ONCB testing routine.
PC2940	R		Argument reordering routine.
PC2950	R		Address temporary extinction routine.
PC2960	R		VDA length operand routine.
PC2970	R		Statement options flag byte clearing routine.
PC2980	R		GOOB text table processing routine.
PC2990	R		Locator processing routine.
PC3000	R		Scan the variables dictionary, identifying those variables which require symbol tables by virtue of the fact that they are declared in blocks currently active when a PUT DATA, GET DATA, or SIGNAL CHECK of all known variables occurs.
PC3100	R		Non-structured variable processing routine.
PC3200	R		Structured variable processing routine.
PC3300	R		Create a locator-required text table (ARG 02).
PC4000	R		Round the pseudo constants pool to a 4-byte boundary, and write in the pseudo constants pool the requisite number of dummy symbol table elements (as counted by routine PC3000).
PC5000	R		Scan the variables and general dictionaries, building symbol tables for those identifiers requiring them.
PC5100	R		Build the symbol tables for external variables, and, if required, build the DEDs and fill the dummy symbol table elements.
PC5200	R		Entered when a variable is encountered which requires a symbol table element, and which is in a different block to the previous such variable.
PC5300	R		End-of-variable processing routine.
PC5400	R		Scan down the chain of general dictionary references of label constants requiring symbol tables, and build the tables.
PC5500	R		Scan down the chain of general dictionary references of entry points requiring symbol tables, and build the tables.

Continued on next page

PC6000	R		Phase termination routine. Complete the creation of the pseudo constants pool and pass control to the next phase.
SRTM00	S		Handle references to temporaries arising from constants.
SRQT00	S		Handle Q-temp. references.
SRTP00	S		Test if a given member of the variables dictionary requires a long or a short symbol table, and if identifier, if BASED or DEFINED, is allowed by this compiler for PUT DATA.
SRCN00	S		Determine which library modules (for conversions) are required for the variables in PUT/GET DATA.
SRLC00	S		Test if a given member of the variables dictionary requires a locator.
SRLD00	S		Create compile-time DEDs for literal constants.
SRMJ00	S		Insert ARG (02) text tables into the output text stream to inform Phase PA that the variable referenced in operand 1 requires a locator and a descriptor.
SRTD00	S		Entered when a text table is encountered in which operand 1 is flagged as requiring a DED. This subroutine sets up a 3-byte compiler DED and assigns a dictionary reference to the field VARREF to be passed to subroutine SRBD00.
SRBD00	S		Control the building of object-time DEDs.
SROB00	S		Convert a 3-byte compiler DED in the field DEDSTG into object form in the pseudo constants pool workspace.
SROD00	S		Output DED and FED entries into the pseudo constants pool.
SRSE00	S		Called when a particular variable requires a symbol table element. It calls subroutine SRST00, and also fills in the dummy symbol table element created by routine PC4000.
SRSI00	R		Static initial allocation subroutine.
SRST00	S		Construct symbol tables for variables, label constants, and entry points.
SRPD00	S		Insert dummy entries in the pseudo constants pool for rounding purposes.
ZTRANI	T		Used in subroutine SRST00 to translate the symbol table identifier name from the internal representation into EBCDIC.
XBREAK	CSECT		)
XNXROUT	CSECT		)
XTXPG	CSECT		)
XRFAB	CSECT		)
XDSTAT	CSECT		) XROUT
XNSRT	CSECT		)
XBRIC	CSECT		)
XMESGR	CSECT		)
XDIREC	CSECT		)
XLINK	CSECT		)
XSTG	CSECT		Private storage for Phase PC.

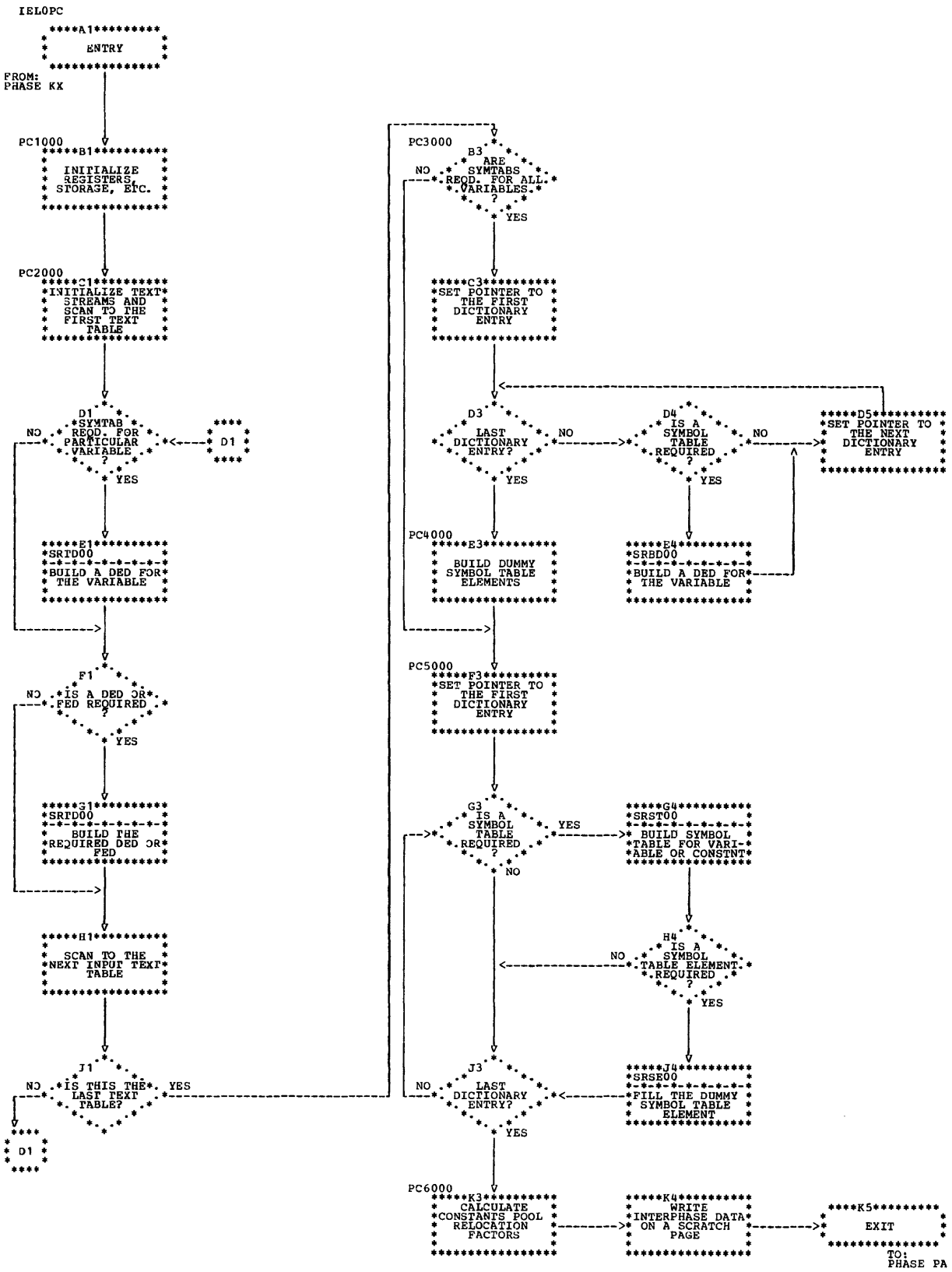


Chart 3.36. Symbol Table Resolution Phase (Phase PC)

Name	Type	Base registers	Function
YDSECT	DSECT	R4	1. Layout of FCB entries in the general dictionary. 2. Layout of CONDITION condition entries in the general dictionary.
YDSECT2	DSECT	R5	Layout of entries in the chaining fields.
CONDES	DSECT	R3	1. Format of entries on the constant descriptor pages. 2. Temporary stack format.
IEL0PA	CSECT	R6-R9	Entry point to Phase PA.
PA1010	R		Phase initialization.
PA2000	R		Perform a preliminary scan of the variables dictionary allocating locators for static external items and anchor slots for controlled variables.
PA3000	R		Text scanning routine. Control the scan of the input text. Pass control to one of the processing routines according to the type of text table encountered.
PA3100	R		Target DED determination routine. Determine the target DED required for possible constant conversions. The DED depends upon both the text table and operand types.
PA3100	E		Entry point for BC text tables.
SRDS00	S		Page discarding subroutine.
SROP00	S		Operand examination subroutine.
SRFL00	S		File allocation subroutine.
SREN00	S		Environment block allocation subroutine.
SRLC00	S		Locator/descriptor allocation subroutine.
SRDM00	S		Descriptor mapping routine.
SRC000	S		Subroutine to allocate all types of PCP entry.
SRC100	E		Output space in the pseudo constants pool for a STATIC INITIAL variable.
SRC200	E		Allocation of storage for RECORD and KEY descriptors.
SRC300	E		Entry point for when an entry is required in STATIC for a label.
SRC310	E		Entry point for when an entry is required in STATIC for a CONDITION CSECT.
SRC320	E		Entry point for when an entry is required in STATIC for a STATIC ONCB.
SRC330	E		Entry point for when an adcon is required for symbol table addressing.
SRC350	E		Entry is required for padding purposes to correct alignment.
SRC360	E		Adcon required for addressing an external variable.
SRC400	E		Entry point to assign the information common to different sorts of adcon and determine which sort is required.
SRC420	E		Adcon for variable.
SRC425	E		Adcon for constant.
SRC430	E		Adcon for DED.
SRC435	E		Adcon for temp. or Q-temp.
SRC440	E		Adcon for record/key descriptor.
SRC445	E		Adcon for FCB.
SRC450	E		Adcon for locator.
SRC455	E		Adcon for symbol table.
SRC460	E		Adcon for CONDITION condition.
SRC465	E		Adcon for symbol table element list.
SRC470	E		Adcon for controlled variable anchor slot.
SRC475	E		Adcon for label.
SRC485	E		Adcon for entry name.
SRC490	E		Adcon for FETCH control block.
SRC500	E		File - FCB.
SRC510	E		File - environment block.
SRC530	E		File - environment block constant.
SRC540	E		File - constant part of DTF.
SRC550	E		File - variable part of DTF.

SRC560	E		File - adcon for external file.
SRC570	E		File - generate padding for alignment.
SRC580	E		File - buffer if OPEN has been optimized.
SRC600	E		Convert constants in the text into the correct form for insertion in the pseudo constants pool.
SRC700	E		Convert constants in the dictionary into their required form for output to the pseudo constants pool.
SRC800	E		Assign the initial values to storage previously allocated at SRC100 for static initial variables.
SRC900	E		Chaining routine common to all constants. Functions include offset determination, constant descriptor creation, duplicate checking, and output of constant descriptor.
SRSM00	S		Static initial mapping subroutine.
SRLB00	S		Subroutine to call the library to perform a constant conversion.
SRCB00	S		Subroutine to perform character-to-bit conversion.
SRLM00	S		Subroutine to perform long move operations.
SRLZ00	S		Subroutine to clear long fields; either to zero or to blanks.
SRBK00	S		Subroutine to output entries (constant descriptors or PCP entries) in the second output text stream.
SRFB00	S		Subroutine to access the dictionary.
PA3200	R		Process HASH and FLWUNIT text tables.
PA3300	R		Process SINIT, IASSN, AID, and ENDAID text tables.
PA3400	R		Process ALIST, ARG, and CALL text tables.
PA3500	R		ON statement handling routine. Process ONS text tables.
PA3600	R		Process FETCH/RELEASE statements. (Applies to the OS version of the compiler only.)
PA3700	R		Operand examination routine.
PA3800	R		End of text table processor.
PA4000	R		Forward chaining routine.
PA5000	R		Pseudo constants pool creation routine.
PA6000	R		End-of-phase routine.
SRFL00	S		File allocation subroutine.
XRFAB	CSECT		)
XDSTAT	CSECT		)
XBREAK	CSECT		) XROUT
XNXROUT	CSECT		)
XTXPG	CSECT		)
XMESGR	CSECT		)
XDIREC	CSECT		)
XSTG	CSECT		Private storage for Phase PA.

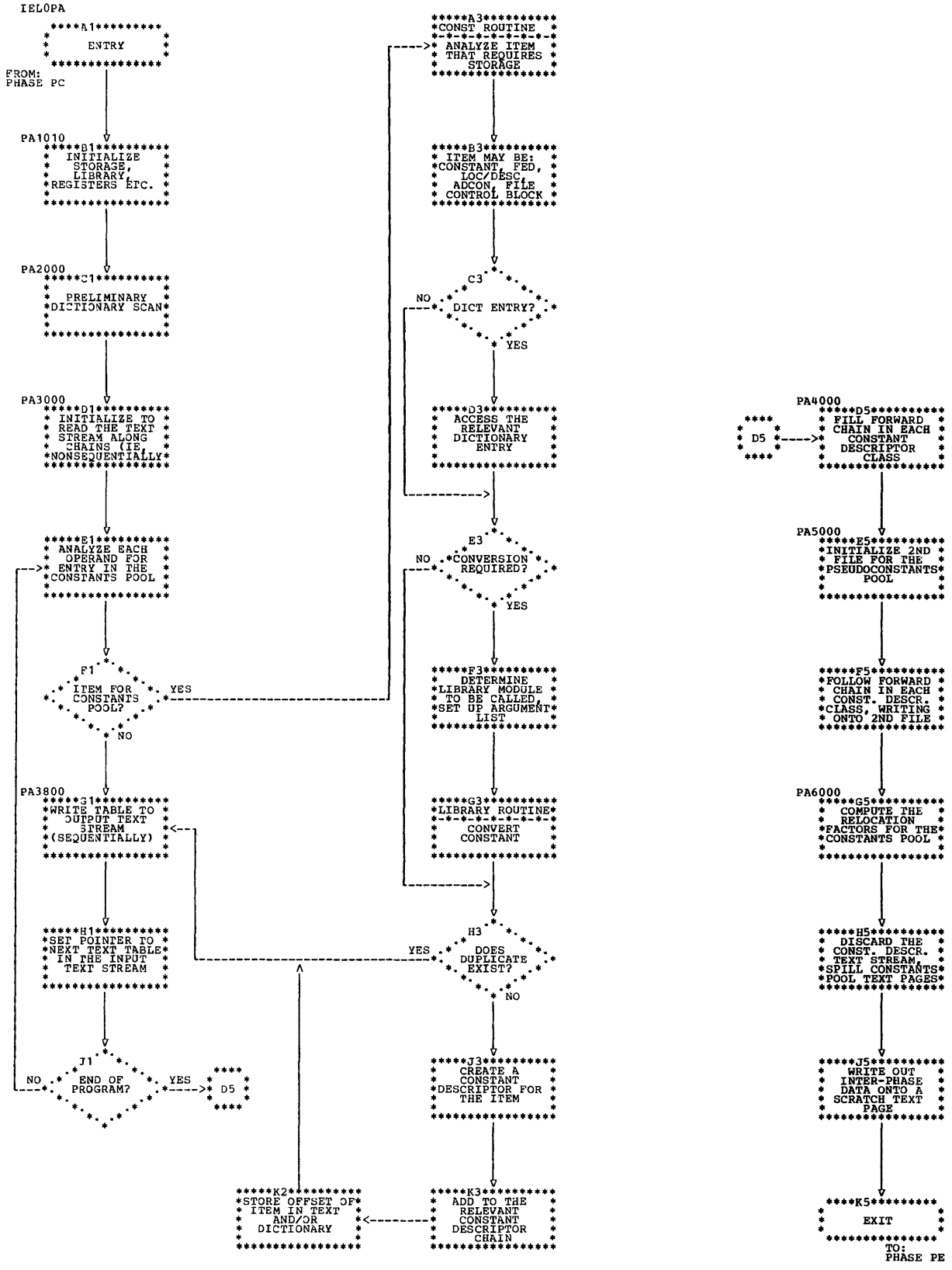


Chart 3.37. Constants Analysis Phase (Phase PA)

Name	Type	Base registers	Function
YSDICT	DSECT	R3	Storage dictionary structure.
OFFTBLE	DSECT	R6	Layout of storage counters for each DSA.
IEL0PE	CSECT	R7-R9	Entry point to Phase PE.
PE	R		Initialization routine.
NXT	R		Scan the variables dictionary and examine the code byte of every entry for data variable. Branch to ADAVAR or SDAVAR, depending on the type of variable.
NXTP	S		End of dictionary page detected. Discard it.
ADAVAR	R		Create a storage dictionary entry for the data variable if it is AUTOMATIC, BASED, CONTROLLED, DEFINED, or a parameter.
TST8	R		Allocate a storage offset in the DSA for the variable. At this point, test if the variable requires 8-byte alignment.
TST4	R		Test if the variable requires 4-byte alignment.
TST2	R		Test if the variable requires 2-byte alignment.
TST1	R		Test if the variable requires 1-byte alignment.
TSTB	R		Test if the variable requires bit alignment.
CONT	R		After the variable offset has been stored in the storage dictionary, and if the table of offsets has been updated for the current block, store the updated table in the second file, and the reference to the table in the list in XSTG.
DEFND	S		Process DEFINED variable.
CTLD01	S		Process CONTROLLED variable.
PARAM	S		Make storage dictionary entry for a parameter.
STARAUT	S		Make storage dictionary entry for AUTOMATIC structure or array.
STRUC	S		Process AUTOMATIC structure.
SKNTRY	S		Build skeleton storage dictionary entry.
LOCDESC	R		Allocate storage in the DSA for the locator, and store the locator's offset in the storage dictionary entry.
CMNLD	S		Determine if storage for a descriptor is required.
SDAVAR	R		Create a storage dictionary entry for the data variable if it is STATIC.
S8B	R		Allocate a storage offset in STATIC for the variable. At this point, test if the variable requires 8-byte alignment.
TST4S	R		Test if the variable requires 4-byte alignment.
TST2S	R		Test if the variable requires 2-byte alignment.
TST1S	R		Test if the variable requires 1-byte alignment.
TSTBS	R		Test if the variable requires bit alignment.
STARST	S		Make a storage dictionary entry for a STATIC structure or array.
STINIT	S		Make a storage dictionary entry for a STATIC INITIAL item.
STEXT	S		Make a storage dictionary entry for a STATIC external item.
CONTS	R		Bump the dictionary reference to the next entry. Mark the next available location in the storage dictionary with the end-of-page marker.
NDICT	R		Scan the RECORD/KEY descriptor chain in the general dictionary. Reserve 8 bytes in AUTOMATIC storage for each RECORD/KEY descriptor that requires it. Store the offset of this storage in the descriptor entry in the general dictionary.

Continued on next page

EOTRK	R		Calculate the total size of the variables in STATIC storage after the last entry in the variables dictionary has been processed.
OFFLP	R		Create a base table for variables in AUTOMATIC storage. Calculate the number of regions of AUTOMATIC storage in each procedure entry in the table and assign the appropriate number in the base table.
NONCST	S		Test if decimal arithmetic storage is required by final assembly.
LIBADC	R		Scan the bit string XLIBSTR in XCOMM, and count the number of bits set on.
CGSADC	R		Scan the bit string XCOMSTR in XCOMM, and count the number of bits set on.
SCAN2	R		Scan through the storage dictionary and relocate the offsets by the relocation factors previously calculated.
LDRTN	R		if the storage dictionary entry contains a locator calculate the number which must be located.
REGN1	S		Test if a descriptor is present, and relocate accordingly.
NODSCR	R		Process AUTOMATIC variable. The following subroutines determine the storage class and relocate the variable according to its alignment:
ABP8	S		Variable is 8-byte aligned.
ABP4	S		Variable is 4-byte aligned.
ABP2	S		Variable is 2-byte aligned.
ABP1	S		Variable is 1-byte aligned.
ABPB	S		Variable is bit aligned. Bit variables are allocated to Class A storage.
SC2A1	R		Relocation routine for AUTOMATIC variables -- Class A.
SC2B1	R		Relocation routine for AUTOMATIC variables -- Class B.
SC2C1	R		Relocation routine for AUTOMATIC variables -- Class C.
ABPARM	R		Calculate the first parameter base.
TMPVAR	R		Allocate a base for a temporary operand.
COMNA	R		Create an entry in the byte array for the variable.
AUTAR	R		Relocate an AUTOMATIC array.
STRUC2	R		Process a structure.
STROKC	E		Entry point to STRUC2 if the structure is CONTROLLED.
STROKD	E		Entry point to STRUC2 if the structure is DEFINED.
CHKREL	S		Check relocated item for addressability. '
STAT	R		Relocate the offset by the relocation factor calculated previously if the entry in the storage dictionary is flagged as STATIC. The following subroutines determine the storage class and relocate the variable according to its alignment:
SBP8	S		Variable is 8-byte aligned.
SBP4	S		Variable is 4-byte aligned.
SBP2	S		Variable is 2-byte aligned.
SBP1	S		Variable is 1-byte aligned.
SBPB	S		Variable is bit aligned.
SC2A1S	R		Relocation routine for STATIC variables -- Class A.
SC2B1S	R		Relocation routine for STATIC variables -- Class B.
SC2C1S	R		Relocation routine for STATIC variables -- Class C.
RSTIC	S		Compute base number and offset for locator and descriptor.
SLDLP	S		Compute storage base and offset in STATIC after relocation.
STATAR	R		Relocate a STATIC array.
RELSIN	R		Relocate a STATIC INITIAL item.
CTLD02	R		Relocate the STATIC INTERNAL adcon for the CONTROLLED variable.
COMN	R		Assign the relocated offset to the storage dictionary.
SCNRKD	R		After relocation of the offsets to the storage dictionary, scan the RECORD/KEY desciptor chain and relocate offsets. Test if there are any entries which require location.
BITRVO	S		Locate the virtual origin of STATIC or AUTOMATIC bit arrays.

Continued on next page

SCAN3	R		Scan the storage dictionary to resolve DEFINED references. Test if DEFINED flag is set to determine whether or not a further scan of the storage dictionary is required.
SCAN33	S		DEFINED dictionary entry found.
TXT000	R		Controls scan through text. TXT010, TXT020 etc., represent pieces of code executed for different classes of text tables.
TMPACC	S		Accesses information for any given temporary.
TMPLEN	S		Determines lengths of temporary operands.
ENDPH	R		Test if offset tables exist on a set of text pages. If so, these text pages can be discarded. Make the last page referenced spillable and pick up the reference to the first page. Follow the text page chain, discarding the text pages.
XRFAB	CSECT		)
XRFSEQ	CSECT		)
XBREAK	CSECT		) XROUT
XTXPG	CSECT		)
XDIREC	CSECT		)
XSTG	CSECT		Private storage for Phase PE.

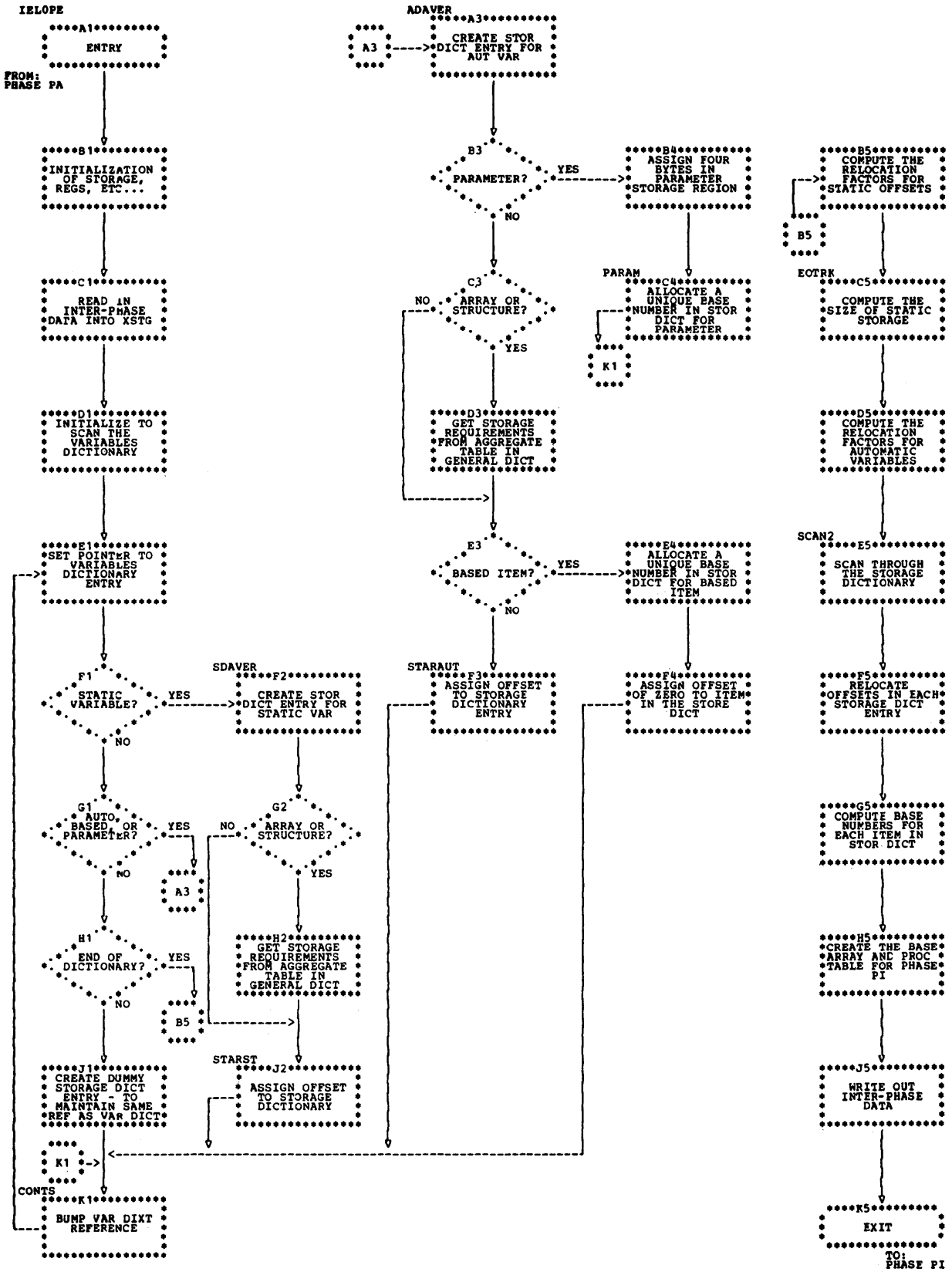


Chart 3.38. Storage Allocation Phase (Phase PE)

Addressing of Storage (Phase PI)

Name	Type	Base registers	Function
YSDICT	DSECT	R3	Storage dictionary structure.
QTEL	DSECT	R2	Layout of QT table element.
SKDS	DSECT		Stack description field DSECT.
IELOPI	CSECT	R6-R9	Entry point to Phase PI.
IELOPI	R		Phase initialization.
TXTLP	R		Scan the text, examining every text table for variables which may require addressing code.
TOP1	R		Test operand 1 for a constant, a tempory operand, a DED, or a variable.
TOP2	R		Test operand 2 for a constant, a temporary operand, a DED, or a variable.
TOP3	R		Test operand 3 for a variable.
ONTXT	R		Process ON text tables.
CNVTBL	R		When text table is a CONV table, check to see if a library call is involved.
LLADR	R		Entered when a LADDR(02) text table has been found.
MOVEPAR	R		Entered when a MOVE(01) text table has been found.
SETENV	R		Set the environment for entry points.
RELADC	R		Relocate entry point address constants.
EOTT	R		Test the text table operator for end-of-program.
BCOFFT	R		Test operand 2 for an offset from the DSA. If it is an offset, fill in the base number.
SHDR	R		Process PROC, BEGIN, or ON-BEGIN text tables.
DSAONS	R		Generate a text table in the prologue to allow the final assembly stage to chain together the dynamic ONCBs.
ADDT	R		Create a BADDR text table to enable the register allocation phase (QA) to address the temporary storage area at the end of the DSA.
BLK0R	S		Output the relocation factors for temporary storage.
PRC3	R		If required, output the addressing information for this block.
PGERTN	R		Set R4 to point at the base table entry for the current block.
PROLG	R		Generate text tables in the prologue to initialize AGGREGATE and STRING locators.
MVRTN	S		Create a text table to move locator skeleton from STATIC to AUTOMATIC.
PROLG2	R		Create text tables in the prologue to initialize RECORD and KEY descriptors in AUTOMATIC.
FNDQT	S		Search the QT stack for a Q-temp., or space for a Q-temp.
MODBSE	S		Modify offsets to module 4096.
GENBAD	S		Generate a BADDR(06) text table for indirect addressing.
OUTTXT	S		Generate statements from ADDTXT.
OUTR1	S		Generate BADDR(08) or (09) text tables.
FLWNT	R		Entered if a flow unit header is encountered in the text scan.
QTLCHK	S		Check whether locators are required for Q-temps.
ONPCHK	R		Test for OFFS text table. If found, check if the total offset if less than 4096.
OP2QT	R		Operand 2 is a Q-temp. Search OFFLST for an entry qualified by the Q-temp.
OFFCHK	R		Examine each operand for a Q-temp. marked as last use. If one is found, then check if it appears in OFFLST, and delete it.
BLDT	R		Build OFFS or NDX text table from the information in the QT stack.

Continued on next page

ADDRT	R		Change OFFS text table to (ASSN + NDX) table. If the offset is a literal number, change OFFS table to (LA + NDX).
TALQT3	S		Entered if operand 3 is a Q-temp., in any text table except OFFS and PTSAT. Test if code is required to align an unaligned variable.
LIBADC	S		Scan the bit string XLIBSTR in XCOMM and create a 2-byte entry in an output table for each bit in the string.
CGSRTN	S		Scan the bit string XCOMSTR in XCOMM and create a 2-byte entry in an output table for each bit set on in the string.
CGSTXT	R		Entered if the text table is a CGS table. Fill in the offset of the address constant for the compiler-generated subroutine.
LIBCNV	S		Test if the current text table contains a call to a library routine. If so, take the appropriate action.
CMN	R		Entered if one of the operands in the text table being analyzed is a constant, or a STATIC or AUTOMATIC variable.
CONST	R		If the operand references a constant, relocate the offset by the relocation factor passed by the constants phase (PA).
TSIZ	S		Test if the relocated offset exceeds 4095 bytes.
RKDSC	S		Entered if the operand references a RECORD/KEY descriptor.
DECWSP	S		If the operand contains an offset in decimal workspace, fill in the base number of the workspace. If the operand references an offset in the DSA, fill in the DSA's base number.
TPV	R		Test if the text table operand is a STATIC or an AUTOMATIC variable.
ARQD	R		Set bit in the addressing vector to indicate the addressing required.
ADLOC	R		Address either locators or descriptors.
INDAD	S		Used for all indirect addressing.
RETN	R		Continue text scan.
TCMPLX	R		Test if the imaginary part of a COMPLEX variable can be addressed using the same base as the real part.
TREQ	R		Analyze the DED of the temporary operand and perform any necessary rounding of the offset to maintain alignment.
DEDC	R		If a DED/FED has been encountered, relocate the offset by the relocation factor passed by Phase PA.
DEDK7	S		Relocate DED offsets in a KONST(07) text table.
TEMALL	R		Allocate storage for all temporary operands.
FNDT	R		Search the temporary storage stack for a temporary operand or the next available space for one.
ARGTMP	R		Entered if a temporary operand or a Q-temp. requires a string locator.
QTMP	S		Generate a text table to move a skeleton locator from STATIC to temporary storage.
NQT	S		Generate a text table to initialize address in locator.
EOT	R		After text has been scanned, write out end-of-program marker, spill the output page, discard the input page, complete the second file, and dump it.
OPTAB	T		Table of operand code bytes for address processing.
ADDTXT	T		Addressing table.
XTXPT	CSECT		)
XRFAB	CSECT		)
XBRIC	CSECT		) XROUT
XBREAK	CSECT		)
XDIREC	CSECT		)
XSTG	CSECT		Private storage for Phase PI.

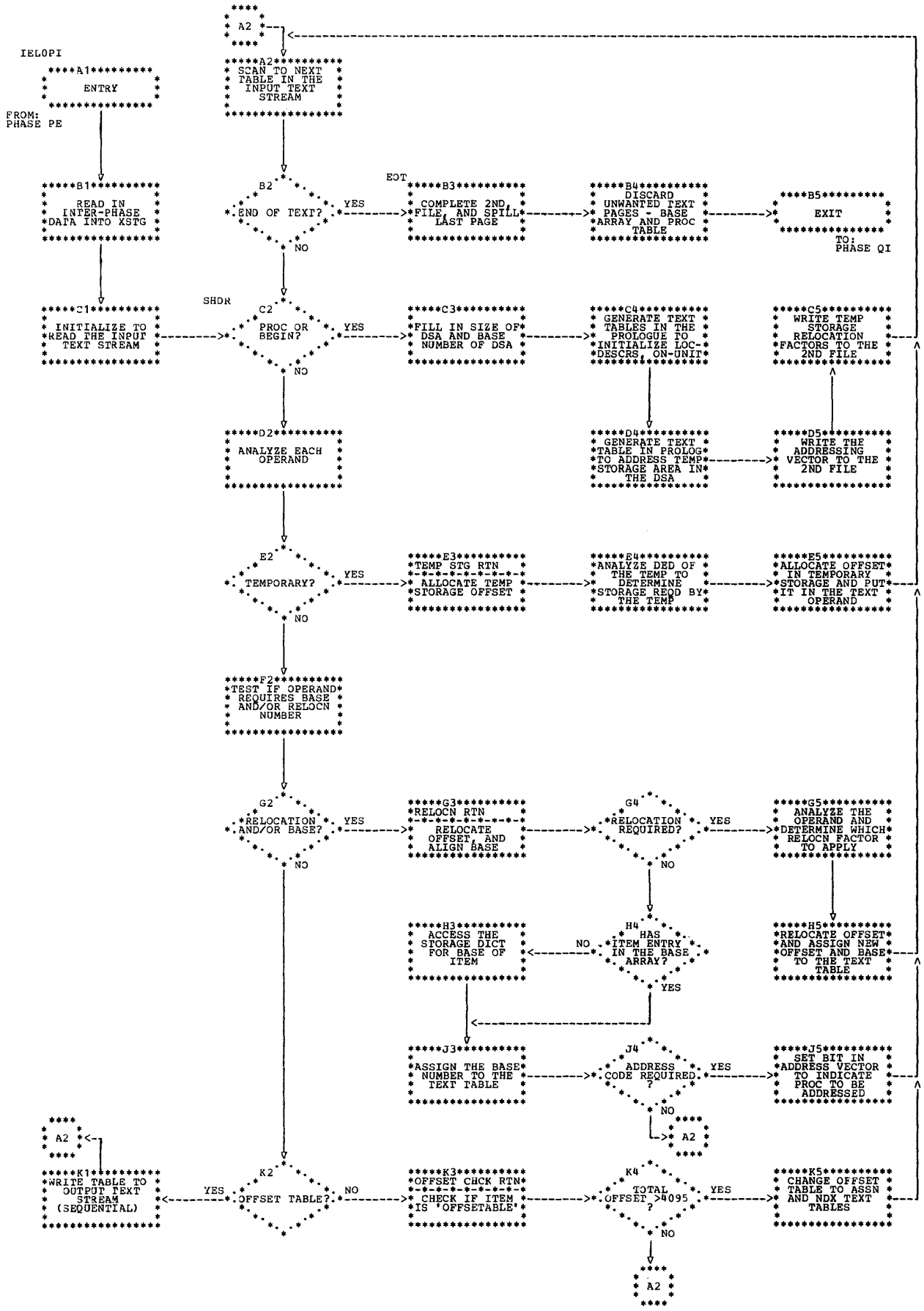


Chart 3.39. Addressing of Storage Phase (Phase PI)

Name	Type	Base registers	Function
IEL0QI	CSECT	R4-R7, R9	Entry point to Phase QI.
IEL0QI	R		Initialize phase and scan of input text pages.
LAB1	R		Examine text table and branch to appropriate routine.
FLWZ	R		Examine and process flow unit information.
PROCZ	R		Copy the addressing and temporary storage information for each block seen in the sequential text scan into the phase working storage, for ease of access.
LLADZ	R		Process LLAD text table.
VDAZ	R		Process VDA text table.
LAZ	R		Process LA text table.
BCZ	R		Process BC text table.
GKBC	R		Carries out further processing of branch tables (BC, GOTO, etc.).
OFFSZ	R		Process OFFS and NDX text tables.
CONVZ	R		Process CONV text table.
BADDRZ	R		Address optimization.
MASSNZ	R		Process MASSN text table.
ASSNZ	R		Process ASSN text table.
DIVZ	R		Process DIVIDE text table.
MULTZ	R		Process MULT text table.
PLUSZ	R		Process PLUS text table.
MINUSZ	R		Process MINUS text table.
DINCZ	R		Process DINC text table.
SCIZ	R		Process SCI text table.
GOTOZ	R		Process GOTO text tables.
GOOBZ	R		Process GOOB text table.
CALLZ	R		Process CALL text table.
ENTRYZ	R		Process ENTRY text tables.
MOVEZ	R		Process MOVE text table.
PENDZ	R		Process PEND text table.
FTCHZ	R		Process FETCH text tables.
ONCADZ	R		Process ONCAD text table.
GSLZ	R		Process GSL text tables.
ARGZ	R		Process ARG text table.
KONSTZ	R		Process KONST text table.
SNZ	R		Process SN and SL text tables.
ACUMZ	R		Process ACCUM text tables.
PLONK	R		Process text tables which merely require possible relocation of temporary operands.
ENDPROGZ	R		Carry out final housekeeping and pass control to Phase QA.
STOPUM	R		Stop on invalid code byte.
RELOC	S		Relocate the offset of a temporary operand, and recalculate its base number. Check for global temporary operands set in optimized inner loops and set bit in vector if so.
RDIR	S		Examine the current text table for items requiring relocation. For each one found, call subroutine RELOC.
ACMCHK	S		Check for occurrences of the accumulator in a loop and replace them by the global temp. used to accumulate the result.
INAC	S		Insert addressing code to address outer blocks.
BADCHK	S		Stack BADDR text tables and output them when the tables requiring addressing are reached. Move parameter addressing out of loops. Common BADDR 05 text tables in non-optimized case. Saves the bases of based LCVs for future storage.
EXCHK	S		Check if an operand of a text table is floating and of extended precision.
LFCHK	S		Check MOVE and CONV text tables for setting of loop

			control variable.
STHED	S		Process SN and SL text tables.
BLCVA	S		Process head of loop with based loop control variable.
BLCVB	S		Recalculate temporary storage size at the end of the block when storage for based LCVs is known.
BLCVC	S		Process end of loop with based LCV.
OUTSR2	S		Output a text table.
OUTSR2K	R		Output a KONST(09) text table.
RRGEN	S		When, in a PLUS or MULT text table, operand 1 = operand 2, generate assignment of operand 1 to a temporary operand.
ABTST	S		Check operand for abnormality.
XINIT	CSECT		)
XTXPG	CSECT		)
XRFAB	CSECT		)
XBREAK	CSECT		) XROUT
XBRIC1	CSECT		)
XDIREC	CSECT		)
XSTG	CSECT	RA	Private storage for Phase QI.

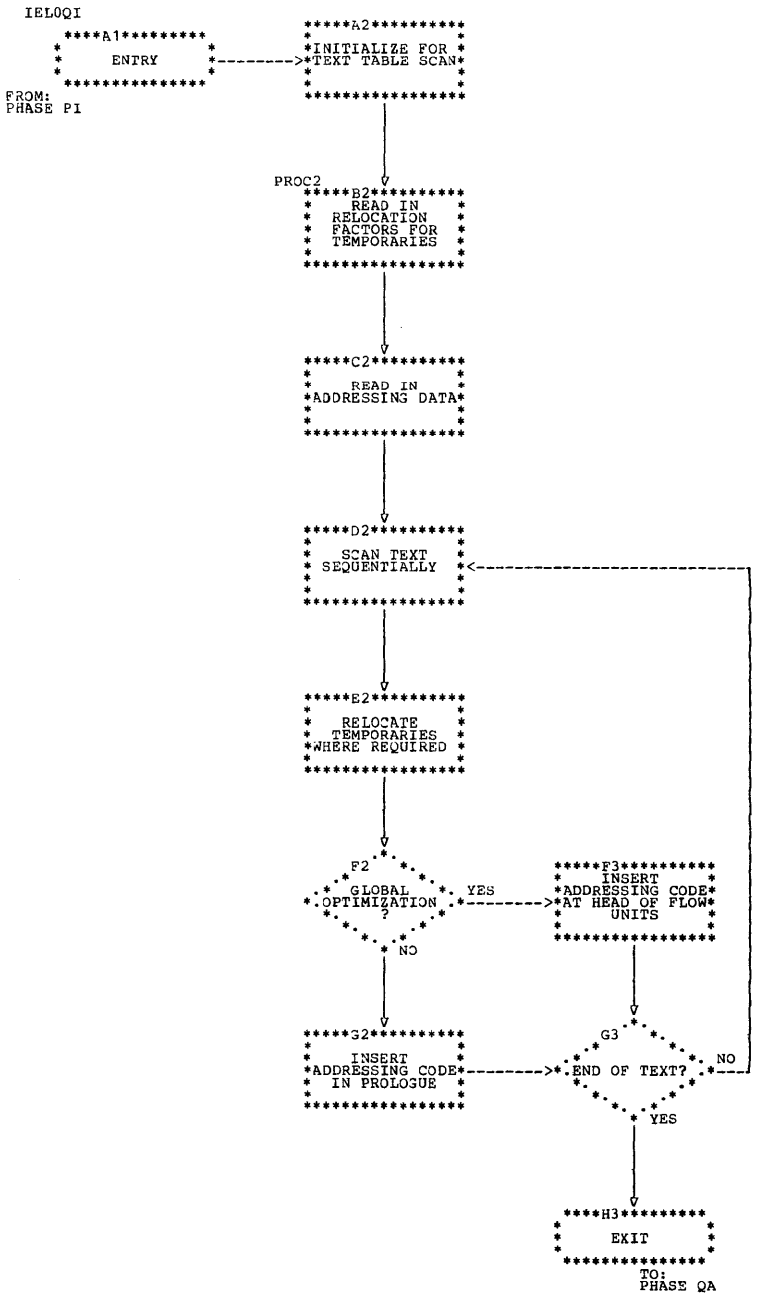


Chart 3.40. Optimized Addressing (Phase QI)

Register Allocation (Phase QA)

Name	Type	Base registers	Function
IEL0QA	CSECT		Entry point to Phase QA
QA	R		Initialize registers, storage, etc.
GRETA	R		Get a suitable register usage table element.
RRES	R		Determine whether an item is register-resident.
FBAS	R		Determine whether a base is register-resident.
SCAN1	R		Determine whether operand 1 of a text table is register-resident.
SCAN2	R		Determine whether operand 2 of a text table is register-resident.
SCAN3	R		Detemine whether operand 3 of a text table is register resident
TEST3	R		Clear operand 3 of a text table from a register, if necessary.
FRF1	R		Find a work register for operand 1.
FRF2	R		Find a work register for operand 2.
FRF3	R		Find a work register for operand 3.
FBF1	R		Find a base register for operand 1.
FBF2	R		Find a base register for operand 2.
FBF3	R		Find a base register for operand 3.
HOLD1	R		Determine whether operand 1 is to continue to be register-resident.
HOLD2	R		Determine whether operand 2 is to continue to be register-resident.
HOLD3	R		Determine whether operand 3 is to continue to be register-resident.
FORIT	R		Reserve a specified register.
WONIN	R		Find an even/odd register pair when one item in an operation is register-resident.
FPAR	R		Find an even/odd register pair.
SHIFTY	R		Handle an operand 1 shifting operation.
SWIT12	R		Switch operands 1 and 2.
QSR	R		Process a Q-temp.
GTA	R		Acquire temporary storage for a register.
FFR	R		Find a free register.
FTO	R		Calculate the offset of a register-resident item which has been relegated to main storage-resident.
XINIT	CSECT	R9	)
XBREAK	CSECT	RF	)
XTXPG	CSECT	R9	) XROUT
XBRIC	CSECT	R9	)
XSTG	CSECT	RA	Private storage for Phase QA.

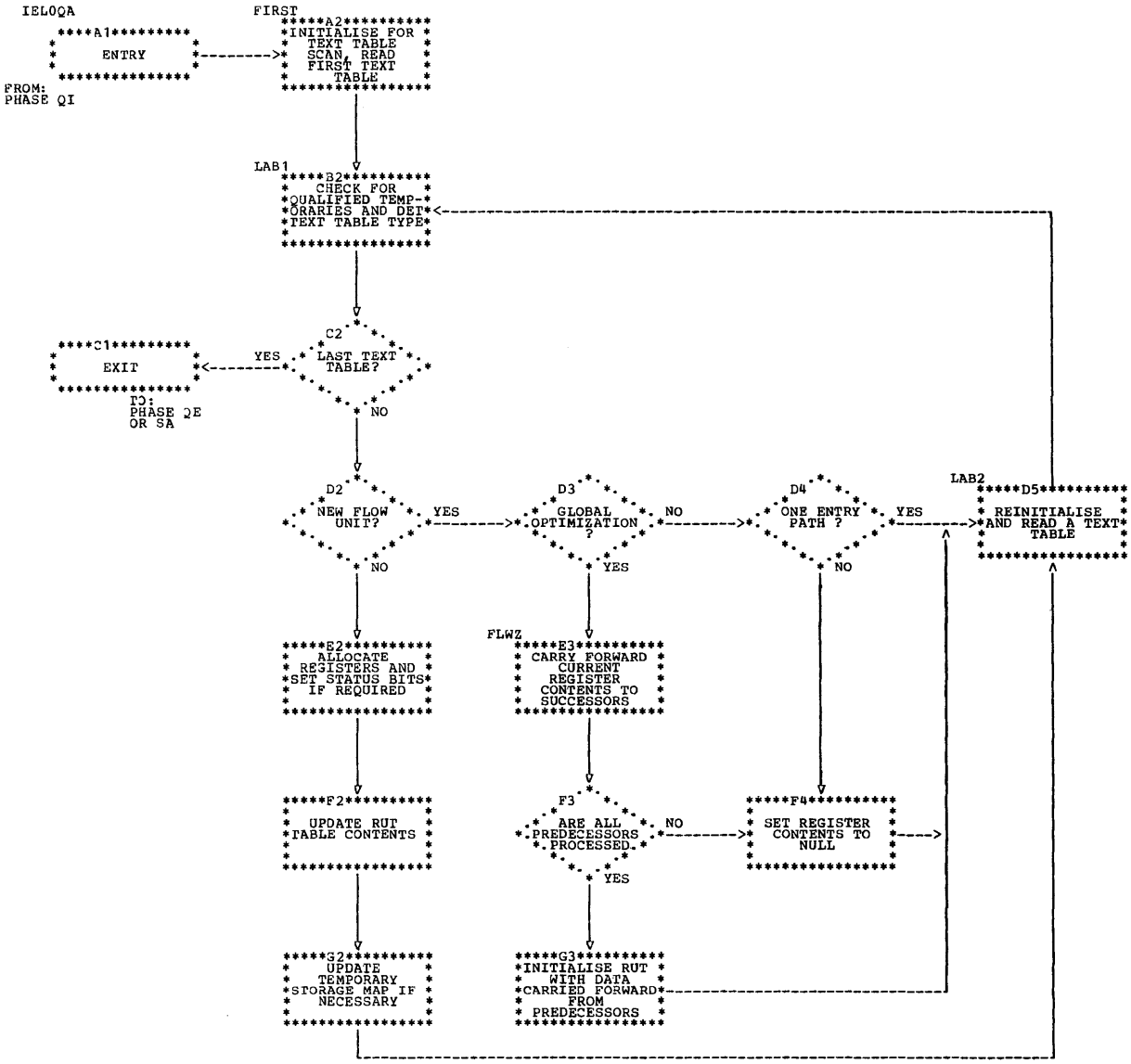


Chart 3.41. Register Allocation Phase (Phase QA)

Elimination of Unnecessary Storage Operations (Phase QE)

Name	Type	Base registers	Function
IEL0QE	CSECT	R4-R7	Entry point for Phase QE.
HERE	R		Set up base registers.
FIRST	R		Phase initialization.
LAB2	R		Output the text table after it has been processed.
LAB3	R		Input the next text table.
LAB1	R		Examine the text table and branch to the appropriate processing routine.
FLWZ	R		Process flow-unit header. Only the first flow-unit header of each block is passed from Phase QA.
PROCZ	R		Process block header table. If this block has been optimized, optimization bit vectors follow the block header. These are read in.
ASSNZ	R		Process ASSN text tables.
ITDOZ	R		Process ITDO text tables.
ACUMZ	R		Process ACCUM text tables.
KONSTZ	R		Process KONST text tables.
PLONK	R		Process text tables which merely require testing for temporary operands.
ENDPROGZ	R		Carry out final housekeeping and pass control to Phase SA.
STOPUM	R		Stop on invalid input code byte.
RDIR	S		Check operand 3 of each text table to see if it is a global register temporary operand. If so, one of two vectors is inspected. 1. If it is the accumulator global temporary, the accumulator bit vector is inspected, using the accumulator number (picked up from the ACCUM text table) as an index. If the bit is on, the accumulator was lost from its register and a store is required. 2. If it is any other global temporary, the storage offset (divided by 4) is used as an index to the global bit vector. If the bit is on, the temporary was lost from its register and a store is required.
VECTOR			Addressing vector. Elements contain offsets of start of code for each text table.
GBVEC			Phase QE examines the following 256-bit bit vectors output after each block header table by Phase QA. These vectors are stored in the phase storage area (XSTG) of Phase QE. This shows which global temporary operands were never loaded and hence need never be stored. Phase QE switches the store flag off in tables which have them as results. If the bit in the vector is on, the temporary requires storage and is loaded.
LCVEC			This shows which of the KONST(0C) text tables which appear at the head of inner do-loops should be changed to stores of the loop control variable. A store will be necessary if the LCV is addressed in the loop or is busy-on-exit at some branch out of the loop (except for simple cases which are optimized by Phase QA). For this determination two other vectors are also examined. These are the vectors on abnormal information, which have been obtained from the general dictionary by Phase QI and passed to Phase QE in the text stream. The KONST(0C) table must be changed to a store if the LCV is used in an I/O on-unit (test vector ABIONU) or if it is used in a computational on-unit (test vector ABCONU) with order specified. If the bit in LCVEC is on, it indicates that the KONST(0C) table must be changed to a store.

Continued on next page

BLCVEC			This shows which of the KONST(10) text tables are to be changed to stores. The KONST(10) table is a dummy store of a base of a based LCV or of the base of a parameter at the head of a loop out of which Phase QI has moved the parameter addressing code. A store is necessary if the base is lost from its register before the end of the loop. If the bit in the vector is on, a store is required.
ACMVEC			This shows which of the global temporary operands used as accumulators were never loaded and hence need never be stored. Phase QE switches the store flag off in text tables which have them as results. If the bit in the vector is on, a store is required.
ABIONU			Abnormal information bit vector (see LCVEC above). If the bit in the vector is on it indicates that the LCV is used in an I/O on-unit.
ABCONU			Abnormal information bit vector (see LCVEC above). If the bit in the vector is on it indicates that the LCV is used in a computational on-unit.
XINIT	CSECT		)
XTXPG	CSECT		)
XBREAK	CSECT		) XROUT
XBRIC1	CSECT		)
XMESGR	CSECT		)
XSTG	CSECT	RA	Private storage for Phase QE.

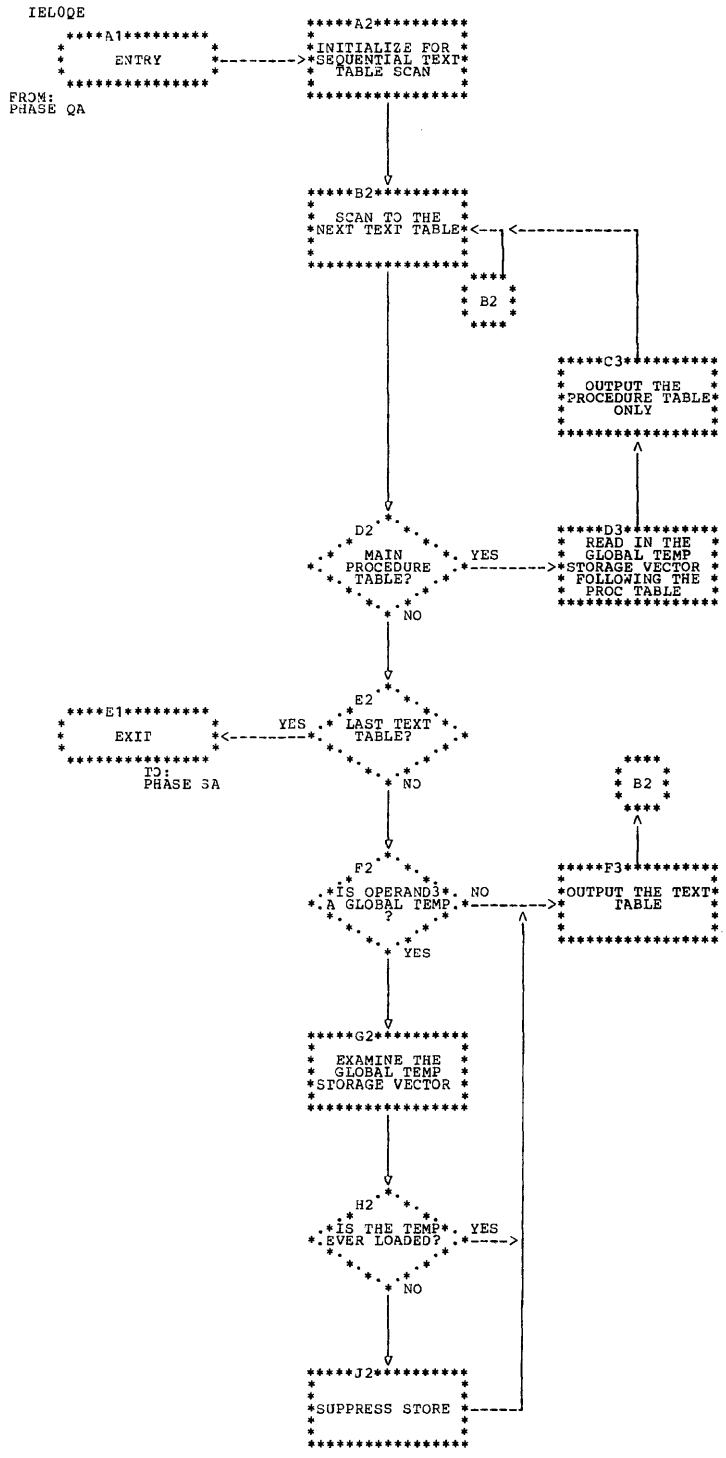


Chart 3.42. Elimination of Unnecessary Store Operations (Phase OE)

Code Generation (Phases SA, SQ, SD, and SC)

Note: Each of the code generation phases consists of two modules, a root module and a non-root module. The root modules of all four phases are identical in organization. The non-root modules are similar, the major difference being that each phase processes only certain types of text table, thus the code skeleton blocks and directories are different for each phase. The following organization table outlines the organization of root and non-root modules, showing, in the case of the non-root module, the minor differences between phases, and listing those text tables either processed or partly processed by each phase.

Name	Type	Base registers	Function
SART	CSECT	R6-R8	Entry point to the root module of the phase. This CSECT is identical for all five phases.
CONCON	S		Generate a constant in a form suitable for output.
NLIST	S		Process a sequence of code which will not be followed by listing information even when the CODE option has been specified.
MKOUT	S		Output a text table marker.
SCALMV	S		Align to a fullword boundary then output code.
SCMV	S		Output routine.
DECLGTH	R		Calculate the length of a decimal operand in bytes.
STARTUP	R		Initialize scan of input pages.
INITIO	R		Examine first text table in the input stream.
MAIN1	S		Select bit strips.
LOAD1	S		Set base loads in bit strip if necessary.
MAIN4	S		Build register and operand arrays to facilitate further processing.
MAIN5	S		Select skeleton instructions for processing.
RO3	S		Use the skeleton op code to index translate table.
TRTABLE	T		Changes op code in preparation for instruction processing routine selection.
RRR	S		RR instruction processing routine.
PSEUDO	S		Handle pseudo markers in text tables.
PSCOPY	S		Copy a number of bytes from text table to output buffer.
PSMARK	S		Move a marker from a code skeleton to output buffer.
PSTTLAB	S		Deal with internal text table labels.
PSTTBR	S		Deal with branches within text tables.
PSTTBXLE	S		Process markers for BXLEs within text tables.
RXX	R		RX instruction processing routine.
RX4BIT	S		Process the 4 bits which refer to the operand.
RX2	S		Entered when the operand field is a register operand.
RX0	S		Change the operator if the operand has greater than default precision.
BRRX	R		RX branch instruction processing routine.
BAL1	S		Fill register fields for RX branch.
LARX	R		LA instruction processing routine.
BRRS	R		RS branch instruction processing routine.
RXD	S		Deal with label references.
LMSTM	R		LM/STM instruction processing routine.
RSSH	R		SHIFT instruction processing routine.
SS1L	S		SS instruction processing routine (one length).
SS2L	S		SS instruction processing routine (two lengths).
SI	R		SI instruction processing routine.
BSDP	R		Calculate a base and offset.
LISTRX	R		Append listing information to object code.

Continued on next page

XTXPG	CSECT	R9	)
XRFAB	CSECT	R9	)
XBRIC1	CSECT	R9	)
XBREAK	CSECT	RF	) XROUT
XINIT	CSECT	R9	)
XDIREC	CSECT	R9	)
XSTG	CSECT	RA	Working storage for the phase
SASEG	CSECT	R7-R8	Entry point to the non-root module of the phase.
SASEG	R		Initialization routine.
SCSCHB	R		Scan input text stream.
SCSCHB1	S		Test the type of the found text table against the directories to determine whether it can be processed by this phase.
BRI	R		Entered if the text table can be processed.
SCSRCH1	S		If the input pointer has found a marker, process this marker and continue input scan.
SCSRCH2	S		If the input pointer has found a text table which cannot be processed by this phase, put out special marker and move text table to output.
SCSRCH3	S		If the text table was preceded by special case code, output the contents of the output buffer and return to routine SCSCHB to continue input scan.
SCSRCH5	S		Entered when the input pointer points to code which must be transferred directly to the output.
NEXTPZ	R		Pass control to the next phase.
LEVEL1	T		First-level directory (256 bytes), used to determine whether a table with a given value of IOP1 (see figure 5.96) is to be processed by the phase, and if so, how many second-level directory entries exist for that value of IOP1.
LEVEL2	T		Second-level directory, containing one entry for each different type of text table, as defined by IOP1/IOP2, to be processed by the phase. Each entry contains the address of the code skeleton block used in generating code for this type of table.
	T		One code skeleton block for each type of table processed by the phase. Each block contains: • a code skeleton array containing a list of skeleton instructions, • bit-strip arrays used in selecting instructions from the code skeleton array, • special case coding for the table. The text tables either processed or partly processed by each phase are as follows: Phase SA: PROC CHANE CGSR ENTRY CHECKC GEN BEGIN NOCHECK GOTO ONB SN GOOB CALL SL ITDO ONS GSN BADDR RETURN GSL OFFS PEND VDA

Continued on next page

			Phase SQ: ASSN MULT BC PLUS DIVIDE LADDR PPLUS SHIFT OFFS MINUS DINC KONST PMINUS SCI REVERT Phase SD: LADDR ONCAD ALIST OFFS ALOBIF KONST CONV ARG Phase SC: COMP MASSN MAP BCB AND MOVE RESDES OR TRT OFFS NOT KONST
CONREG	S		Set base registers for operands 4, 5, and 6 in OPTAB.
STNOCH	R		(Phase SA only) Output text table markers.
OFFCD	R		Evaluate coded offset modifiers.
OFFCD	CSECT	R9	CSECT for routine OFFCD.
LENCD	R		(Phases SD, and SC only.) Evaluate coded length modifiers.
LENCD	CSECT	R9	CSECT for routine LENCD.
SHCD	R		Process coded shifts from skeletons.
SHCD	CSECT	R9	CSECT for routine SHCD.
IMCD	R		(Not Phase SQ.) Compute an immediate field.
XBRIC1	CSECT	R9	)
XBREAK	CSECT	R9	)
XTXPG	CSECT	R9	)
XRFAB	CSECT	R9	) XROUT
XDIREC	CSECT	R9	)
XRFSEQ	CSECT	R9	)
XSTG	DSECT	RA	Private working storage for module.

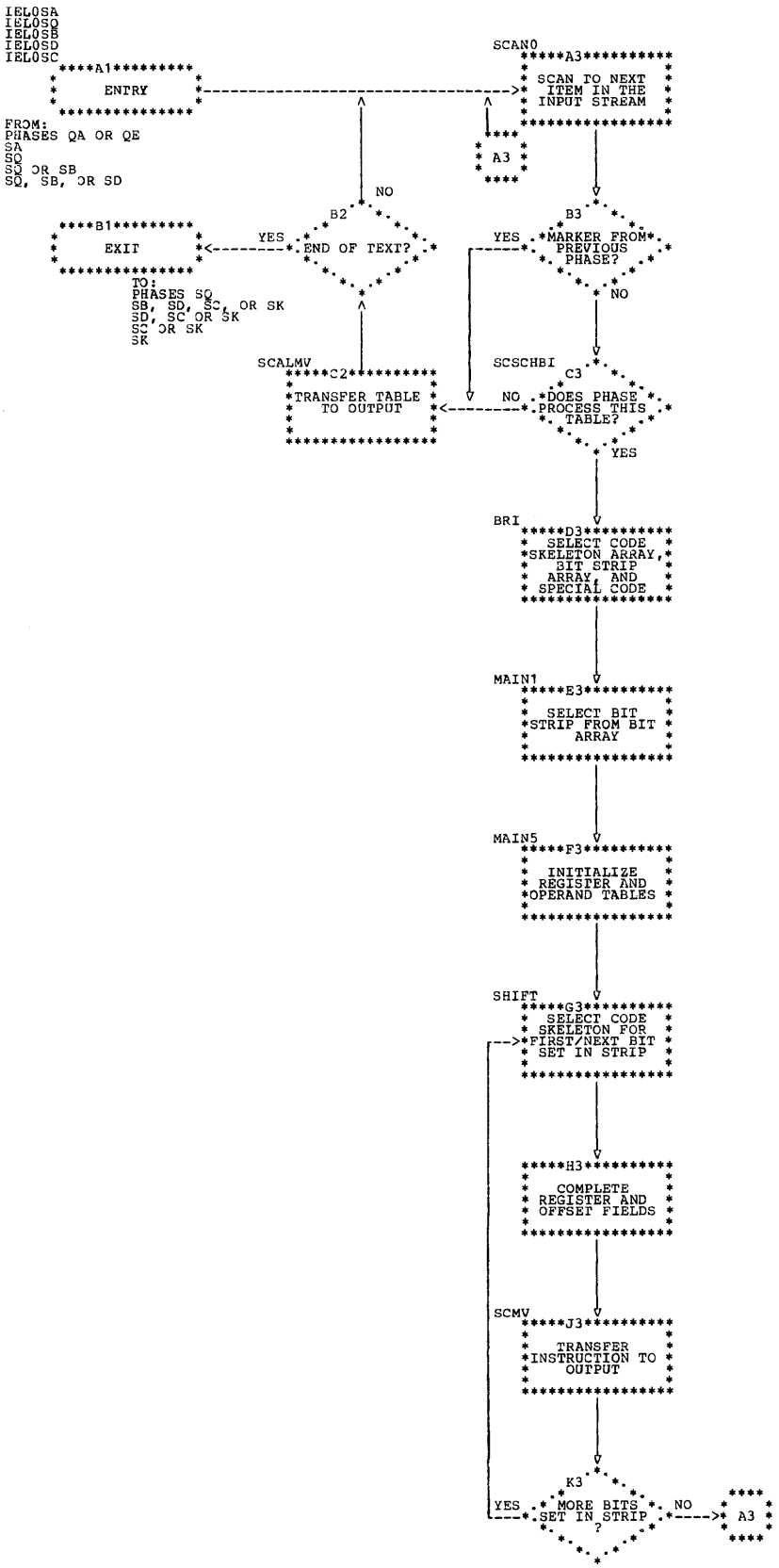


Chart 3.43. Code Generation - Passes 1, 2, 3, and 4 (Phases SA, SQ, SD, and SC)

Name	Type	Base registers	Function
ZTEXTL	DSECT	R1	Describes the input stream.
LABELS	DSECT	R3	Describes label table entries.
REGTAB	DSECT	R5	Describes register-usage-table entries.
IEL0SK	CSECT	R6-R8	Entry point to Phase SK.
IEL0SK	R		Initialize registers, storage, text page handling, label table handling, etc.
MSCAN	R		Examine next item in input; branch to one of the following routines or, if no special action required, skip to next item and update code count.
BR1	R		Remove NOPRs if not required.
LA1	R		Remove LA R,0(0,R). Change LA R,0(0,R') to LR R,R' .
SLA1	R		Change SLA R,1 to AR R,R .
MVC1	R		Merge contiguous MVCs if possible.
XC1	R		Change XC FIELD(1),FIELD to MVI FIELD,0 .
RXD	R		Deal with RX instructions where the operand is not aligned on the correct boundary. Such an operand is moved to/from correctly aligned work space.
SCAN1	R		First scan of the input text stream, examining marker bytes. Divides the code into addressable regions. Branches to one of the following subroutines, depending on the marker examined: NEXT TTLABP LABP TTABLEP UGOP LADP CGOP TTSTMTP PROCP LABVARP BASEP BUMP4 FINP ALIGNMP STP LOOPMKP PROLBP KALLP KALLVP GOBACKP
SCAN2	R		Second scan of the input text stream. Offsets of labels from the origin of their containing region are calculated as the code is scanned and are put into the label table. Because the regions are fixed, exact space can be calculated. Branches to one of the following subroutines, depending on the marker examined: NEXT LADP LABO RLDDO UGOP LABVARO CGOP ENTO PROCO BUMP4 FINO MOVEDO STOO BLENDO TTLABO SCGSO ECGSO
BREAK	S		Called with RB pointing at an instruction or marker. Updates the register-usage table to reflect the new state, then deletes, output, and/or bumps over the instruction or marker, and increments the code count as applicable.

Continued on next page

LABPAGE	S		Called with RB pointing at a label number. Calculate which page this label lies in, check whether the page is in main storage, and if not, fetch it in.
DIAGX	S		Note that an entry will be generated in the diagnostic statement number table. If the offset exceeds 32K, the dictionary reference diagnostic statement counters are updated.
FLOW	S		Generate the code to provide execution flow trace.
OPTAB	T		Interpretation of System/360 op. codes.
MKTAB	T		Interpretation of marker identifier bytes.
XINIT	CSECT		)
XTXPG	CSECT		)
XBRIC1	CSECT		)
XBRIC3	CSECT		) XROUT
XBREAK	CSECT		)
XRFAB	CSECT		)
XMESGR	CSECT		)
XDIREC	CSECT		)
XSTG	CSECT		Private storage for Phase SK.

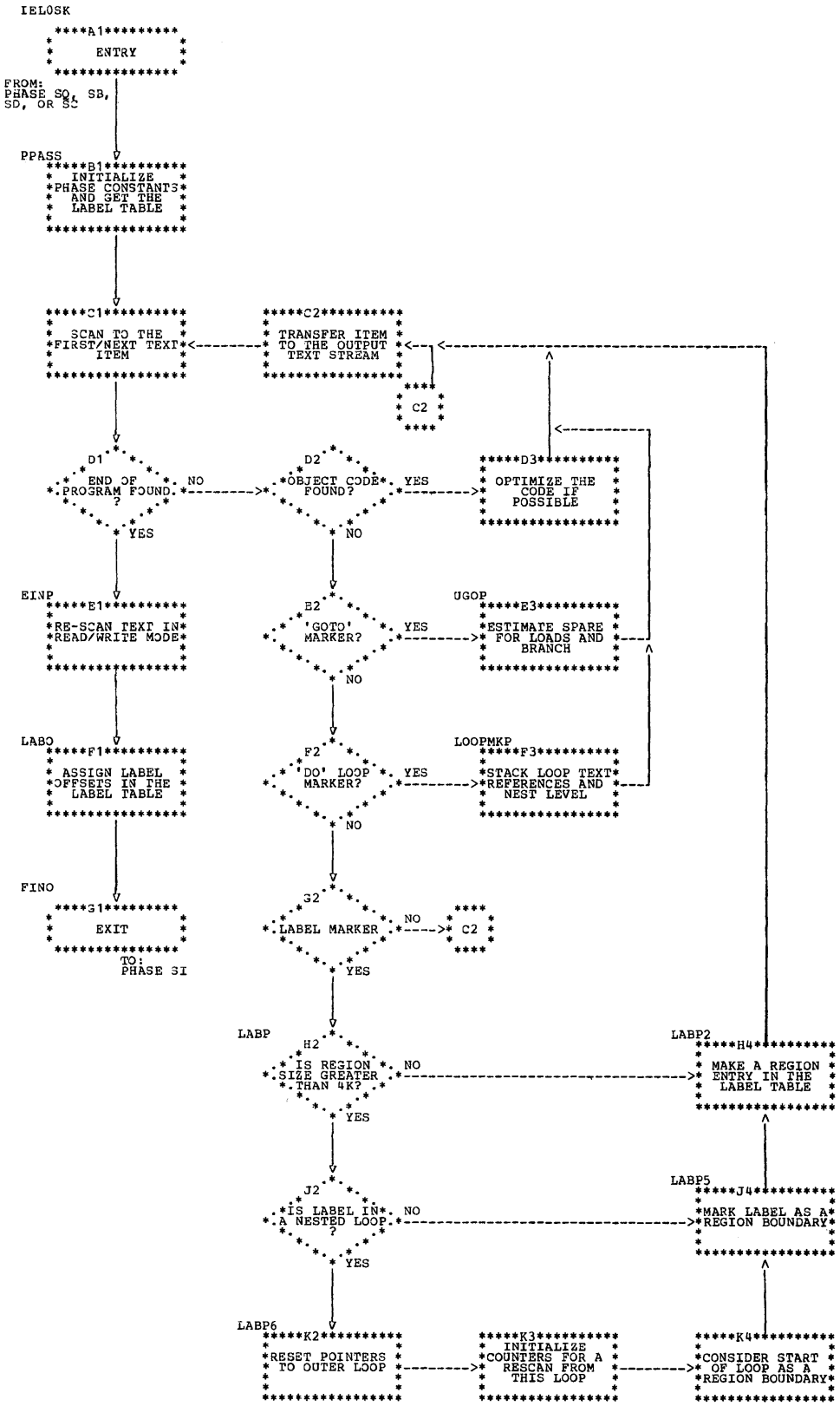


Chart 3.44. Label Resolution Phase (Phase SK)

Final Assembly (Phase SI)

Name	Type	Base registers	Function
IEL0SI	CSECT	R6,R7, R8,R9	Entry point to Phase SI.
SI	R		Initialize registers, storage, etc.
INITT	R		Initialize TXT and RLD cards.
SCAN1	R		Scan the input text stream.
OBJOUT	R		Add code to TXT records.
REC	R		Output a card/card image.
EMPTY	R		Complete not fully used TXT/RLD records.
PSOUT	R		Add code to TXT records if code requires replication.
RLD0	R		Make an RLD entry.
RLDOUT	R		Add an RLD entry to an RLD card.
ESDEXTN	R		Create an external 7-character name from a user identifier.
SIADOUT	R		Make an entry in the third output text stream.
DIAGX	R		Determine the need for an entry in the diagnostic statement table, and calculate its space requirement.
LABPAGE	R		Get a required label page and index to the required label.
ESDSD	R		Add information to an ESD entry.
ESPOUT	R		Make an ESD entry in a card/card image.
PSPL	R		Index to routines which process the constants pool.
XINIT	CSECT	R9	)
XRFAB	CSECT	R9	) XROUT
XTXPG	CSECT	R9	)
XBRIC3	CSECT	R9	)
XSTG	CSECT	RA	Private storage for Phase SI.

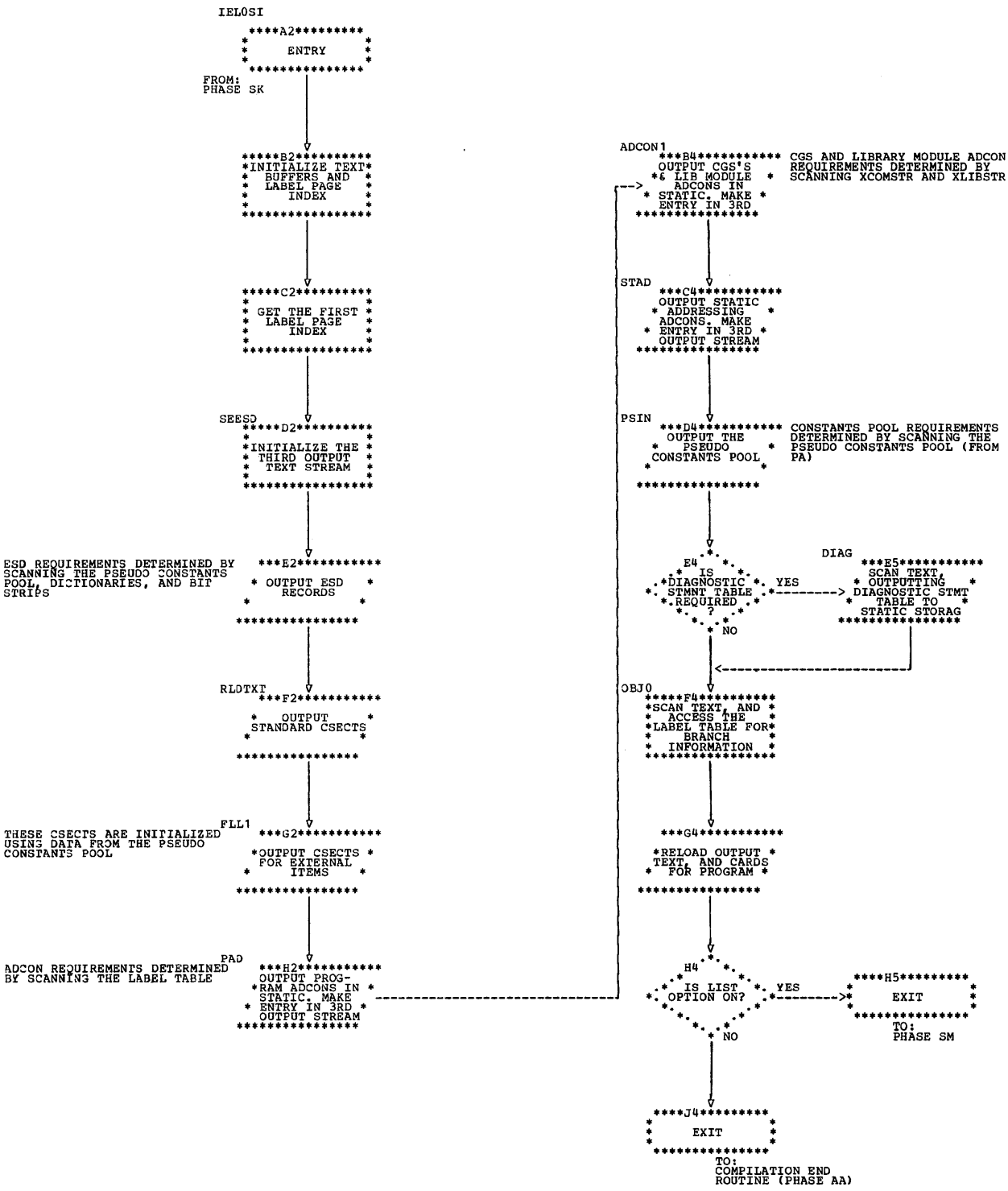


Chart 3.45. Final Assembly Phase (Phase SI)

Listings (Phase SM)

Name	Type	Base registers	Function
IEL0SM	CSECT		Entry point to Phase SM.
SM00	R		Initialize registers, storage, input/output, etc.
ALT00	R		Generate an aggregate length table.
SR00	R		List the storage requirements.
ESD00	R		List the ESD entries.
SL0	R		List the static storage.
SLE0	R		List the static external entries.
OFF00	R		List statement offsets.
SCAN	R		Scan the input text stream.
MARKERS	R		Process markers.
RRINST	R		Process an RR instruction.
RXINST	R		Process an RX instruction.
RSINST	R		Process an RS instruction
SIINST	R		Process an SI instruction.
SSINST	R		Process an SS instruction.
NEXTPZ	R		Exit to the next phase.
SCA	S		Scan over a marker.
SCAN1	S		Scan over an RR instruction.
SCAN2	S		Scan over an RX instruction.
SCAN3	S		Scan over an RS instruction.
SCAN4	S		Scan over an SI instruction.
SCAN5	S		Scan over an SS instruction.
SPACE	S		Insert a blank in the print buffer.
COUNTER	S		Print the location counter.
COMM	S		Insert a comma in the print buffer.
ONE	S		Insert a byte in the print buffer.
HALF1	S		Insert a half-byte in the print buffer.
HALF2	E		Entry point to subroutine HALF1.
OE	S		Skip over a null byte.
OPCODE	S		Insert an operation code in the print buffer.
PAD	S		Insert a decimal field in the print buffer.
REGISTER	S		Insert a register value in the print buffer.
PRINTIT	S		Output the buffer contents to the print file.
CHECK	S		Check the print buffer pointer position.
PUTITIN	S		Move a symbolic field into the print buffer.
EXPAND	S		Expand a pseudo-constant into the print buffer.
BUMP1	S		Increment the print buffer pointer.
BUMP2	E		Entry point to subroutine BUMP1.
XINIT	CSECT	R9	)
XRFAB	CSECT	R9	)
XDIREC	CSECT	R9	) XROUT
XPRNT	CSECT	R9	)
XTXPG	CSECT	R9	)
XBRICI	CSECT	R9	)
XSTG	CSECT	RA	Private storage for Phase SM.

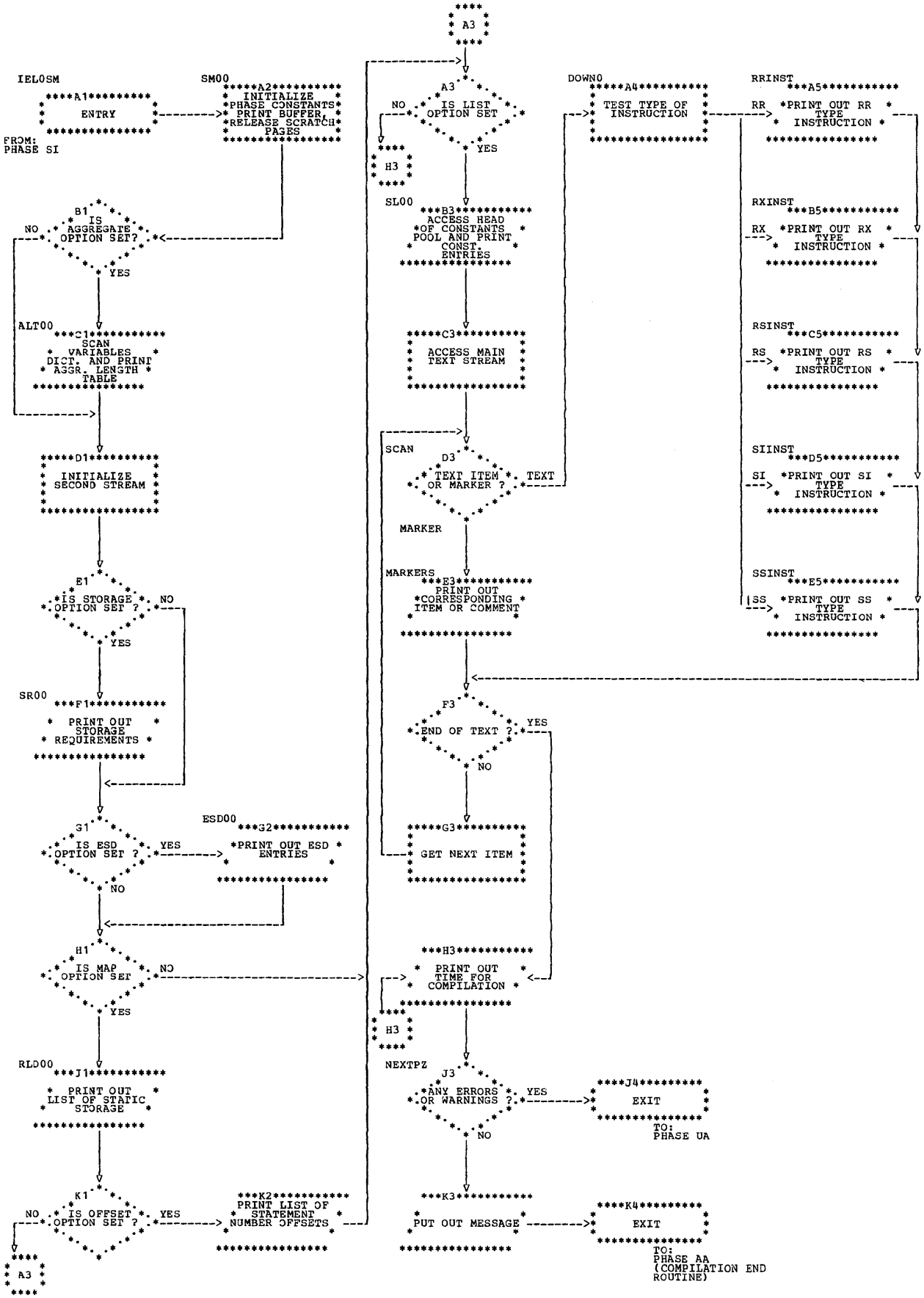


Chart 3.46. Object-code Listing Phase (Phase SM)

Diagnostic-Message Editor (Phase UA)

Name	Type	Base registers	Function
IEL0UA	CSECT	R9	Entry point to root module of Phase UA.
IEL0UAS	CSECT	R9,R7&R6	Entry point to sort module of Phase UA.
UF1	R		Test FLAG option.
U1	R		Build sort units, i.e., generate a page stream of message entries from the input message stream for sorting purposes.
SU1	DSECT	R4	)
SU2	DSECT	R5	) Describe sort units.
SCHELL	R		Sort routine to sort all sort units in one page into order of severity, line number, and message number.
U97	R		This routine takes the sorted pages and merges them into a single text stream.
IEL0UAC	CSECT	R9,R7&R6	Entry to point to print module of Phase UA.
UEDLST	R		Search edit table for a particular message number entry. If found, check that statement number in table entry is lowest encountered so far. If there is no table entry for that message, make one.
U55	R		Initialize print dataset. Print error-message page headings.
U124	R		Test FLAG option.
U38	R		Start of message printing loop. Determine the message number of the next message to be printed.
MED	DSECT	R1	Describe entries in the table of edited messages
UR3	R		Skip over a special purpose edited-message sort unit.
U100	R		Determine whether an edited message is to be printed, and if so, obtain the start of the list of special purpose edited-message sort units.
U99	R		Print the message severity subheading before the first message of a new severity level, and the message introduction information (i.e., message number, severity code, and statement number or numbers, if edited).
U90	R		Examine the message table (MESTAB) to determine whether a message has been provided for the number in the error message entry. If not, print a diagnostic.
U35	R		Message interpreting routine. Examine each code of a coded message to determine whether it is 1. a keyword, 2. a level marker, 3. an end-of-message marker, 4. a text parameter, 5. a statement-number parameter, 6. a dictionary parameter, 7. a quote marker, 8. an alternative text marker. If it is a level marker, multiply the level number by 256 to add to the keyword code following and to construct the correct keyword code with which to reference the keyword table (KEYTAB).
U32	R		Obtain a keyword from KEYTAB, given the keyword code.
U26	R		Handle 'no blank' characters by concatenating words in a sub-buffer. Concatenate punctuation characters with keywords, if necessary.
U28	R		Scan to the next message, or invoke the control phase clean-up routine if the end of the message stream is reached.
U27	R		Convert a statement number to character form, when found in the middle of the message.
U29	R		Access a names dictionary entry when a dictionary reference parameter is encountered in a message.

U30	R		Determine a text parameter's type, and place appropriate text in the message stream.
UBUF	S		Has three entry points as follows:
UBUF	E		Normal entry point. Text pointed to by RB, the length of which is in RC, is moved into the print buffer, and the line length, ULINE, updated by the amount moved in. If the line is full, it is printed, and the next buffer initialized. ULINE is incremented by one more than the length of the text moved into the buffer, to leave a space between each item on the line.
UBUF1	E		As for UBUF (E), except that the previous buffer is printed and a new line established for the new text item. A space is not left after this item, because the first item moved into the buffer is always the message identification letters.
UBUF3	E		As for UBUF (E), except that no allowance is made for space after the item.
ZTRAN1	T		Internal code-to-EBCDIC translation table.
XINIT	CSECT	)	
XDISC	CSECT	)	
XDIREC	CSECT	)	
XPRNTN	CSECT	) XROUT	
XBREAK	CSECT	)	
XTXPG	CSECT	)	
XBRIC2	CSEC	)	
XPRNT	CSECT	)	
XSTG	CSECT		Private storage for Phase UA.
IEL0UAT	CSECT		Second module, containing message tables.
MCDE	T		Table of codes giving, for each message number; 1. severity of message, 2. editing requirements for message, 3. presence or absence of line number parameter in message.
MESTAB	T		Table of coded messages.
MESREF	T		Table relating messages number to coded message in MESTAB.
KEYTAB	T		Table of keywords used in messages.
KEYREF	T		Table relating length of a keyword to its position in KEYTAB.
IEL0UAK	CSECT		

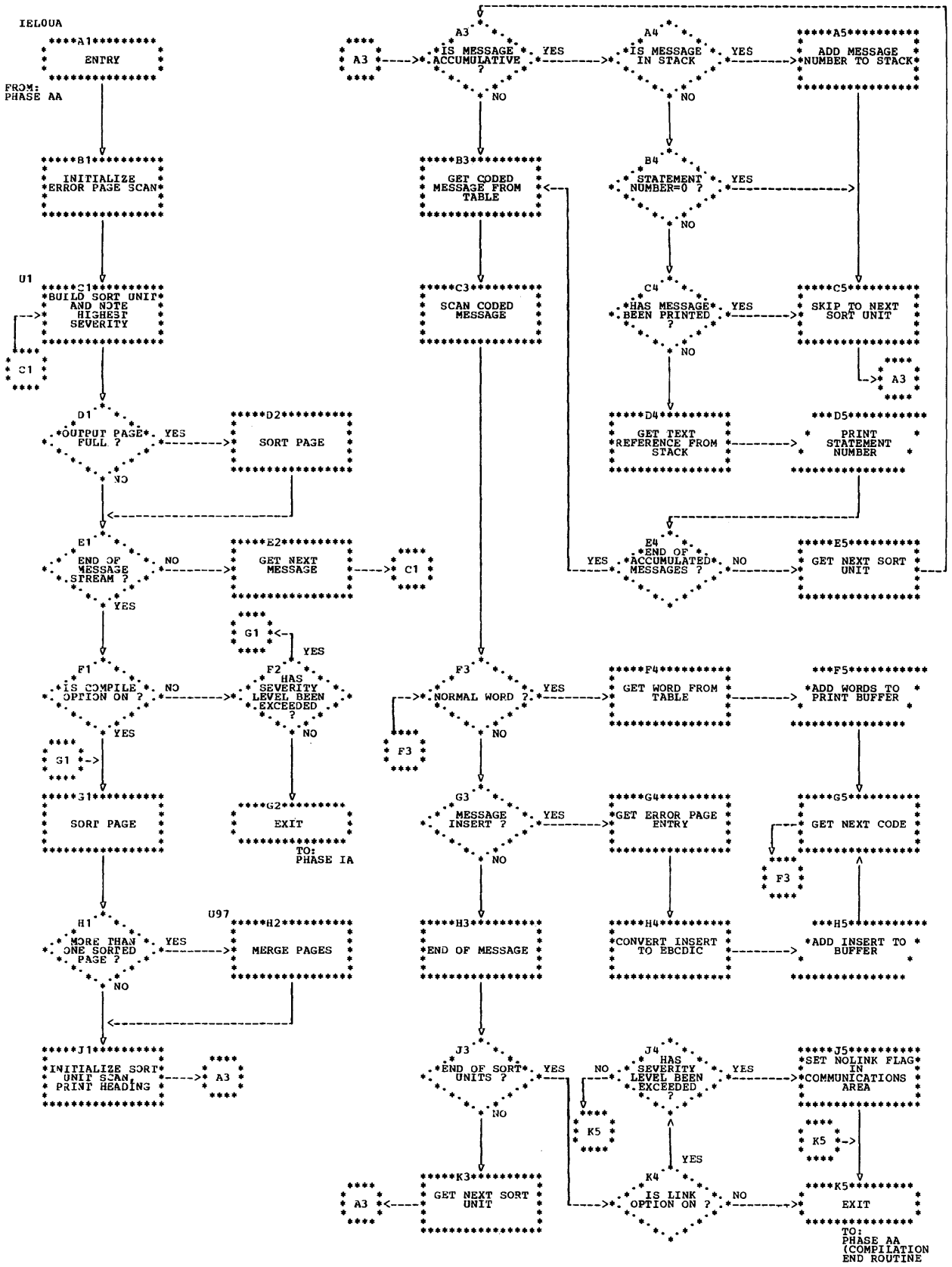


Chart 3.47. Diagnostic-message Editing Phase (Phase UA)

Dump Phase (Phase AI)

Name	Type	Base registers	Function
IEL0AI	CSECT	R8,R6, R9	Entry point to Phase AI.
DUMP	R		Initialize registers, storage, files, and parameters.
PGHDTS	R		Dump page header and chain information, if required.
REGTST	R		Dump registers, if required.
COMRGD	F		Dump the communications region (XCOMM), if required.
XSTGDP	R		Dump the variable storage area (XSTG), if required.
RUTDMP	R		Dump the register usage table, if required.
TRCTDP	R		Dump the flow trace table.
EXTDMP	R		Dump the execution trace table.
DICDMP	R		Dump the dictionaries.
FORMDC	R		Produce a formatted dictionary dump.
GENFMT	R		Routine used by FORMDC to dump a general dict. entry.
VRSFMT	R		Routine used by FORMDC to dump a variables dict. entry.
UPSLTS	R		Update dictionary type text byte and continue dump.
DENTDP	R		Dump a single dictionary entry.
TXDUMP	R		Initialize code bytes for dumping routines.
NOTXFM	R		Dump current output pages.
WHLTXT	R		Dump current input page.
NOCURT	R		Dump the current output text stream.
ERPDUMP	R		Dump the error text page.
DMPENT	R		Print range dump, if required, and exit from the phase.
PRSQTP	S		Produce a formatted dump of a text page, or group of pages.
FLHDP	S		Dump a flow-unit header.
SQLTXT	S		Process non-chained text.
RDINSC	S		Scan Type 1 text produced by the syntax analysis stage.
RDINPR	S		Print Type 1 text produced by the syntax analysis stage.
DICTSC	S		Scan Type 1 text at dictionary build time.
PRTSTR	S		Construct formatting strings for use by the TXHERE routine, and print text if necessary.
TXTBSC	S		Scan past a text table.
TXTBPR	S		Print a text table.
NOSQTP	S		Scan along a text table chain, printing the tables in formatted style.
PRTNSQ	S		Output a text table reference to a buffer, and prepare to output the table itself.
OFTPRT	S		Output to a buffer the offset of a text entry from the start of the page.
PRTSBD	S		Set up a subheading for text entry fields.
TXMRTN	S		Set up and print a message giving a page's text reference.
DMPTTL	S		Print the buffer pointed at by R1.
FMTBUF	S		Formatting subroutine.
TXHERE	E		Entry point for text.
DPTRCB	S		Print the contents of TRCBUF on the next line.
TRL40	S		Translate 40 hex bytes to character form.
DPTRCS	S		Skip two lines and print the contents of TRCBUF.
DMPCOR	S		Produce a hex dump of main storage area R1 to R2.
DMPTR	S		Dump a line of X'20' bytes.
TRLSIX	S		Translate three bytes into six characters.
PRT325	S		Print a number of 32-byte strings.
TRLBIT	S		Translate a fullword into a string of character bits.
	T		Miscellaneous constants, messages, and tables.
TXBYTES	CSECT		Table for converting text code bytes into symbolic form.

Continued on next page

XINIT	CSECT	R9	)	
XRFAB	CSECT	R9	)	
XPRNT	CSECT	R9	)	XROUT
XNEXT	CSECT	R9	)	
XTXPG	CSECT	R9	)	
XSTG	CSECT	RA	Private storage for Phase AI.	

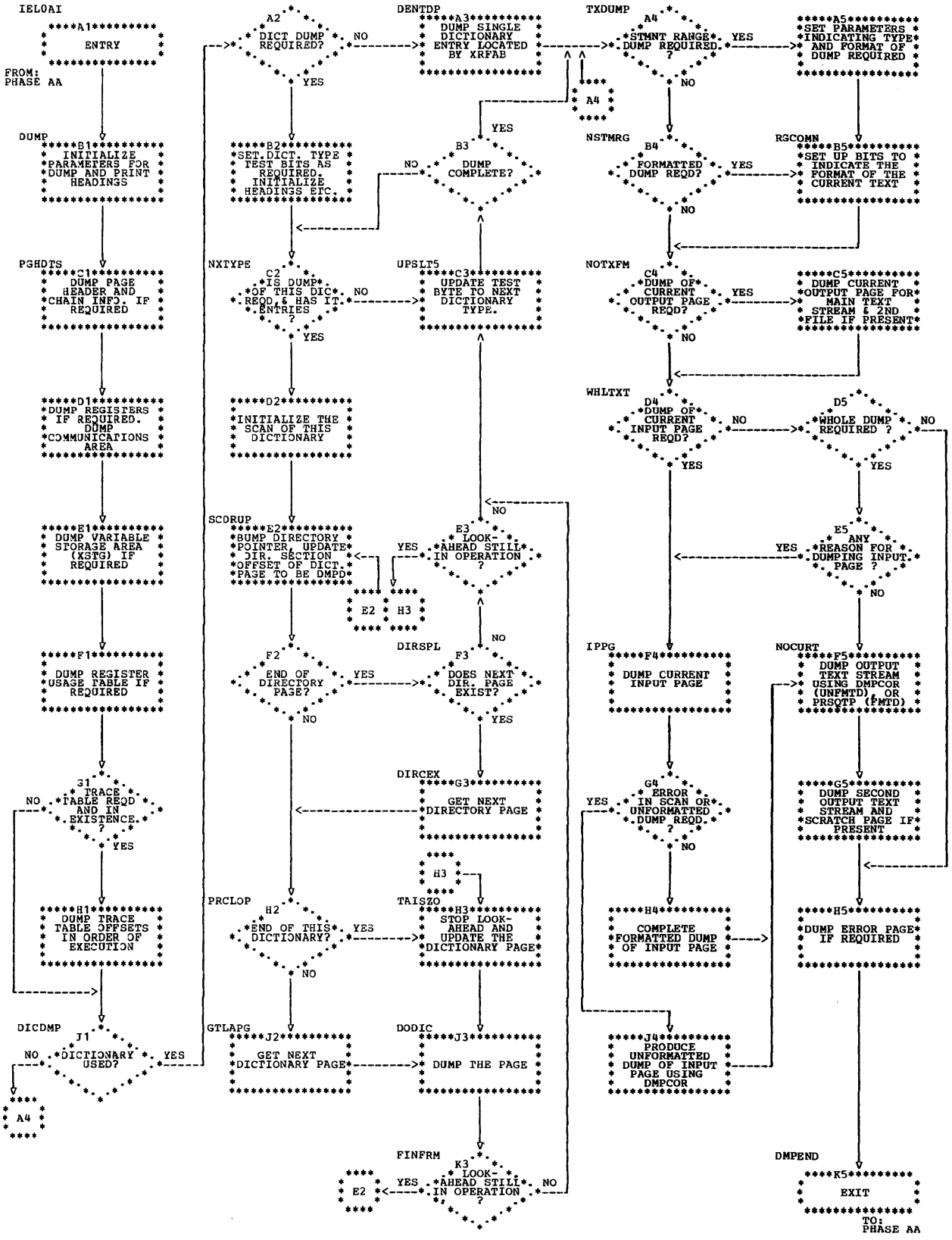


Chart 3.48. Dump Phase (Phase AI)

INTRODUCTION

This section is a directory to the phases of the compiler that perform particular processing functions involved in compilation. It is based on features of the PL/I language that can be included in source programs submitted for processing by the compiler. To enable the directory to be used for different purposes, the information it contains is presented in two differently-organized lists.

The first list is arranged for use when the processing of a particular language feature, (e.g., statement type, problem-data type, problem-data attribute, program-control-data type, etc.), is to be examined. The various language features that can appear in a PL/I program are listed in alphabetic order. Beside each language feature, the phases that can be involved in its processing are listed in alphabetic order, together with a brief description of the processing performed.

The second list is arranged for use when the processing performed by a particular phase is to be examined. The phase identifiers, (i.e. the last two characers of each phase name), are listed in alphabetic order. Beside each phase identifier, the language features processed by the phase are listed in alphabetic order, together with a brief description of the processing performed.

LANGUAGE FEATURE	FEATURE QUALIFICATIONS	COMPILER PHASE	COMPILER PROCESSING FUNCTIONS
All language		BA	Convert 48-char source to 60-char, and BCD to EBCDIC (MACRO not specified).
		CA	Print listing if INSOURCE specified. Convert 48-char source to 60-char, and BCD to EBCDIC (MACRO specified).
		II	Translate from Type 1 to Type 2 text.
		IK	On specification of the XREF and/or ATTRIBUTES compiler options, prints cross-references and attributes of all identifiers.
		QA	Assign absolute register numbers where required.
		QE	Allocate storage for global temporaries. Remove unnecessary stores of global temporaries.
		QI	Relocate temporary storage. Insert addressing code in prologue or optimal flow unit, if OPT=TIME. Diagnose compiler error conditions. Reserve loop control register for binary loops.
		SM	Diagnose compiler error conditions.
Arithmetic op.		EA	Operator/operand syntax analysis.
		EE	Syntax analysis: operator/operand.
		EI	Check for syntax errors local to the statement. Syntax analysis: operator/operand.
		II	Analyze expressions.
		KV	Optimize multiplication and exponentiation.
		KX	Set flags for code generation.
		OM	Commoning and moving of expressions.
		PA	Generate constants used in arithmetic operations.
		SQ	Generate object code (binary, float, and ext. float ops.).

LANGUAGE FEATURE	FEATURE QUALIFICATIONS	COMPILER PHASE	COMPILER PROCESSING FUNCTIONS
Array x-section		EI	Check for syntax errors local to the statement.
		ID	Create reduced descriptors in arguments, LIST I/O, and WAIT. Insert text for filling in reduced descriptors.
		IE	Check for errors. Syntax expansion: generation of do-loop control temporaries in assignments and stream I/O expressions.
		IQ	Map arrays.
		KE	Expand array cross-section text.
Assignment statement (aggregate)	Array/struc, multiple, BY NAME	IE	Check for errors in non-scalar assignments. Syntax expansion: non-scalar assignments. Expand character and bit string aggregate assignments.
		IQ	Map aggregates.
		KA	Insert statements into statement-type chains.
		KE	Expand array and structure assignments, and compiler-generated do-loops.
		KX	Set flags for code generation.
		OC	Expand STRING assignments.
		OI	Analyze connection information.
Assignment statement (element)	Data types, multiple	GM	Check for errors on left hand side.
		IE	Check for errors in scalar assignments (when the statement contains scalar-returning functions which in turn have embedded aggregate assignments). Syntax expansion: scalar assignments.
		II	Check for mismatch between source and target, and for illegal target. Complete dict. entries for targets in chameleon temp. assignments. Translate Type 1 text assignments to ASSN and CONV tables.
		KX	Set flags for code generation.

Compiler processing functions listed by language feature

LANGUAGE FEATURE	FEATURE QUALIFICATIONS	COMPILER PHASE	COMPILER PROCESSING FUNCTIONS
Assignment statement (element)	Data types, multiple	OA	Extract alias information concerning locator, label, and entry assignments.
		OC	Expand STRING assignments.
		OM	Commoning and moving of expressions.
ALIGNED/ UNALIGNED		GA	Check for conflicting attributes. Set a bit in the dictionary entry.
		II	Argument matching/generic selection.
		IQ	Map ALIGNED/UNALIGNED BIT strings in aggregates.
		SK	sign unaligned operands of RX instructions.
ALLOCATE stmt.	CONTROLLED/ BASED,SET,IN	EA	Syntax analysis: page splitting.
		EE	Set up a dictionary text file ALLOCATE chain for each block. Check attributes of allocated variables, and check SET/IN options.
		GE	Check allocate attributes against declaration attributes. Merge allocation and declaration information. Create allocate-time aggregate table dictionary entries. Generate extent and INITIAL assignments.
		IA	Reposition/generate bound assignments to aggregate descriptors. Generate mapping operators for allocated variables. Syntax expansion: extent expressicn assignments. Convert offset locators to pointers.
		IE	Check for illegal use of aggregates in attribute specification.
		II	Translate Type 1 text extent expressions, MAP statements, and ALLOCATE statements to Type 2 text tables.
		IQ	Allocate storage for BASED and CONTROLLED aggregates.
		KA	Process and error-check resulting ARRAY INITIAL and STATIC INITIAL assignments.

LANGUAGE FEATURE	FEATURE QUALIFICATIONS	COMPILER PHASE	COMPILER PROCESSING FUNCTIONS
ALLOCATE stmt.	CONTROLLED/ BASED,SET,IN	PA	Allocate storage for controlled anchor slot.
		PI	Addressing of CONTROLLED variables. Create locator-descriptors for CONTROLLED variables.
AREA		EE	Syntax analysis: AREA arguments.
		GA	Check for conflicting attributes. Set a bit in the dictionary entry.
AUTOMATIC stge.		PA	Allocate storage and initialize static areas. Construct area locator/descriptor if required.
		GA	Check for conflicting attributes. Set a bit in the dictionary entry.
		IQ	Map AUTOMATIC aggregates.
		PE	Allocate storage for AUTOMATIC variables.
		PI	Addressing of AUTOMATIC variables.
Base	DECIMAL/BINARY	EE	Syntax analysis: arguments.
		GA	Check for implementation limit violations.
		II	Check for illegal conversions. Analyze expressions prior to generation of CONV etc., tables. Generate Type 2 text tables from expressions.
		KK	Expand arithmetic and mathematical bifs.
		OM	Examine precision, scale, etc., in commoned expressions.
Built-in		GA	Create dictionary entries for explicitly-declared bifs.
		GI	Create dictionary entries for contextually declared bifs.
		GM	Create dictionary entries for implicitly declared bifs.
		IA	Check for illegal use of bifs as locator qualifiers. Syntax expansion: special cases.

Compiler processing functions listed by language feature

LANGUAGE FEATURE	FEATURE QUALIFICATIONS	COMPILER PHASE	COMPILER PROCESSING FUNCTIONS
Built-in		ID	Check for errors in aggregate-manipulation BIFs and pseudovariables. Create temporaries or reduced descriptors.
		IE	Syntax expansion: aggregate bifs and pseudovariables.
		II	Argument checking. Generate BIF tables and CONV tables for arguments.
		KA	Process STRING bifs and check for errors.
		KK	Check the validity of arguments to certain bifs. Expand all non-string bifs.
		KV	Expand CHAR, BIT, UNSPEC, SUBSTR bifs.
		OC	Expand TRANSLATE and BOOL bifs.
		OX	Optimize REPEAT, LENGTH, INDEX, VERIFY, HIGH, LOW, and STRING bifs.
BASED storage	REFER	GA	Check for conflicting attributes. Set a bit in the dictionary entry.
		GE	Detects errors in REFER declaration. Puts REFER information in aggregate descriptors.
		IA	Generate RESDES operators to reserve space for special temporary locator-descriptors. Examine references to BASED REFER, etc., and decide whether or not mapping is necessary. If so, generate code to set up descriptors and MAP operators. Syntax expansion: map-on-reference cases and unqualified BASED variables.
		IE	Check for errors. Syntax expansion: generation of temporaries.
		IQ	Map BASED aggregates.
		PE	Associate a storage base number with each BASED variable to assist in BASED-variable addressing.
		PI	Addressing of BASED variables.

LANGUAGE FEATURE	FEATURE QUALIFICATIONS	COMPILER PHASE	COMPILER PROCESSING FUNCTIONS
BEGIN stmt.	Options	EE	Set up a dictionary text file entry chain for each block, and a block-header chain. Syntax analysis: options and arguments.
		GA	Check the validity of options on BEGIN statements. Create a block-header dictionary entry for the block.
		OA	Create a block optimization entry.
		OE	Complete the block optimization entry.
		OI	Analyze connection information.
		SA	Generate object code.
Compile-time (%) stmts.		CA	Remove all % symbols except from %PAGE, %SKIP, %CONTROL, %PRINT, and %NOPRINT statements. Execute all compile-time (%) statements.
		EA	Execute %PAGE, %SKIP, %CONTROL, %PRINT, and %NOPRINT statements if SOURCE specified.
Constant		EA	Syntax analysis: format constants.
		EE	Syntax analysis: format constants.
		EI	Check for syntax errors local to the statement. Syntax analysis: format constants.
		GM	Create dictionary entries for constants.
		II	Check for illegal usage, and convert arguments to temps. Generate Type 2 text.
		OM	Optimize the production and usage of literal constants.
		PA	Allocate storage for constants.
Conversion		II	Check for illegal conversions. Generate CONV Type 2 text tables.
		KX	Expand text for inline conversions.

Compiler processing functions listed by language feature

LANGUAGE FEATURE	FEATURE QUALIFICATIONS	COMPILER PHASE	COMPILER PROCESSING FUNCTIONS
Conversion		OC	Expand text for inline conversions.
		OM	Commoning and moving of expressions.
		PA	Compile-time conversion of constants.
		SD	Generate object code.
CALL stmt.	Argument list, function ref., options.	GI	Create dictionary entries from contextual declarations for bifs, and TASK or EVENT variables.
		ID	Create aggregate temporaries, including reduced descriptors.
		IE	Check for errors in CALL statements. Syntax expansion: CALL statements.
		II	Check for errors in argument matching. Translate CALL statements from Type 1 to Type 2 text, generating ALIST, ARG, CALL, FNCT, etc., text tables.
		IM	Create dictionary entries for temps. created for arguments needing to be remapped before calls to FORTRAN/COBOL. Insert library calls into text to set up correct environments before and after CALL.
		KA	Check for errors (match number of arguments against number of parameters).
		KT	Produce calling sequence.
		KV	Unnest calls.
		OA	Extract calling information for blocks.
		OE	Analyze flow information.
		OI	Analyze connection information.
		PA	Common argument lists. Allocate storage for argument lists.
		SA	Generate object code.
CHECK condition	Name list	KT	Process CHECK statements, and insert the appropriate library calls for check variables in the text.
		PC	Construct symbol tables for items in check lists.

LANGUAGE FEATURE	FEATURE QUALIFICATIONS	COMPILER PHASE	COMPILER PROCESSING FUNCTIONS
CHECK condition	Name list	SA	Generate object code (for CHECK condition without list).
		SQ	Generate object code (for CHECK condition with list).
CLOSE stmt.	ENV.	EE	Syntax analysis: options.
		KL	Generate an argument list and library module call for the statement.
		KM	Optimize by completing FCB and DTF at compile time if possible.
CONTROLLED storage		GA	Check for conflicting attributes. Set a bit in the dictionary entry.
		IQ	Map CONTROLLED aggregates.
		PA	Allocate storage for controlled anchor slot.
		PI	Addressing of CONTROLLED variables. Allocate storage for CONTROLLED variables.
Data-list item		PC	Check validity of based items in data directed I/O.
Dimension		EE	Syntax analysis: expressions and REFER items.
		GE	Create aggregate descriptor dictionary entries.
		ID	Check number of subscripts in subscript lists. Create aggregate temporaries and reduced descriptors in arguments, I/O, and WAIT. Flag statement header to indicate subscripted variable present.
		IE	Check for errors in expressions. Syntax expansion: expressions.
		II	Check for illegal usage in expressions.
		IM	Create dictionary entries for FORTRAN-mapped temp. arrays created for arguments or parameters in ILC. Remap temp. arrays for calls to or from FORTRAN.

Compiler processing functions listed by language feature

LANGUAGE FEATURE	FEATURE QUALIFICATIONS	COMPILER PHASE	COMPILER PROCESSING FUNCTIONS
Dimension		IM	Create assignments between FORTRAN-mapped temps. and original arrays.
		PA	Construct array locators and descriptors if required.
DECLARE stmt.		EA	Syntax analysis: page splitting.
		EE	Set up a dictionary text file DECLARE chain for each block. Syntax analysis: nesting and factoring.
		GA	Detect declaration errors. Create dictionary entries for explicitly declared names. Unfactor attributes.
		IE	Check for illegal use of aggregates in attribute specification.
		KA	Process and error-check resulting ARRAY INITIAL and STATIC INITIAL assignments.
		KL	Check FILE declaration for conflicting ENVIRONMENT options and attributes, and for correct MEDIUM specification. Create DTF, ENVB, and FCB entries in the general dictionary, and a names dictionary entry for the generated LIOCS module name required for the DTF.
		KM	Check file declaration, FCB, and ENVB for possible open and close optimization. Complete FCB and DTF if possible.
DEFAULT stmt.		EA	Syntax analysis: page splitting.
		EE	Set up a dictionary text file DEFAULT chain for each block. Syntax analysis: nesting and factoring.
		GA	Detect DEFAULT specification errors. Build the DEFAULT directory.
		GE	Use the DEFAULT directory.
		GI	Use the DEFAULT directory.
		GM	Use the DEFAULT directory.
		IE	Check for illegal use of aggregates in attribute specification.
		KK	Process and error-check resulting ARRAY INITIAL and STATIC INITIAL assignments.

LANGUAGE FEATURE	FEATURE QUALIFICATIONS	COMPILER PHASE	COMPILER PROCESSING FUNCTIONS
DEFINED	POSITION/iSUB	EE	Syntax analysis: bases.
		GA	Check for conflicting attributes. Set a bit in the dictionary entry.
		GE	Diagnose source errors. Generate addressing statements. Classify defining.
		IA	Generate ASSN tables for creation cf locator-descriptors. Insert MAP operators into the main text stream. Insert POS expressions and iSUB lists into the main text stream.
		ID	Flag statement header to indicate iSUB-defined item present.
		IE	Syntax expansion: iSUB-U statements containing functions with aggregate assignments.
		KE	Expand iSUB defining text.
		OA	Create value lists for defining bases.
DELAY stmt.		PA	Allocate addressing adcon or locator if required.
		EE	Syntax analysis: options.
		IE	Check for illegal use of aggregates.
		KT	Check that a DELAY expression is scalar. Expand the DELAY statement.
DISPLAY stmt.	REPLY,EVENT	EE	Syntax analysis: options.
		IE	Check for illegal use of aggregates.
		KT	Check the attributes of DISPLAY, REPLY, and EVENT expressions. Expand the DISPLAY statement.

Compiler processing functions listed by language feature

LANGUAGE FEATURE	FEATURE QUALIFICATIONS	COMPILER PHASE	COMPILER PROCESSING FUNCTIONS
DO stmt.	TO,BY,WHILE, UNTIL,REPEAT	EA	Analyze type compatibility and syntax of repetitive specifications.
		II	Check for illegal DO specifications. Translate Type 1 text into Type 2 text tables (DO1, DO2, etc.,).
		KA	Insert statements into statement-type chains. Complete the processing of the DOWHILE statement. Complete the processing of the DOUNTIL statement.
		KI	Process do-loops. Expand DO1, DO2, and DO3 into appropriate assignments and branch to produce codes.
		OE	Analyze flow information.
		OI	Analyze connection information.
		OM	Modify control variable usage.
Expression	Data type	EI	Check for syntax errors local to the statement.
		ID	Check the validity of aggregate expression arguments. Insert text-assigning aggregate expression arguments to temporaries.
		II	Check validity of expressions containing file, entry, label, etc.
END stmt.	Label	EA	Syntax analysis: block nesting.
		KA	Insert statements into statement-type chains.
		OE	Analyze flow information.
		OI	Analyze connection information.
		SA	Generate object code.
ENTRY attribute	Parameter attribute list ORDER/REORDER	EE	Syntax analysis: parameter attribute lists in ENTRY declarations.
		GA	Create parameter descriptor entries in the general dictionary.
		PE	Count the number of parameters in each block and pass this number to Phase PI.

LANGUAGE FEATURE	FEATURE QUALIFICATIONS	COMPILER PHASE	COMPILER PROCESSING FUNCTIONS
ENTRY stmt.	Multiple labs.	EE	Set up a dictionary text file entry chain for each block. Syntax analysis: options.
		GA	Create entry-point entries in the general dictionary.
		II	Translate from Type 1 to Type 2 text, and generate GSLs.
		IM	Create dictionary entries for new external procedure to be called by FORTRAN/COBOL. Insert library calls into text to set up correct environments before and after CALL. Create new external procedure and entry points, and calls to old procedure and entry points.
		KT	Produce appropriate branch code in prologue.
		OE	Analyze flow information.
		OI	Analyze connection information.
		SA	Generate object code.
EVENT attribute		KK	Expand EVENT bifs.
EXIT stmt.		KT	Expand the EXIT statement.
Format item		KQ	Create dictionary entries for FEDs.
		PC	Allocate storage for FEDs.
FILE	Const/var, STREAM/RECORD, KEYED,INPUT, OUTPUT,etc.	EE	Syntax analysis: record I/O arguments.
		EI	Check for syntax errors local to the statement. Syntax analysis: stream I/O arguments.
		GA	Check for conflicting attributes.

Compiler processing functions listed by language feature

LANGUAGE FEATURE	FEATURE QUALIFICATIONS	COMPILER PHASE	COMPILER PROCESSING FUNCTIONS
FILE	Const/var, STREAM/RECORD, KEYED, INPUT, OUTPUT, etc.	GA	Create file constant dictionary entries. Set a bit in the dictionary entry.
		GI	Create dictionary entries for contextually declared files.
		GM	Create dictionary entries for implicitly declared files.
		II	Check for illegal usage in expressions.
		IM	Create dictionary entries for COBOL-mapped temporary structures for I/O with COBOL files. Create assignments between COBOL-mapped temporary structures and original structures.
		KL	Check file declaration for conflicting ENVIRONMENT options and attributes, and for correct MEDIUM specification. Create DTF, ENVB, and FCB entries in the general dictionary, and a names dictionary entry for the generated LIOCS module name required for the DTF.
		KM	Check file declaration, FCB, and ENVB for possible optimization of open and close, and complete FCB and DTF where possible. Check record I/O files.
		PA	Allocate storage for file constants.
FORMAT stmt.		EI	Check for syntax errors local to the statement. Syntax analysis: arguments, repetitive specifications, and factored items.
		II	Generate FORME, FIT, etc., Type 2 text tables.
		KQ	Construct FEDs. Expand FMTLST/FORME/FIT/FITE text tables to GEN/LA/CAIL, etc.
		PC	Allocate storage for FEDs.
		SK	Remove unnecessary code. Detect statement too large.
FREE stmt.	CTL/BASED	EA	Syntax analysis: namelists.
		IA	Syntax expansion for implied AREAs.
		IQ	Free allocated storage.
		PI	Addressing of BASED variables.

LANGUAGE FEATURE	FEATURE QUALIFICATIONS	COMPILER PHASE	COMPILER PROCESSING FUNCTIONS
FUNCTION refs.		KA	Check for errors (match number of arguments against number of parameters).
GENERIC		EE	Syntax analysis: valid clauses.
		GE	Check for attribute and entry point errors. Create dictionary entries.
		II	Check for possibility of selection. Perform selection. Generate ALIST, ARG, CALL, etc., Type 2 text tables.
		KK	Process bifs passed as arguments.
GET stmt.	FILE/STRING, EDIT/LIST/DATA, COPY,SKIP,COL, data-list, format-list	EI	Check for syntax errors local to the statement. Syntax analysis: options, data lists, and DO-groups.
		KA	Insert statements into statement-type chains.
		KQ	Check for: errors in statement-type chains, invalid items in data lists, invalid options on statements. Match data and format lists. Create dictionary entries for FEDs. Allocate registers for special usage in stream I/O. Expand stream I/O statements to include library calls, CONV tables, etc.
		PC	Allocate storage for symbol tables built for GET DATA.
		SK	Detect statement too large.
GO TO stmt.	Label const/var	EA	Analyze target syntax.
		GM	Check for label constant or variable.
		IE	Check for illegal use of aggregates.
		KV	Optimize branching.
		OE	Analyze flow information.
		OI	Analyze connection information.

Compiler processing functions listed by language feature

LANGUAGE FEATURE	FEATURE QUALIFICATIONS	COMPILER PHASE	COMPILER PROCESSING FUNCTIONS
GO TO stmt.	Label const/var	SA	Generate object code.
IF stmt.	THEN,ELSE	PA	Allocate storage for label constant if go-to-outer block.
		EI	Syntax analysis: nesting and position.
		IE	Check for illegal use of aggregates.
		II	Check for illegal expressions in IF clauses. Translate Type-1 text IF statements into Type-2 text IF, GOTO, and GSL tables.
		KA	Complete the statement processing. Insert statements into statement-type chains. Representative text table processing, and statement analysis.
INITIAL	CALL/nocall	EE	Syntax analysis: INITIAL values, with replication and nesting.
		GA	Check for conflicting attributes. Set a bit in the dictionary entry.
		GE	Generate INITIAL assignment tables.
		IA	Generate multiple assignments and AID operators in the main text stream.
		KE	Expand automatic array initial text.
		PA	Allocate STATIC INITIAL storage.
Label	Constant	GI	Create dictionary entries for label constants.
		II	Check for illegal usage in expressions and assignments.
		OE	Analyze flow information.
		OI	Analyze connection information.
		PA	Allocate storage for label constants.
Locator	POINTER/AREA	GA	Check for conflicting attributes. Set a bit in the dictionary entry.
		IA	Check for illegal qualifier chains. Generate locator chains.

LANGUAGE FEATURE	FEATURE QUALIFICATIONS	COMPILER PHASE	COMPILER PROCESSING FUNCTIONS
Locator	POINTER/AREA	IE	Check for errors. Syntax expansion: generation of temporaries.
		II	Generate PTS, PTSAT, etc., Type-2 text tables.
		OA	Create value lists for locators.
LABEL	Variable	II	Check for illegal usage in expressions and assignments. Chaining.
		OA	Create value lists for LABEL variables.
		OE	Analyze flow information.
		OI	Analyze connection information.
		PE	Allocate storage for LABEL variables.
		SK	Remove if not referenced.
LIKE		GA	Check for conflicting attributes. Expand 'LIKE' specifications.
LOCATE stmt.	FILE, SET, KEYFROM	EE	Syntax analysis: options.
		IA	Create special locator-descriptors for REFER structures. Special aggregate-mapping action for REFER structures. Special syntax expansion action for REFER structures.
		IM	Create dictionary entries for COBOL-mapped temporary structures for I/O with COBOL files. Create assignments between COBOL-mapped temporary structures and original structures.
		IQ	Map aggregates as for BASED variables.
		KM	Generate library call, or inline code when possible.
Mode	REAL/COMPLEX	EE	Syntax analysis: arguments.
		EI	Check for syntax errors local to the statement.
		GA	Check for implementation limit violations.

Compiler processing functions listed by language feature

LANGUAGE FEATURE	FEATURE QUALIFICATIONS	COMPILER PHASE	COMPILER PROCESSING FUNCTIONS
Mode	REAL/COMPLEX	II	Check for illegal conversions. Analyze expressions prior to generation of CONV, etc., tables. Generate Type-2 text tables from expressions.
		KK	Expand arithmetic, mathematical, and COMPLEX bifs.
		OM	Examine precision, scale, etc., in commoned expressions.
		PA	Allocate storage for COMPLEX constants.
		PE	Allocate storage for COMPLEX variables.
		PI	Addressing of COMPLEX variables or COMPLEX temporaries.
Object code		SI	Produce ESD/TXT/RLD records for program, and all external CSECTs for input into linkage editor.
		SM	List the object code.
ON stmt.	On-unit/SYSTEM, SNAP	EA	Syntax analysis: block handling and condition argument lists.
		GI	Create dictionary entries for contextually declared conditions.
		KA	Insert statements into statement-type chains.
		OE	Analyze abnormal information (variables set and used in on-units).
		PA	Allocate storage for static ONCBs.
		PE	Allocate storage for dynamic CNCBs.
		PI	Addressing of on-units.
		SA	Generate object code.
ON-condition		EA	Syntax analysis: arguments.
		OE	Analyze variable usage information, and abnormal information.
		SA	Generate object code (unqualified ON-condition).
		SQ	Generate object code (qualified ON-condition).

LANGUAGE FEATURE	FEATURE QUALIFICATIONS	COMPILER PHASE	COMPILER PROCESSING FUNCTIONS
OPEN stmt.	ENV, options	EE	Syntax analysis: options.
		GM	Gather OPEN file attributes into a dictionary entry.
		IE	Check for illegal aggregates.
		KA	Insert statements into statement-type chains.
		KL	Check attributes with those declared. Create OCB (Open Control Block) and generate an argument list and library module call for the statement.
		KM	Optimize by completing FCB and DTF at compile time if possible.
PARAMETER storage		GA	Check for conflicting attributes. Set a bit in the dictionary entry.
		OA	Create value lists for parameters.
		PI	Reserve the appropriate storage for the parameter list for each block (number of parameters in each block is passed by Phase PE) in the temporary storage area based on R4.
Precision		EE	Syntax anlysis: arguments.
		KK	Expand arithmetic and mathematical bifs.
		OM	Examine precision, scale, etc., in commoned expressions.
Prefix		EA	Syntax analysis: checklists and prefix switches.
		II	Generate Type-2 text.
		SA	Generate object code.
Prologue text		IE	Scan-analysis and error checking in prologue text. Sorting of prologue text. Syntax expansion: prologue text (in functions with embedded aggregate assignments).
		QI	Insert addressing code in prologue or optimal flow unit, if OPT=TIME.
		SA	Generate object code.

Compiler processing functions listed by language feature

LANGUAGE FEATURE	FEATURE QUALIFICATIONS	COMPILER PHASE	COMPILER PROCESSING FUNCTIONS
Pseudovariable		IA	Detect illegal arguments. Syntax expansion: special cases.
		IE	Check for errors in aggregate bifs and pseudovariables. Syntax expansion: aggregate bifs and pseudovariables.
		KK	Check the validity of arguments to certain pseudovariables. Expand non-string pseudovariables.
		KV	Process UNSPEC and SUBSTR pseudovariables to produce correct object code.
		OX	Process STRING pseudovariable to produce library call, if necessary.
PICTURE data	Alpha/numeric	EE	Syntax analysis: expanded PICTURE specifications (declare).
		EI	Check for syntax errors local to the statement. Syntax analysis: expanded PICTURE specifications (format list).
		GA	Check for illegal pictures in declarations. Create picture table entries for PICTURE declarations.
		GM	Check for illegal pictures in format lists. Create picture table entries in the general dictionary, for PICTURE format items.
		KV	Optimize PICTURE comparisons.
		KX	Expand PICTURE conversions.
		OC	Expand PICTURE conversions.
		OM	Examine precision, scale, etc., in commoned expressions.
		PC	Build and allocate storage for pictured DEDs and FEDs.
PROCEDURE stmt.	OPTIONS, data attributes multiple labels	EE	Set up a dictionary text file entry chain for each block, and a block-header chain. Syntax analysis: blocking, and statement options.

LANGUAGE FEATURE	FEATURE QUALIFICATIONS	COMPILER PHASE	COMPILER PROCESSING FUNCTIONS
PROCEDURE stmt.	OPTIONS, data attributes multiple labels	GA	Create block-header and entry-point entries in the general dictionary.
		IM	Chain in new external procedure to be called by FORTRAN/COBOL. Create dictionary entries for new external procedure to be called by FORTRAN/COBOL. Insert library calls into text to set up correct environments before and after CALL. Create new external procedure and entry points, and calls to old procedure and entry points.
		KT	Generate prologue code for procedure and entry points.
		OA	Create a block optimization entry.
		OE	Complete the block optimization entry.
		OI	Analyze connection information.
		SA	Generate object code.
PUT stmt.	FILE/STRING, EDIT/LIST/DATA, COPY,SKIP,COL, LINE,PAGE, data-list, format-list	EI	Check for syntax errors local to the statement. Syntax analysis: options, data lists, and DO-groups.
		KA	Insert statements into statement-type chains.
		KQ	Check for: errors in statement-type chains, invalid items in data lists, invalid options on statements. Match data and format lists. Create dictionary entries for FEDs. Allocate registers for special usage in stream I/O. Expand stream I/O statements to include library calls, CONV tables, etc.
		PC	Allocate storage for symbol table built for PUT DATA.
		SK	Detect statement too large.
RE/IRREDUCIBLE		GA	Set a bit in the entry-point dictionary entry.

Compiler processing functions listed by language feature

LANGUAGE FEATURE	FEATURE QUALIFICATIONS	COMPILER PHASE	COMPILER PROCESSING FUNCTIONS
READ stmt.	FILE,INTO/SET, KEY/KEYTO, IGNORE,EVENT, NOLOCK	EE	Syntax analysis: options.
		II	Check for illegal options.
		IM	Create dictionary entries for COBOL-mapped temporary structures for I/O with COBOL files. Create assignments between COBOL-mapped temporary structures and original structures.
		KA	Insert statements into statement-type chains.
		KM	Generate library call, or inline code when possible.
RECURSIVE opt.		GA	Set a bit in the block-header entry.
RETURN stmt.	Expression	EA	Syntax analysis of RETURN expressions.
		IE	Check for illegal use of aggregates.
		KT	Check that a RETURN expression is scalar. Expand the statement.
		OE	Analyze flow information.
		SA	Generate object code.
RETURNS	Attribute	EE	Syntax analysis: returned-value attributes (on DCL and PROC/ENTRYs statements).
		GA	Check for conflicting attributes. Set a bit in the dictionary entry.
REVERT stmt.		EA	Syntax analysis of condition names.
REWRITE stmt.	FILE,KEY, FROM,EVENT	EA	Syntax analysis of condition names.
		EE	Syntax analysis: options.
		II	Check for illegal options.

LANGUAGE FEATURE	FEATURE QUALIFICATIONS	COMPILER PHASE	COMPILER PROCESSING FUNCTIONS
REWRITE stmt.	FILE,KEY, FROM,EVENT	IM	Create dictionary entries for COBOL-mapped temporary structure for I/O with COBOL files. Create assignments between COBOL-mapped temporary structures and original structures.
		KA	Insert statements into statement-type chains.
		KM	Generate library call.
		KT	Check whether a computational condition is enabled. Expand the statement.
Scale	FIXED/FLOAT	EE	Syntax analysis: arguments.
		GA	Check for implementation limit violations.
		II	Check for illegal conversions. Analyze expressions prior to generation of CONV, etc., tables. Generate Type 2 text tables from expressions.
		KK	Expand arithmetic and mathematical bifs.
		OM	Examine precision, scale, etc., in commoned expressions.
Scope	IN/EXTERNAL	GA	Check for conflicting attributes. Set a bit in the dictionary entry.
		PE	Allocate storage for ADCONs for EXTERNAL variables.
Storage class		EE	Syntax analysis: BASED pointers.
		GA	Check for conflicting attributes. Set a bit in the dictionary entry.
		PA	Create entries in the constants pool.
		PE	Allocate static and dynamic storage.
		PI	Allocate temporary storage.
		QA	Assign absolute register numbers where required.

Compiler processing functions listed by language feature

LANGUAGE FEATURE	FEATURE QUALIFICATIONS	COMPILER PHASE	COMPILER PROCESSING FUNCTIONS
Storage class		QE	Allocate storage for global temporaries.
		QI	Relocate temporary storage.
String data	BIT/CHAR/PIC, fixed/VARYING, non-adjustable/ adjustable	EE	Syntax analysis: arguments and REFER items.
		GA	Add details of attributes to string item's dictionary entry.
		IA	Create locator-descriptors for adjustable-length cases. Aggregate mapping for adjustable-length cases. Syntax expansion for adjustable-length cases.
		II	Analyze expressions, perform argument matching, etc., and generate Type 2 text tables.
		IQ	Map and allocate storage for adjustable strings.
		KV	Optimize usage of the UNSPEC, CHAR, BIT, and SUBSTR bifs and PSVs.
		OC	Expand TRANSLATE and BOOL bifs. Expand character and bit string assignments.
		OX	Expand REPEAT, LENGTH, INDEX, VERIFY, HIGH, LOW, and STRING bifs and pseudovariables.
		PA	Create locator-descriptor skeletons. Allocate storage for locator-descriptor skeletons.
		PE	Map temporary storage for aggregate temporaries.
		PI	Addressing of strings. Addressing of locator-descriptors. Map temporary storage, with the exception of aggregate temps.
		SC	Code production for string operaticns.
String op.	BIT/CHARACTER	EE	Syntax analysis: arguments and REFER items.
		II	Analyze expressions.

LANGUAGE FEATURE	FEATURE QUALIFICATIONS	COMPILER PHASE	COMPILER PROCESSING FUNCTIONS
String op.	BIT/CHARACTER	IQ	Map adjustable strings.
		KA	Process all STRING-operation statements, and check for errors.
		OC	Expand =, ||,&|,¬,GT,GE,E,NE,LE,LT string operations.
		OM	Commoning and moving of expressions.
		OX	Optimize BC, BCB, and complex string operations.
		SC	Generate object code.
Structure		EE	Syntax analysis: level numbers.
		GE	Create aggregate descriptor dictionary entries.
		ID	Create aggregate temporaries and reduced descriptors in arguments and LIST I/O.
		IE	Check for errors in expressions. Syntax expansion: expressions.
		II	Check for illegal usage in expressions.
		IM	Create dictionary entries for COBOL-mapped temp. structures for ILC calls or I/O with COBOL files. Create assignments between COBOL-mapped temporary structures and original structures.
		IQ	Generate text tables to produce mapping and storage allocation code. Create aggregate table entries in the general dictionary. Create locator-descriptors for string and aggregate mapping. Map structures. Generate text tables to produce mapping code.
		PA	Construct structure locators and descriptors if required.
		PC	Expand structure text in DATA-directed I/O.
		PE	Allocate storage for structure.

Compiler processing functions listed by language feature

LANGUAGE FEATURE	FEATURE QUALIFICATIONS	COMPILER PHASE	COMPILER PROCESSING FUNCTIONS
Subscripted qualified name		EA	Syntax analysis: arguments, qualifications, and double subscripts.
		EE	Syntax analysis: arguments, qualifications, and double subscripts.
		EI	Check for syntax errors local to the statement. Syntax analysis: arguments, qualifications, and double subscripts.
		GM	Resolve names.
		II	Generate Type 2 text.
SIGNAL stmt.		PC	Allocate storage for symbol tables built for SIGNAL CHECK.
STATIC storage		GA	Check for conflicting attributes. Set a bit in the dictionary entry.
		IA	STATIC INITIAL syntax expansion.
		IQ	Map STATIC aggregates.
		PA	Create entries in the constants pool. Allocate storage for and map static initial storage.
		PE	Allocate storage for STATIC variables.
		SI	Expand constants into TXT records.
		SM	List static storage.
STOP stmt.		KT	Expand the statement.
TASK attribute		KK	Expand tasking bifs.
WAIT stmt.		EA	Syntax analysis: options.
		IE	Check for illegal use of aggregates.
		KA	Insert statements into statement-type chains.

LANGUAGE FEATURE	FEATURE QUALIFICATIONS	COMPILER PHASE	COMPILER PROCESSING FUNCTIONS
WAIT stmt.		KT	Expand the statement.
WRITE stmt.	FILE,FROM, KEYFROM,EVENT	EE	Syntax analysis: options.
		II	Check for illegal options.
		IM	Create dictionary entries for COBOL-mapped temporary structures for I/O with COBOL files. Create assignments between COBOL-mapped temporary structures and original structures.
		KA	Insert statements into statement-type chains.
		KM	Generate library call, or inline code when possible.

Compiler processing functions listed by language feature

COMPILER PROCESSING FUNCTIONS LISTED BY PHASE

Note: For ease of reference, this part of the directory is organized in alphabetic order of phase name, rather than in phase-loading order.

COMPILER PHASE	LANGUAGE FEATURE	FEATURE QUALIFICATIONS	COMPILER PROCESSING FUNCTIONS
BA	All language		Convert 48-char source to 60-char, and BCD to EBCDIC (MACRO not specified).
CA	All language		Print listing if INSOURCE specified. Convert 48-char source to 60-char, and BCD to EBCDIC (MACRO specified).
	Compile-time (%) stmts.		Remove all % symbols except from %PAGE, %SKIP, %CONTROL, %PRINT, and %NOPRINT statements. Execute all compile-time (%) statements.
EC	All language		Move keyword and Translate tables onto 1 or 2 text pages for use by syntax analysis.
EA	Arithmetic op.		Operator/operand syntax analysis.
	ALLOCATE stmt.	CONTROLLED/ BASED,SET,IN	Syntax: page splitting.
	Compile-time (%) stmts.		Execute %PAGE, %SKIP, %CONTROL, %PRINT, and %NOPRINT statements if SOURCE specified.
	Constant		Syntax analysis: format constants.
	DECLARE stmt.		Syntax analysis: page splitting.
	DEFAULT stmt.		Syntax analysis: page splitting.
	DO stmt.	TO,BY,WHILE	Analyze type compatibility and syntax of repetitive specifications.
	END stmt.	Label	Syntax analysis: block nesting.
	FREE stmt.	CONTROLLED/ BASED	Syntax analysis: namelists.
	GO TO stmt.	Label const/var	Analyze target syntax.
	ON stmt.	On-unit/SYSTEM, SNAP	Syntax analysis: block handling and condition argument lists.
	ON-condition		Syntax analysis: arguments.
	Prefix		Syntax analysis: checklists and prefix switches.
	RETURN stmt.	Expression	Syntax analysis of RETURN expressions.
	REVERT stmt.		Syntax analysis of condition names.

COMPILER PHASE	LANGUAGE FEATURE	FEATURE QUALIFICATIONS	COMPILER PROCESSING FUNCTIONS
EA	REWRITE stmt.	FILE,KEY, FROM,EVENT	Syntax analysis of condition names.
	Subscripted qualified name		Syntax analysis: arguments, qualifications, and double subscripts.
EE	Arithmetic op.		Syntax analysis: operator/operand.
	ALLOCATE stmt.	CONTROLLED/ BASED,SET,IN	Set up a dictionary text file ALLOCATE chain for each block. Check attributes of allocated variables, and check SET/IN options.
	AREA		Syntax analysis: AREA arguments.
	Base	DECIMAL/BINARY	Syntax analysis: arguments.
	BEGIN stmt.	Options	Set up a dictionary text file entry chain for each block, and a block-header chain. Syntax analysis: options and arguments.
	Constant		Syntax analysis: format constants.
	CLOSE stmt.	ENV	Syntax analysis: options.
	Dimension		Syntax analysis: expressions and REFER items.
	DECLARE stmt.		Set up a dictionary text file DECLARE chain for each block. Syntax analysis: nesting and factoring.
	DEFAULT stmt.		Set up a dictionary text file DEFAULT chain for each block. Syntax analysis: nesting and factoring.
	DEFINED	POSITION/iSUB	Syntax analysis: bases.
	DELAY stmt.		Syntax analysis: options.
	DISPLAY stmt.	REPLY,EVENT	Syntax analysis: options.
	ENTRY attribute	Parameter attribute list ORDER/REORDER	Syntax analysis: parameter attribute lists in ENTRY declarations.
	ENTRY stmt.	Multiple labs.	Set up a dictionary text file entry chain for each block. Syntax analysis: options.

Compiler processing functions listed by phase

COMPILER PHASE	LANGUAGE FEATURE	FEATURE QUALIFICATIONS	COMPILER PROCESSING FUNCTIONS
EE	FILE	Const/var, STREAM/RECORD, KEYED,INPUT, OUTPUT,etc.	Syntax analysis: record I/O arguments.
	GENERIC		Syntax analysis: valid clauses.
	INITIAL	CALL/nocall	Syntax analysis: INITIAL values, with replication and nesting.
	LOCATE stmt.	FILE,SET, KEYFROM	Syntax analysis: options.
	Mode	REAL/COMPLEX	Syntax analysis: arguments.
	OPEN stmt.	ENV, options.	Syntax analysis: options.
	Precision		Syntax analysis: arguments.
	PICTURE data	Alpha/numeric	Syntax analysis: expanded PICTURE specifications (declare).
	PROCEDURE stmt.	OPTIONS, data attributes multiple labels	Set up a dictionary text file entry chain for each block, and a block-header chain. Syntax analysis: blocking, and statement options.
	READ stmt.	FILE,INTO/SET, KEY/KEYTO, IGNORE,EVENT, NOLOCK	Syntax analysis: options.
	RETURNS	Attribute	Syntax analysis: returned-value attributes (on DCL and PROC/ENTRYs statements).
	REWRITE stmt.	FILE,KEY, FROM,EVENT	Syntax analysis: options.
	Scale	FIXED/FLOAT	Syntax analysis: arguments.
	Storage class		Syntax analysis: BASED pointers.
	String data	BIT/CHAR/PIC, fixed/VARYING, non-adjustable/ adjustable	Syntax analysis: arguments and REFER items.
	String op.	BIT/CHARACTER	Syntax analysis: arguments and REFER items.
	Structure		Syntax analysis: level numbers.
	Subscripted qualified name		Syntax analysis: arguments, qualifications, and double subscripts.

COMPILER PHASE	LANGUAGE FEATURE	FEATURE QUALIFICATIONS	COMPILER PROCESSING FUNCTIONS
EE	WRITE stmt.	FILE,FROM, KEYFROM,EVENT	Syntax analysis: options.
EI	Arithmetic op.		Check for syntax errors local to the statement. Syntax analysis: operator/operand.
	Array x-section		Check for syntax errors local to the statement.
	Constant		Check for syntax errors local to the statement. Syntax analysis: format constants.
	Expression	Data type	Check for syntax errors local to the statement.
	FILE	Const/var, STREAM/RECORD, KEYED,INPUT, OUTPUT,etc.	Check for syntax errors local to the statement. Syntax analysis: stream I/O arguments.
	FORMAT stmt.		Check for syntax errors local to the statement. Syntax analysis: arguments, repetitive specifications, and factored items.
	GET stmt.	FILE/STRING, EDIT/LIST/DATA, COPY,SKIP,COL, data-list, format-list	Check for syntax errors local to the statement. Syntax analysis: options, data lists, and DO-groups.
	IF stmt.	THEN,ELSE	Syntax analysis: nesting and position.
	Mode	REAL/COMPLEX	Check for syntax errors local to the statement.
	PICTURE data	Alpha/numeric	Check for syntax errors local to the statement. Syntax analysis: expanded PICTURE specifications (format list).
	PUT stmt.	FILE/STRING, EDIT/LIST/DATA, COPY,SKIP,COL, LINE,PAGE, data-list, format-list	Check for syntax errors local to the statement. Syntax analysis: options, data lists, and DO-groups.
	Subscripted qualified name		Check for syntax errors lcoal to the statement.

Compiler processing functions listed by phase

COMPILER PHASE	LANGUAGE FEATURE	FEATURE QUALIFICATIONS	COMPILER PROCESSING FUNCTIONS
EI	Subscripted qualified name		Syntax analysis: arguments, qualifications, and double subscripts.
GA	ALIGNED/ UNALIGNED		Check for conflicting attributes. Set a bit in the dictionary entry.
	AREA		Check for conflicting attributes. Set a bit in the dictionary entry.
	AUTOMATIC stge.		Check for conflicting attributes. Set a bit in the dictionary entry.
	Base	DECIMAL/BINARY	Check for implementation limit violations.
	Built-in		Create dictionary entries for explicitly-declared bifs.
	BASED storage	REFER	Check for conflicting attributes. Set a bit in the dictionary entry.
	BEGIN stmt.	Options	Check the validity of options on BEGIN statements. Create a block-header dictionary entry for the block.
	CTL storage		Check for conflicting attributes. Set a bit in the dictionary entry.
	DECLARE stmt.		Detect declaration errors. Create dictionary entries for explicitly declared names. Unfactor attributes.
	DEFAULT stmt.		Detect DEFAULT specification errors. Build the DEFAULT directory.
	DEFINED	POSITION/iSUB	Check for conflicting attributes. Set a bit in the dictionary entry.
	ENTRY attribute	Parameter attribute list ORDER/REORDER	Create parameter descriptor entries in the general dictionary.
	ENTRY stmt.	Multiple labs.	Create entry-point entries in the general dictionary.

COMPILER PHASE	LANGUAGE FEATURE	FEATURE QUALIFICATIONS	COMPILER PROCESSING FUNCTIONS
GA	FILE	Const/var, STREAM/RECORD, KEYED,INPUT, OUTPUT,etc.	Check for conflicting attributes. Create file constant dictionary entries. Set a bit in the dictionary entry.
	INITIAL	CALL/nocall	Check for conflicting attributes. Set a bit in the dictionary entry.
	Locator	POINTER/AREA	Check for conflicting attributes. Set a bit in the dictionary entry.
	LIKE		Check for conflicting attributes. Expand 'LIKE' specifications.
	Mode	REAL/COMPLEX	Check for implementation limit violations.
	Parameter storage		Check for conflicting attributes. Set a bit in the dictionary entry.
	PICTURE data	Alpha/numeric	Check for illegal pictures in declarations. Create picture table entries for PICTURE declarations.
	PROCEDURE stmt.	OPTIONS, data attributes multiple labels	Create block-header and entry-point entries in the general dictionary.
	RE/IRREDUCIBLE		Set a bit in the entry-point dicticnary entry.
	RECURSIVE opt.		Set a bit in the block-header entry.
	RETURNS	Attribute	Check for conflicting attributes. Set a bit in the dictionary entry.
	Scale	FIXED/FLOAT	Check for implementation limit violations.
	Scope	IN/EXTERNAL	Check for conflicting attributes. Set a bit in the dictionary entry.
	Storage class		Check for conflicting attributes. Set a bit in the dictionary entry.
	String data	BIT/CHAR/PIC, fixed/VARYING, non-adjustable/ adjustable	Add details of attributes to string item's dictionary entry.
	STATIC storage		Check for conflicting attributes.

Compiler processing functions listed by phase

COMPILER PHASE	LANGUAGE FEATURE	FEATURE QUALIFICATIONS	COMPILER PROCESSING FUNCTIONS
GA	STATIC storage		Set a bit in the dictionary entry.
GE	ALLOCATE stmt.	CONTROLLED/ BASED,SET,IN	Check allocate attributes against declaration attributes. Merge allocation and declaration information. Create allocate-time aggregate table dictionary entries. Generate extent and INITIAL assignments.
	BASED storage	REFER	Detects errors in REFER declaration. Puts REFER information in aggregate descriptors.
	Dimension		Create aggregate descriptor dictionary entries.
	DEFAULT stmt.		Use the DEFAULT directory.
	DEFINED	POSITION/iSUB	Diagnose source errors. Generate addressing statements. Classify defining.
	GENERIC		Check for attribute and entry point errors. Create dictionary entries.
	INITIAL	CALL/nocall	Generate INITIAL assignment tables.
	Structure		Create aggregate descriptor dictionary entries.
GI	Built-in		Create dictionary entries for contextually declared bifs.
	CALL stmt.	Argument list, function ref., options.	Create dictionary entries from contextual declarations for bifs, and TASK or EVENT variables.
	DEFAULT stmt.		Use the DEFAULT directory.
	FILE	Const/var, STREAM/RECORD, KEYED,INPUT, OUTPUT,etc.	Create dictionary entries for contextually declared files.
	Label	Constant	Create dictionary entries for label constants.
	ON stmt.	On-unit/SYSTEM, SNAP	Create dictionary entries for contextually declared conditions.

COMPILER PHASE	LANGUAGE FEATURE	FEATURE QUALIFICATIONS	COMPILER PROCESSING FUNCTIONS
GM	Assignment statement (element)	Data types, multiple	Check for errors on left hand side.
	Built-in		Create dictionary entries for implicitly declared bifs.
	Constant		Create dictionary entries for constants.
	DEFAULT stmt.		Use the DEFAULT directory.
	FILE	Const/var, STREAM/RECORD, KEYED,INPUT, OUTPUT,etc.	Create dictionary entries for implicitly declared files.
	GO TO stmt.	Label const/var	Check for label constant or variable.
	OPEN stmt.	ENV, options	Gather OPEN file attributes into a dictionary entry.
	PICTURE data	Alpha/numeric	Check for illegal pictures in format lists. Create picture table entries in the general dictionary, for picture format items.
	Subscripted qualified name		Resolve names.
IA	ALLOCATE stmt.	CONTROLLED/ BASED,SET,IN	Reposition/generate bound assignments to aggregate descriptors. Generate mapping operators for allccated variables. Syntax expansion: extent expression assignments. Convert offset locators to pointers.
	Built-in		Check for illegal use of bifs as locator qualifiers. Syntax expansion: special cases.
	BASED storage	REFER	Generate RESDES operators to reserve space for special temporary locator-descriptors. Examine references to BASED REFER, etc., and decide whether or not mapping is necessary. If so, generate code to set up descriptors and MAP operators. Syntax expansion: map-on-reference cases and unqualified BASED variables.

Compiler processing functions listed by phase

COMPILER PHASE	LANGUAGE FEATURE	FEATURE QUALIFICATIONS	COMPILER PROCESSING FUNCTIONS
IA	DEFINED	POSITION/iSUB	Generate ASSN tables for creation of locator-descriptors. Insert MAP operators into the main text stream. Insert POS expressions and iSUB lists into the main text stream.
	FREE stmt.	CONTROLLED/ BASED	Syntax expansion for implied AREAs.
	INITIAL	CALL/nocall	Generate multiple assignments and AID operators in the main text stream.
	Locator	POINTER/AREA	Check for illegal qualifier chains. Generate locator chains.
	LOCATE stmt.	FILE,SET, KEYFROM	Create special locator-descriptors for REFER structures. Special aggregate-mapping action for REFER structures. Special syntax expansion action for REFER structures.
	Pseudovariable		Detect illegal arguments. Syntax expansion: special cases.
	String data	BIT/CHAR/PIC, fixed/VARYING, non-adjustable/ adjustable	Create locator-descriptors for adjustable-length cases. Aggregate mapping for adjustable-length cases. Syntax expansion for adjustable-length cases.
	STATIC storage		STATIC INITIAL syntax expansion.
ID	Array x-section		Create reduced descriptors in arguments, LIST I/O, and WAIT. Insert text for filling in reduced descriptors.
	Built-in		Check for errors in aggregate-manipulation bifs and pseudovariables. Create temporaries or reduced descriptors.
	CALL stmt.	Argument list, function ref., options.	Create aggregate temporaries, including reduced descriptors.

COMPILER PHASE	LANGUAGE FEATURE	FEATURE QUALIFICATIONS	COMPILER PROCESSING FUNCTIONS
ID	Dimension		Check number of subscripts in subscript lists. Create aggregate temporaries and reduced descriptors in arguments, I/O, and WAIT. Flag statement header to indicate subscripted variable present.
	DEFINED	POSITION/iSUB	Flag statement header to indicate iSUB-defined item present.
	Expression	Data type	Check the validity of aggregate expression arguments. Insert text-assigning aggregate expression arguments to temporaries.
	Structure		Create aggregate temporaries and reduced descriptors in arguments and LIST I/O.
IE	Array x-section		Check for errors. Syntax expansion: generation of dc-loop control temporaries in assignments and stream I/O expressions.
	Assignment statement (aggregate)	Array/struc., multiple, BY NAME	Check for errors in non-scalar assignments. Syntax expansion: non-scalar assignments. Expand character and bit string aggregates assignments.
	Assignment statement (element)	Data types, multiple	Check for errors in scalar assignments (when the statement contains scalar-returning functions which in turn have embedded aggregate assignments). Syntax expansion: scalar assignments.
	ALLOCATE stmt.	CONTROLLED/ BASED,SET,IN	Check for illegal use of aggregates in attribute specification.
	Built-in		Syntax expansion: aggregate bifs and pseudovariables.
	BASED storage	REFER	Check for errors. Syntax expansion: generation of temporaries.
	CALL stmt.	Argument list, function ref., options.	Check for errors in CALL statements. Syntax expansion: CALL statements.
	Dimension		Check for errors in expressions.

Compiler processing functions listed by phase

COMPILER PHASE	LANGUAGE FEATURE	FEATURE QUALIFICATIONS	COMPILER PROCESSING FUNCTIONS
IE	Dimension		Syntax expansion: expressions.
	DECLARE stmt.		Check for illegal use of aggregates in attribute specification.
	DEFAULT stmt.		Check for illegal use of aggregates in attribute specification.
	DEFINED	POSITION/iSUB	Syntax expansion: iSUB-U statements containing functions with aggregate assignments.
	DELAY stmt.		Check for illegal use of aggregates.
	DISPLAY stmt.	REPLY,EVENT	Check for illegal use of aggregates.
	GO TO stmt.	Label const/var	Check for illegal use of aggregates.
	IF stmt.	THEN,ELSE	Check for illegal use of aggregates.
	Locator	POINTER/AREA	Check for errors. Syntax expansion: generation of temporaries.
	OPEN stmt.	ENV, options	Check for illegal aggregates.
	Prologue text		Scan-analysis and error checking in prologue text. Sorting of prologue text. Syntax expansion: prologue text (in functions with embedded aggregate assignments).
	Pseudovariable		Check for errors in aggregate bifs and pseudovariables. Syntax expansion: aggregate bifs and pseudovariables.
	RETURN stmt.	Expression	Check for illegal use of aggregates.
	Structure		Check for errors in expressions. Syntax expansion: expressions.
	WAIT stmt.		Check for illegal use of aggregates.
II	All language		Translate from Type-1 to Type-2 text.
	Arithmetic op.		Analyze expressions.
	Assignment statement (element)	Data types, multiple	Check for mismatch between source and target, and for illegal target.

COMPILER PHASE	LANGUAGE FEATURE	FEATURE QUALIFICATIONS	COMPILER PROCESSING FUNCTIONS
II	Assignment statement (element)	Data types, multiple	Complete dict. entries for targets in chameleon temp. assignments. Translate Type-1 text assignments to ASSN and CONV tables.
	ALIGNED/ UNALIGNED		Argument-matching/generic selection.
	ALLOCATE stmt.	CONTROLLED/ BASED,SET,IN	Translate Type-1 text extent expressions, MAP statements, and ALLOCATE statements to Type-2 text tables.
	Base	DECIMAL/BINARY	Check for illegal conversions. Analyze expressions prior to generation of CONV, etc., tables. Generate Type-2 text tables from expressions.
	Built-in		Argument checking. Generate BIF tables and CONV tables for arguments.
	Constant		Check for illegal usage, and convert arguments to temps. Generate Type 2 text.
	Conversion		Check for illegal conversions. Generate CONV Type-2 text tables.
	CALL stmt.	Argument list, function ref., options.	Check for errors in argument matching. Translate CALL statements from Type-1 to Type-2 text, generating ALIST, ARG, CALL, FNCT, etc., text tables.
	Dimension		Check for illegal usage in expressions.
	DO stmt.	TO,BY,WHILE	Check for illegal DO specifications. Translate Type-1 text into Type-2 text tables (DO1, DO2, etc.).
	Expression	Data type	Check validity of expressions containing file, entry, label, etc.
	ENTRY stmt.	Multiple labs.	Translate from Type-1 to Type-2 text, and generate GSLs.
	FILE	Const/var, STREAM/RECORD, KEYED,INPUT, OUTPUT,etc.	Check for illegal usage in expressions.

Compiler processing functions listed by phase

COMPILER PHASE	LANGUAGE FEATURE	FEATURE QUALIFICATIONS	COMPILER PROCESSING FUNCTIONS
II	FORMAT stmt.		Generate FORME, FIT, etc. Type-2 text tables.
	GENERIC		Check for possibility of selection. Perform selection. Generate ALIST, ARG, CALL, etc. Type-2 text tables.
	IF stmt.	THEN,ELSE	Check for illegal expressions in IF clauses. Translate Type-1 text IF statements into Type-2 text IF, GOTO, and GSL tables.
	Label	Constant	Check for illegal usage in expressions and assignments.
	Locator	POINTER/AREA	Generate PTS, PTSAT, etc. Type-2 text tables.
	LABEL	Variable	Check for illegal usage in expressions and assignments. Chaining.
	Mode	REAL/COMPLEX	Check for illegal conversions. Analyze expressions prior to generation of CONV, etc., tables. Generate Type-2 text tables from expressions.
	Prefix		Generate Type-2 text.
	READ stmt.	FILE,INTO/SET, KEY/KEYTO, IGNORE,EVENT, NOLOCK	Check for illegal options.
	REWRITE stmt.	FILE,KEY, FROM,EVENT	Check for illegal options.
	Scale	FIXED/FLOAT	Check for illegal conversions. Analyze expressions prior to generation of CONV, etc., tables. Generate Type-2 text tables from expressions.
	String data	BIT/CHAR/PIC, fixed/VARYING, non-adjustable/ adjustable	Analyze expressions, perform argument matching, etc., and generate Type-2 text tables.
	String op.	BIT/CHARACTER	Analyze expressions.
	Structure		Check for illegal usage in expressions.

COMPILER PHASE	LANGUAGE FEATURE	FEATURE QUALIFICATIONS	COMPILER PROCESSING FUNCTIONS
II	Subscripted qualified name		Generate Type-2 text.
	WRITE stmt.	FILE,FROM, KEYFROM,EVENT	Check for illegal options.
IK	All language.		On specification of the XREF and/or ATTRIBUTES compiler options, prints cross-references and attributes of all identifiers.
IM	CALL stmt.	Argument list, function ref., options	Create dictionary entries for temps. created for arguments needing to be remapped before calls to FORTRAN/COBOL. Insert library calls into text to set up correct environments before and after CALL.
	Dimension		Create dictionary entries for FORTRAN-mapped temp. arrays for calls to or from arguments or parameter in ILC. Remap temp. arrays for calls to or from FORTRAN. Create assignments between FORTRAN-mapped temps. and original arrays.
	ENTRY stmt.	Multiple labs.	Create dictionary entries for new external procedure to be called by FORTRAN/COBOL. Insert library calls into text to set up correct environments before and after CALL. Create new external procedure and entry points, and calls to old procedure and entry points.
	FILE	Const/var, STREAM/RECORD, KEYED,INPUT, OUTPUT,etc.	Create dictionary entries for COBOL-mapped temporary structures for I/O with COBOL files. Create assignments between COBOL-mapped temporary structures and original structures.
	LOCATE stmt.	FILE,SET, KEYFROM	Create dictionary entries for COBOL-mapped temporary structures for I/O with COBOL files. Create assignments between COBOL-mapped temporary structures and original structures.
	PROCEDURE stmt.	OPTIONS, data attributes multiple labels	Chain in new external procedure to be called by FORTRAN/COBOL. Create dictionary entries for new external procedure to be called by FORTRAN/COBOL.

Compiler processing functions listed by phase

COMPILER PHASE	LANGUAGE FEATURE	FEATURE QUALIFICATIONS	COMPILER PROCESSING FUNCTIONS
IM	PROCEDURE stmt.	OPTIONS, data attributes multiple labels	Insert library calls into text to set up correct environments before and after CALL. Create new external procedure and entry points, and calls to old procedure and entry points.
	READ stmt.	FILE,INTO/SET, KEY/KEYTO, IGNORE,EVENT, NOLOCK	Create dictionary entries for COBOL-mapped temporary structures for I/O with COBOL files. Create assignments between COBOL-mapped temporary structures and original structures.
	REWRITE stmt.	FILE,KEY, FROM,EVENT	Create dictionary entries for COBOL-mapped temporary structure for I/O with COBOL files. Create assignments between COBOL-mapped temporary structures and original structures.
	Structure		Create dictionary entries for COBOL-mapped temp. structures for ILC calls or I/O with COBOL files. Create assignments between COBOL-mapped temporary structures and original structures.
	WRITE stmt.	FILE,FROM, KEYFROM,EVENT	Create dictionary entries for COBOL-mapped temporary structures for I/O with COBOL files. Create assignments between COBOL-mapped temporary structures and original structures.
IQ	Array x-section		Map arrays.
	Assignment statement (aggregate)	Array/struc, multiple, BY NAME	Map aggregates.
	ALIGNED/ UNALIGNED		Map ALIGNED/UNALIGNED BIT strings in aggregates.
	ALLOCATE stmt.	CONTROLLED/ BASED,SET,IN	Allocate storage for BASED and CONTROLLED aggregates.
	AUTOMATIC stge.		Map AUTOMATIC aggregates.
	BASED storage	REFER	Map BASED aggregates.
	CONTROLLED storage		Map CONTROLLED aggregates.
	FREE stmt.	CONTROLLED/ BASED	Free allocated storage.

COMPILER PHASE	LANGUAGE FEATURE	FEATURE QUALIFICATIONS	COMPILER PROCESSING FUNCTIONS
IQ	LOCATE stmt.	FILE,SET, KEYFRCM	Map aggregates as for BASED variables.
	String data	BIT/CHAR/PIC, fixed/VARYING, non-adjustable/ adjustable	Map and allocate storage for adjustable strings.
	String op.	BIT/CHARACTER	Map adjustable strings.
	Structure		Generate text tables to produce mapping and storage allocation code. Create aggregate table entries in the general dictionary. Create locator-descriptors for string and aggregate mapping. Map structures. Generate text tables to produce mapping code.
	STATIC storage		Map STATIC aggregates.
KA	Assignment statement (aggregate)	Array/struc, multiple, BY NAME	Insert statements into statement-type chains.
	ALLOCATE stmt.	CONTROLLED/ BASED,SET,IN	Process and error-check resulting ARRAY INITIAL and STATIC INITIAL assignments.
	Built-in		Process STRING bifs and check for errors.
	CALL stmt.	Argument list, function ref., options.	Check for errors (match number of arguments against number of parameters).
	DECLARE stmt.		Process and error-check resulting ARRAY INITIAL and STATIC INITIAL assignments.
	DO stmt.	TO,BY,WHILE	Insert statements into statement-type chains. Complete the processing of the DOWHILE statement.
	END stmt.	Label	Insert statements into statement-type chains.
	FUNCTION refs.		Check for errors (match number of arguments against number of parameters).

Compiler processing functions listed by phase

COMPILER PHASE	LANGUAGE FEATURE	FEATURE QUALIFICATIONS	COMPILER PROCESSING FUNCTIONS
KA	GET stmt.	FILE/STRING, EDIT/LIST/DATA, COPY,SKIP,COL, data-list, format-list	Insert statements into statement-type chains.
	IF stmt.	THEN,ELSE	Complete the statement processing. Insert statements into statement-type chain. Representative text table processing, and statement analysis.
	ON stmt.	On-unit/SYSTEM, SNAP	Insert statements into statement-type chains.
	OPEN stmt.	ENV, options	Insert statements into statement-type chains.
	PUT stmt.	FILE/STRING, EDIT/LIST/DATA, COPY,SKIP,COL, LINE,PAGE, data-list, format-list	Insert statements into statement/type chains.
	READ stmt.	FILE,INTO/SET, KEY/KEYTO, IGNORE,EVENT, NOLOCK	Insert statements into statement-type chains.
	REWRITE stmt.	FILE,KEY, FROM,EVENT	Insert statements into statement-type chains.
	String op.	BIT/CHARACTER	Process all STRING-operation statements, and check for errors.
	WAIT stmt.		Insert statements into statement-type chains.
	WRITE stmt.	FILE,FROM, KEYFROM,EVENT	Insert statements into statement-type chains.
KE	Array x-section		Expand array cross-section text.
	Assignment statement (aggregate)	Array/struc, multiples, BY NAME	Expand array and structure assignments, and compiler-generated do-loops.
	DEFINED	POSITION/iSUB	Expand iSUB defining text.
	INITIAL	CALL/nocall	Expand automatic array initial text.
KI	DO stmt.	TO,BY,WHILE, UNTIL,REPEAT	Process do-loops.

COMPILER PHASE	LANGUAGE FEATURE	FEATURE QUALIFICATIONS	COMPILER PROCESSING FUNCTIONS
KI	DO stmt.	TO,BY,WHILE, UNTIL,REPEAT	Expand DO1, DO2, and DO3 into appropriate assignments and branch to produce codes.
KK	Base	DECIMAL/BINARY	Expand arithmetic and mathematical bifs.
	Built-in		Check the validity of arguments to certain bifs. Expand all non-string bifs.
	DEFAULT stmt.		Process and error-check resulting ARRAY INITIAL and STATIC INITIAL assignments.
	EVENT attribute		Expand EVENT bifs.
	GENERIC		Process bifs passed as arguments.
	Mode	REAL/COMPLEX	Expand arithmetic, mathematical, and COMPLEX bifs.
	Precision		Expand arithmetic and mathematical bifs.
	Pseudovariable		Check the validity of arguments to certain pseudovariables. Expand non-string pseudovariables.
	Scale	FIXED/FLOAT	Expand arithmetic and mathematical bifs.
	TASK attribute		Expand tasking bifs.
KL	CLOSE stmt.	ENV	Generate an argument list and library module call for the statement.
	DECLARE stmt.		Check FILE declaration for conflicting ENVIRONMENT options and attributes, and for correct MEDIUM specification. Create DTF, ENVB, and FCB entries in the general dictionary, and a names dictionary entry for the generated LIOCS module name required for the DTF.
	FILE	Const/var, STREAM/RECORD, KEYED,INPUT, OUTPUT,etc.	Check file declaration for conflicting ENVIRONMENT options and attributes, and for correct MEDIUM specification. Create DTF, ENVB, and FCB entries in the general dictionary, and a names dictionary entry for the generated LIOCS module name required for the DTF.
	OPEN stmt.	ENV, options	Check attributes with those declared. Create OCB (Open Control Block) and generate an argument list and library module call for the statement.

Compiler processing functions listed by phase

COMPILER PHASE	LANGUAGE FEATURE	FEATURE QUALIFICATIONS	COMPILER PROCESSING FUNCTIONS
KM	FILE	Const/var, STREAM/RECORD, KEYED, INPUT, OUTPUT, etc.	Optimize opening and closing by completing FCB and DTF at compile time where possible. Check record I/O files.
	LOCATE stmt.	FILE,SET, KEYFROM	Generate library call, or inline code when possible.
	READ stmt.	FILE,INTO/SET, KEY/KEYTO, IGNORE,EVENT, NOLOCK	Generate library call, or inline code when possible.
	REWRITE stmt.	FILE,KEY, FROM,EVENT	Generate library call.
	WRITE stmt.	FILE,FROM, KEYFROM,EVENT	Generate library call, or inline code when possible.
KQ	Format item		Create dictionary entries for FEDs.
	FORMAT stmt.		Construct FEDs. Expand FMTLST/FORME/FIT/FITE text tables to GEN/LA/CALL, etc.
	GET stmt.	FILE/STRING, EDIT/LIST/DATA, COPY,SKIP,COL, data-list, format-list	Check for: errors in statement-type chains, invalid items in data lists, invalid options on statements. Match data and format lists. Create dictionary entries for FEDs. Allocate registers for special usage in stream I/O. Expand stream I/O statements to include library calls, COBV tables, etc.
	PUT stmt.	FILE/STRING, EDIT/LIST/DATA, COPY,SKIP,COL, LINE,PAGE, data-list, format-list	Check for: errors in statement-type chains, invalid items in data lists, invalid options on statements. Match data and format lists. Create dictionary entries for FEDs. Allocate registers for special usage in stream I/O.

COMPILER PHASE	LANGUAGE FEATURE	FEATURE QUALIFICATIONS	COMPILER PROCESSING FUNCTIONS
KQ	PUT stmt.	FILE/STRING, EDIT/LIST/DATA, COPY,SKIP,COL, LINE,PAGE, data-list, format-list	Expand stream I/O statements to include library calls, CONV tables, etc.
KT	CALL stmt.	Argument list, function ref., options.	Produce calling sequence.
	CHECK condition	Name list	Process CHECK statements, and insert the appropriate library calls for check variables in the text.
	DELAY stmt.		Check that a DELAY expression is scalar. Expand the DELAY statement.
	DISPLAY stmt.	REPLY,EVENT	Check the attributes of DISPLAY, REPLY, and EVENT expressions. Expand the DISPLAY statement.
	ENTRY stmt.	Multiple labs.	Produce appropriate branch code in prologue.
	EXIT stmt.		Expand the EXIT statement.
	PROCEDURE stmt.	OPTIONS, data attributes multiple labels	Generate prologue code for procedure and entry points.
	RETURN stmt.	Expression	Check that a RETURN expression is scalar. Expand the statement.
	REWRITE stmt.	FILE,KEY, FROM,EVENT	Check whether a computational condition is enabled. Expand the statement.
	STOP stmt.		Expand the statement.
	WAIT stmt.		Expand the statement.
KV	Arithmetic op.		Optimize multiplication and exponentiation.
	Built-in		Expand CHAR, BIT, UNSPEC, SUBSTR bifs.
	CALL stmt.	Argument list, function ref., options	Unnest calls.

Compiler processing functions listed by phase

COMPILER PHASE	LANGUAGE FEATURE	FEATURE QUALIFICATIONS	COMPILER PROCESSING FUNCTIONS
KV	GO TO stmt.	Label const/var	Optimize branching.
	Pseudovariable		Process UNSPEC and SUBSTR pseudovariables to produce correct object code.
	PICTURE data	Alpha/numeric	Optimize picture comparisons.
	String data	BIT/CHAR/PIC, fixed/VARYING, non-adjustable/ adjustable	Optimize usage of the UNSPEC, CHAR, BIT, and SUBSTR bifs and psvs.
KX	Arithmetic op.		Set flags for code generation.
	Assignment statement (aggregate)	Array/struc, multiple, BY NAME	Set flags for code generation.
	Assignment statement (element)	Data types, multiple	Set flags for code generation.
	Conversion		Expand text for inline conversions.
	PICTURE data	Alpha/numeric	Expand PICTURE conversions.
OA	Assignment statement (element)	Data types, multiple	Extract alias information concerning locator, label, and entry assignments.
	BEGIN stmt.	Options	Create a block optimization entry.
	CALL stmt.	Argument list, function ref., options	Extract calling information for blocks.
	DEFINED	POSITION/iSUB	Create value lists for defining bases.
	Locator	POINTER/AREA	Create value lists for locators.
	LABEL	Variable	Create value lists for LABEL variables.
	Parameter storage		Create value lists for parameters.
	PROCEDURE stmt.	OPTIONS, data attributes multiple labels	Create a block optimization entry.
OC	Assignment statement (aggregate)	Array/struc, multiple, BY NAME	Expand STRING assignments.
	Assignment statement (element)	Data types, multiple	Expand STRING assignments.

COMPILER PHASE	LANGUAGE FEATURE	FEATURE QUALIFICATIONS	COMPILER PROCESSING FUNCTIONS
OC	Built-in		Expand TRANSLATE and BOOL bifs.
	Conversion		Expand text for inline conversions.
	PICTURE data	Alpha/numeric	Expand PICTURE conversions.
	String data	BIT/CHAR/PIC, fixed/VARYING, non-adjustable/ adjustable	Expand TRANSLATE and BOOL bifs. Expand character and bit string assignments.
	String op.	BIT/CHARACTER	Expand =, ||,&|,¬,GT,GE,E,NE,LE,LT string operations.
OE	BEGIN stmt.	Options	Complete the block optimization entry.
	CALL stmt.	Argument list, function ref., options.	Analyze flow information.
	DO stmt.	TO,BY,WHILE	Analyze flow information.
	END stmt.	Label	Analyze flow information.
	ENTRY stmt.	Multiple labs.	Analyze flow information.
	GO TO stmt.	Label const/var	Analyze flow information.
	Label	Constant	Analyze flow information.
	LABEL	Variable	Analyze flow information.
	ON stmt.	On-unit/SYSTEM, SNAP	Analyze abnormal information (variables set and used in on-units).
	ON-condition		Analyze variable usage information, and abnormal information.
	PROCEDURE stmt.	OPTIONS, data attributes multiple labels	Complete the block optimization entry.
	RETURN stmt.	Expression	Analyze flow information.
OI	Assignment statement (aggregate)	Array/struc, multiple, BY NAME	Analyze connection information.
	BEGIN stmt.	Options	Analyze connection information.
	CALL stmt.	Argument list, function ref., options	Analyze connection information.
	DO stmt.	TO,BY,WHILE	Analyze connection information.
	END stmt.	Label	Analyze connection information.

Compiler processing functions listed by phase

COMPILER PHASE	LANGUAGE FEATURE	FEATURE QUALIFICATIONS	COMPILER PROCESSING FUNCTIONS
OI	ENTRY stmt.	Multiple labs.	Analyze connection information.
	GO TO stmt.	Label const/var	Analyze connection information.
	Label	Constant	Analyze connection information.
	LABEL	Variable	Analyze connection information.
	PROCEDURE stmt.	OPTIONS, data attributes multiple labels	Analyze connection information.
OM	Arithmetic op.		Commoning and moving of expressions.
	Assignment statement (element)	Data types, multiple	Commoning and moving of expressions.
	Base	DECIMAL/BINARY	Examine precision, scale, etc., in commoned expressions.
	Constant		Optimize the production and usage of literal constants.
	Conversion		Commoning and moving of expressions.
	DO stmt.	TO,BY,WHILE	Modify control variable usage.
	Mode	REAL/COMPLEX	Examine precision, scale, etc., in commoned expressions.
	Precision		Examine precision, scale, etc., in commoned expressions.
	PICTURE data	Alpha/numeric	Examine precision, scale, etc., in commoned expressions.
	Scale	FIXED/FLOAT	Examine precision, scale, etc., in commoned expressions.
	String op.	BIT/CHARACTER	Commoning and moving of expressions.
OX	Built-in		Optimize REPEAT, LENGTH, INDEX, VERIFY, HIGH, LOW, and STRING bifs.
	Pseudovariable		Process STRING pseudovariable to produce library call, if necessary.
	String data	BIT/CHAR/PIC, fixed/VARYING, non-adjustable/ adjustable	Expand REPEAT, LENGTH, INDEX, VERIFY, HIGH, LOW, and STRING bifs and pseudovariables.
	String op.	BIT/CHARACTER	Optimize BC, BCB, and complex string operations.

COMPILER PHASE	LANGUAGE FEATURE	FEATURE QUALIFICATIONS	COMPILER PROCESSING FUNCTIONS
PA	Arithmetic op.		Generate constants used in arithmetic operations.
	ALLOCATE stmt.	CONTROLLED/ BASED,SET,IN	Allocate storage for controlled anchor slot.
	AREA		Allocate storage and initialize static areas. Construct area locator/descriptor if required.
	Constant		Allocate storage for constants. Convert constant to correct type if required.
	Conversion		Compile-time conversion of constants.
	CALL stmt.	Argument list, function ref., options	Common argument lists. Allocate storage for argument lists.
	CONTROLLED storage		Allocate storage for controlled anchor slot.
	Dimension		Construct array locators and descriptors if required.
	DEFINED	POSITION/iSUB	Allocate addressing adcon or locator if required.
	FILE	Const/var, STREAM/RECORD, KEYED,INPUT, OUTPUT, etc.	Allocate storage for file constants.
	GOTO stmt.	Label const/var	Allocate storage for label constant if go-to-outer block.
	INITIAL	CALL/nocall	Allocate STATIC INITIAL storage.
	Label	Constant	Allocate storage for label constants.
	Mode	REAL/COMPLEX	Allocate storage for COMPLEX constants.
	ON stmt.	On-unit/SYSTEM, SNAP	Allocate storage for static ONCBs.
	Storage class		Create entries in the constants pool.
	String data	BIT/CHAR/PIC fixed/VARYING, non-adjustable/ adjustable	Create locator-descriptor skeletons. Allocate storage for locator-descriptor skeletons.
	Structure		Construct structure locators and descriptors if required.
	STATIC storage		Create entries in the constants pool. Allocate storage for and map static initial storage.

PC	CHECK condition	Name list	Construct symbol tables for items in check lists.
	Data-list item		Check validity of based items in data directed I/O.
	Format item		Allocate storage for FEDs.
	FORMAT stmt.		Allocate storage for FEDs.

COMPILER PHASE	LANGUAGE FEATURE	FEATURE QUALIFICATIONS	COMPILER PROCESSING FUNCTIONS
PC	GET stmt.	FILE/STRING, EDIT/LIST/DATA, COPY,SKIP,COL, data-list, format-list	Allocate storage for symbol tables built for GET DATA.
	PICTURE data	Alpha/numeric	Build and allocate storage for pictured DEDs and FEDs.
	PUT stmt.	FILE/STRING, EDIT/LIST/DATA, COPY,SKIP,COL, LINE,PAGE, data-list, format-list	Allocate storage for symbol tables built for PUT DATA.
	Structure		Expand structure text in DATA-directed I/O.
	SIGNAL stmt.		Allocate storage for symbol tables built for SIGNAL CHECK.
PE	AUTOMATIC stge.		Allocate storage for AUTOMATIC variables.
	BASED storage	REFER	Associate a storage base number with each BASED variable to assist in BASED-variable addressing.
	ENTRY attribute	Parameter attribute list ORDER/REORDER	Count the number of parameters in each block and pass this number to Phase PI.
	LABEL	Variable	Allocate storage for LABEL variables.
	Mode	REAL/COMPLEX	Allocate storage for COMPLEX variables.
	ON stmt.	On-unit/SYSTEM, SNAP	Allocate storage for dynamic CNCBs.
	Scope	IN/EXTERNAL	Allocate storage for ADCONs for EXTERNAL variables.
	Storage class		Allocate static and dynamic storage.
	String data	BIT/CHAR/PIC, fixed/VARYING, non-adjustable/ adjustable	Map temporary storage for aggregate temporaries.
	Structure		Allocate storage for structures.
	STATIC storage		Allocate storage for STATIC variables.
PI	ALLOCATE stmt.	CONTROLLED/ BASED,SET,IN	Addressing of CONTROLLED variables. Create locator-descriptors for CONTROLLED variables.

COMPILER PHASE	LANGUAGE FEATURE	FEATURE QUALIFICATIONS	COMPILER PROCESSING FUNCTIONS
PI	AUTOMATIC stge.		Addressing of AUTOMATIC variables.
	BASED storage	REFER	Addressing of BASED variables.
	CONTROLLED storage		Addressing of CONTROLLED variables. Allocate storage for CONTROLLED variables.
	FREE stmt.	CONTROLLED/ BASED	Addressing of BASED variables.
	Mode	REAL/COMPLEX	Addressing of COMPLEX variables or COMPLEX temporaries.
	ON stmt.	On-unit/SYSTEM, SNAP	Addressing of on-units.
	Parameter storage		Reserve the appropriate storage for the parameter list for each block (number of parameters in each block is passed by Phase PE) in the temporary storage area based on R4.
	Storage class		Allocate temporary storage.
	String data	BIT/CHAR/PIC, fixed/VARYING, non-adjustable/ adjustable	Addressing of strings. Addressing of locator-descriptors. Map temporary storage, with the exception of aggregate temps.
QA	All language		Assign absolute register numbers where required.
QE	All language		Allocate storage for global temporaries. Remove unnecessary stores of global temporaries.
	Storage class		Allocate storage for global temporaries.
QI	All language		Relocate temporary storage. Insert addressing code in prologue or optimal flow unit, if OPT=TIME. Diagnose compiler error conditions. Reserve loop control register for binary loops.
	Prologue text		Insert addressing code in prologue or optimal flow unit, if OPT=TIME.

Compiler processing functions listed by phase

COMPILER PHASE	LANGUAGE FEATURE	FEATURE QUALIFICATIONS	COMPILER PROCESSING FUNCTIONS
QI	Storage class		Relocate temporary storage.
SA	BEGIN stmt.	Options	Generate object code.
	CALL stmt.	Argument list, function ref., options	Generate object code.
	CHECK condition	Name list	Generate object code (for CHECK condition without list).
	END stmt.	Label	Generate object code.
	ENTRY stmt.	Multiple labs.	Generate object code.
	GO TO stmt.	Label const/var	Generate object code.
	ON stmt.	On-unit/SYSTEM, SNAP	Generate object code.
	ON-condition		Generate object code (unqualified ON-condition).
	Prefix		Generate object code.
	Prologue text		Generate object code.
	PROCEDURE stmt.	OPTIONS, data attributes multiple lables	Generate object code.
	RETURN stmt.	Expression	Generate object code.
SC	String data	BIT/CHAR/PIC, fixed/VARYING, non-adjustable/ adjustable	Code production for string operations.
	String op.	BIT/CHARACTER	Generate object code.
SD	Conversion		Generate object code.
SI	Object code		Produce ESD/TXT/RLD records for program, and all external CSECTs for input into linkage editor.
	STATIC storage		Expand constants into TXT records.

COMPILER PHASE	LANGUAGE FEATURE	FEATURE QUALIFICATIONS	COMPILER PROCESSING FUNCTIONS
SK	ALIGNED/ UNALIGNED		Sign unaligned operands of RX instructions.
	FORMAT stmt.		Remove unnecessary code. Detect statement too large.
	GET stmt.	FILE/STRING, EDIT/LIST/DATA, COPY,SKIP,COL, data-list, format-list	Detect statement too large.
	LABEL	variable	Remove if not referenced.
	PUT stmt.	FILE/STRING, EDIT/LIST/DATA, COPY,SKIP,COL, LINE,PAGE, data-list, format list	Detect statement too large.
SM	All language		Diagnose compiler error conditions.
	Object code		List the object code
	STATIC storage		List static storage.
SQ	Arithmetic op.		Generate object code (binary, float, and ext. float ops.).
	CHECK condition	Name list	Generate object code (for CHECK condition with list).
	ON-condition		Generate object code (qualified ON-condition).

Compiler processing functions listed by phase

This section describes the format and usage of most data areas that are processed by more than one phase. The meanings of some fields in the data areas described may vary according to the phase and/or the usage of the data area. In this section, the descriptions of alternative meanings for fields are separated by double vertical lines. A single vertical line indicates a general description applied to two or more fields. Symbolic names for fields are shown where they are common to all phases which process them. Data areas are described under the following main headings:

Communication area (XCOMM)
Basic data-handling information
Dictionary entries
Operand code bytes
Six-byte references to operands
Compile-time DEDs
Type-1 text formats
Type-2 text formats
Pseudo constants pool
Extended code formats
Preprocessor dictionary entries

COMMUNICATION AREA - XCOMM

Bytes Hex	Bytes Dec	Symbol	Meaning	
000-047	0-71	XSA1	System-calls save area.	Save areas.
048-08F	72-143	XSA2	Control-calls save area.	
090-093	144-147	XACTL	Control Phase address.	Control Phase addresses.
094-097	148-151	XDMADR	Dump Phase address.	
098-09B	152-155	XAA0300	Phase loading routine.	These 4-byte entries contain the start addresses of the various routines/ phases.
09C-09F	156-159	XAA4200	Dictionary reference page search routine.	
0A0-0A3	160-163	XAA4000	Page accessing routine.	
0A4-0A7	164-167	XAA4100	Dictionary reference page search routine.	
0A8-0AB	168-171	XAA4500	Spill I/O check routine.	
0AC-0AF	172-175			
0B0-0B3	176-179	XAA8000	Discard page routine.	
0B4-0B7	180-183			
0B8-0BB	184-187	XAA0600	Program interrupt handler.	
0BC-0BF	188-191	XAA0500	End of compilation return address.	
0C0-137	192-311	XINPUT	Input dataset.	I/O dataset DTF blocks.
138-1CF	312-463	XPRINT	Print dataset.	
1D0-267	464-615	XSPILL	Spill dataset.	

268-26F	616-623	XCTXUN	Text UNMOVABLE. 00	Page chain head and tail pointers.
270-277	624-731	XCTXMV	Text MOVABLE 08	00, 08, etc. are status chain codes used in OSTAT in each page header.
278-27F	632-639	XCDCUN	Dictionary UNMOVABLE. 10	
280-287	640-647	XCDCMV	Dictionary MOVABLE. 18	
		XDIRPA	Address of dictionary directory.	
288-28F	648-655	XCDRUN	Directory UNMOVABLE 20	
		XCDIRY		
290-297	656-663	XCdrmv	Directory MOVABLE 28	
298-29F	664-671	XCUNUS	UNUSED pages. 30	
2A0-2A7	672-679		Not used.	
2A8-2AF	680-687	XCDISC	DISCARDED pages. 40	
2B0-2B2	688-690	XTXINF	Text page, page in main storage.	Page search sequences for, text, dictionary, and directory pages
2B3-2B8	691-696		Text page, page not in main storage.	
2BA	697	XDUREQ	Dummy required after this phase	
2BA-2BB	698-699	XPCNT	In-core page count.	
2BC	700	XDUCNT	Number of abort calls to dump phase.	
2BD	701	XDICNT	Number of abort calls to error-editor phase.	
2BE-2BF	702-703	XNRT	Number of pages/track.	
2C0-2C2	704-706	XDCINF	Dictionary page, page in core.	
2C3-2C8	707-712		Dictionary page, page not in core.	
2C9	713	XSSW	Page chain select flag byte.	
2CA-2CB	714-715	XPNO	Number of core pages.	
2CC-2CD	716-717	XTXC	Number of text pages in compiler.	
2CE-2CF	718-719	XDCC	Number of dictionary pages in compiler.	
2D0-2D2	720-722	XDRINF	Directory page, page in core.	
2D4-2D8	723-728		Directory page, page not in core.	
2D9	729		Padding bytes.	
2DA-2DD	730-733	XOBJSZ	Size of object code.) Note: These two	
2DE-2DF	734-735	XNUMST	Number of statements.) fields are liable to change.	
2E0-2E3	736-739	XATASL	Address of latest spare TA slot.	Discarded track address table.
2E4-2E6	740-742	XTASL		
2E7-33D	743-829	XTASL1	Discarded TA slots.	
33E-33F	830-831			
340-34F	832-847	XDIRTA	Directory page TAs.	Directory

					variables.
350-353	848-851		Not used.		
354-355	852-853	XDIRNO	4*number of directory pages.		
356-357	854-855		Not used.		
358-359	856-857	XREF	Next reference in general dictionary.	XSQTBL Information on current entries.	Dictionary Information Tables.
35A-35B	858-859	XOFFST	Offset of next reference in general dictionary.		
35C	860	XALGLN	General dictionary alignment length=10 bytes.		
35D-35F	861-863	XTA	TA of current general dictionary page.		
360-361	864-865	XREF	These four fields as above but for the names dictionary.		
362-363	866-867	XOFFST			
364	868	XALGLN	Alignment length=5 bytes.		
365-367	869-871	XTA			
368-369	872-873	XREF	These four fields as above but for the variables dictionary.		
36A-36B	874-875	XOFFST			
36C	876	XALGLN	Alignment length=40 bytes.		
36D-36F	877-879	XTA			
370-371	880-881	XREF	These four fields as above but for the storage dictionary.		
372-373	882-883	XOFFST			
374	884	XALGLN	Alignment length=40 bytes.		
375-377	885-887	XTA			

378-37B	888-891		Not used.	XMSKTB Masks for dictionary references.	Dictionary Information Tables. (contd.).
37C-37D	892-893		Not used.		
37E-37F	894-895	XELTH	General dictionary entry length = 10 bytes.		
380-383	896-899		These three fields as above but for the names dictionary.		
384-385	900-901				
386-387	902-903	XELTH	Entry length = 5 bytes.		
388-38B	904-907		These three fields as above but for the variables dictionary.		
38C-38D	908-909				
38E-38F	910-911	XELTH	Entry length = 40 bytes.		
390-393	912-915		These three fields as above but for the storage dictionary.		
394-395	916-917				
396-397	918-919	XELTH	Entry length = 40 bytes.		
398-399	920-921	XDICEN	No. of entries/page - general dictionary.	XDRTBL Directory segmentation table.	
39A-39B	922-923	XCRDRF	Current directory section offset.		
39C-39D	924-925	XDROFS	Offset of section from directory page.		
39E-39F	926-927	XSECSZ	Size of section in directory page.		
3A0-3A1	928-929	XDICEN	These four fields as above but for the names dictionary.		
3A2-3A3	930-931	XCRDRF			
3A4-4A5	932-933	XDROFS			
3A6-3A7	934-935	XSECSZ			
3A8-3A9	936-937	XDICEN	These four fields as above but for the variables dictionary.		
3AA-3AB	938-939	XCRDRF			
3AC-3AD	940-941	XDROFS			
3AE-3AF	942-943	XSECSZ			
3B0-3B1	944-945	XDICEN	These four fields as above but for the storage dictionary.		
3B2-3B3	946-947	XCRDRF			
3B4-3B5	948-949	XDROFS			
3B6-3B7	950-951	XSECSZ			
3B8-407	952-1031	XREC1	1st source record.		Control variables
408	1032	XSYNSV	Severity of SYNTAX option.		
409	1033	XCMPSV	Severity of COMPILE option.		
40A	1034	XFLGSV	Severity of FLAG option.		

40B	1035	XLNKSV	Severity of LINK option.	Control variables (cont'd.)
40C-433	1036-1075	XNAME	Character string for phase card.	
434-43D	1076-1085	XNSYGBT	System generation option bits.	
43E-447	1086-1095	XNDELBT	Option delete bits.	
448-467	1096-1127	XCGRSZ	Compiler-generated subroutine sizes.	
468-46B	1128-1131	XLOCK	Dict. type and reference of locked page.	
46C-46F	1132-1135	XLOADL	1st component of loaded phase name.	
470-473	1136-1139	XLPHSN	2nd component of loaded phase name.	
474-477	1140-1143	XAPHS	Phase start address.	
478-47B	1144-1147	XPGST	Start of page area.	
47C-47F	1148-1151	XPGND	End of page area.	
480-483	1152-1155	XABIF	Address of next buffer - input dataset.	
484-487	1156-1159	XABPF	Address of next buffer - print dataset.	
488-48D	1160-1163	XAPFSH	Address of print dataset subheading.	
48C-48F	1164-1167	XNWTA1	TA of latest I/O page.	
490-491	1168-1169	XRTKL1	Remainder track length.	
492-493	1170-1171	XNTC1	Number of tracks/cylinder.	
494-497	1172-1175	XSAVTA1	Last TA in previous member of batch.	
498-49B	1176-1179	XOPICA	Old program-interrupt control area address.	
49C-4C3	1180-1219	XTEMP	Temporary storage.	
4C4-4C7	1220-1223	XDSTP	Number of discarded text pages.	
4C8-4CB	1224-1227	XABCF	Address of next buffer - punch dataset.	
4CC-4CF	1228-1231	XABOF	Address of next buffer - load dataset.	
4D0-4D1	1232-1233	XPFLN	Line number of print dataset.	
4D2-4D3	1234-1235	XPFPN	Page number (decimal) of print dataset.	
4D4-4D5	1236-1237	XMUNDP	Max. number of UNMOVABLE dict. pages.	
4D6-4D7	1238-1239	XNUNDP	Number of UNMOVABLE dictionary pages.	
4D8-4DB	1240-1243	XAEND	Partition end address.	

4DC-4E3	1244-1251	XMXDRF	Maximum directory offset for each dict. reached by the preprocessor stage.	Control variables (cont'd.).
4E4-4E6	1252-1254	XCIOTA	TA of unchecked I/O page.	
4E7-4E8	1255-1256	XCPHSN	Name of current phase or overlay.	
4E9	1257	XPHSIN	Phase description byte.	
4EA	1258	XDCSTA	Type bits of current phase.	
4EB	1259	XCTLCD	Control request code.	
4EC	1260	XCSWS	Control phase switches.	
4ED	1261	XBATCH	Batching progress switch.	
4EE	1262	XPROG	Compilation progress switch.	
4EF	1263	XRETCD	Return code.	
4F0-4F1	1264-1265	XAREF	Dictionary ref. of AREA INIT constant.	
4F2-4F3	1266-1267	XNSREC	Number of source records.	
4F4-4F7	1268-1271	XCPL0	XCON0 Constant 0.	Absolute constants.
4F8-4F9	1272-1273	XCONZH	XCON1 Constant 1.	
4FA-4FB	1274-1275	XCPL1		
4FC-4FD	1276-1277		XCON2 Constant 2.	
4FE-4FF	1278-1279	XCPL2		
500-501	1280-1281		XCON3 Constant 3.	
502-503	1282-1283	XCPL3		
504-505	1284-1285		XCON4 Constant 4.	
506-507	1286-1287	XCPL4		
508-509	1288-1289		XCON16 Constant 16.	
50A-50B	1290-1291	XCPL16		
50C-50A	1292-1295	XMN1	XBPO Bit pattern (-).	
510-511	1296-1297		XBP1 Bit pattern (-4).	
512-513	1298-1299	XBP1H		
514-515	1300-1301		XBP2 Bit pattern (zero top byte).	
516-517	1302-1303	XBP2H		
518-51B	1304-1307	XTPCLR	XBP3 Bit pattern (zero top halfword).	

51C	1308	XUPSI	User program switch indicators.	Compilation constants.
51D-524	1309-1316	XJOBN	Job name.	
525	1317		Not used.	
526-527	1318-1319	XPFLS	Print-line size - number of characters.	
528-529	1320-1321	XPFPS	Print-page size - number of lines.	
52A-52B	1322-1323	XSOR	Left-hand source margin.	
52C-52D	1324-1325	XMAG	Right-hand source margin.	
52E-52F	1326-1327	XCC	Carriage control character position in source.	
530-531	1328-1329	XTRCN	Execution time trace number.	
532	1330	XMGIN	Margin indicator character.	
533	1331		Not used.	
534-537	1332-1335	XPHSM	Maximum length of phase.	
538-53B	1336-1339	XSIZE	SIZE option, or default partition size.	
53C-53D	1340-1341	XPAGS	Overall page size.	
53E-53F	1342-1343	XPAGSU	Useable page size.	
540-543	1344-1347	XTIMU	Time of day (units of 1/300 sec.).	
544-54B	1348-1353	XTIME	Job time of day.	
54C-555	1356-1365	XDATE	Date.	
556-557	1366-1367	XILRECL	Input record length.	
558-55B	1368-1371	XMCOUNT	Number of pages used during this member of the batch.	
55C-55F	1372-1375	XSTX	Text reference of first input page.	Text variables.
560-563	1376-1379	XCABS	Current output page address.	
564-567	1380-1383	XSIPT	Start of current input page.	
568-56B	1384-1387	XINSV	Text reference of text start in page.	
56C-56F	1388-1391	XSCRCH	Scratch page pointer.	
570-571	1392-1393		Not used.	
572-573	1394-1395	XNABN	Next available base number.	
574-575	1396-1397	XSPOP	Space left in current output page.	
576-57A	1398-1402	XACDRF	Head of addressing code reference.	
57B	1403		Not used.	

57C-580	1404-1408	XDOCH	Head of statement type chain.	Text variables (cont'd.).
581-585	1409-1413	XRIOCH	Head of statement type chain.	
586-58A	1414-1418	XLABREF	Phases SA-SK: 1st label in pseudo constants pool.	
		XSIOCH	Phases KA-KQ: Head of GET/PUT/FORMAT statement chain.	
58B-58F	1419-1423	XADCPG	Phases SK-SI: Adcon page.	
		XSYSCH	Head of statement type chain.	
590-594	1424-1428	XIFCH	Not used.	
595-599	1429-1433	XSUBCH	Head of statement type chain.	
59A-59E	1434-1438	XLOGCH	Not used.	
59F-5A3	1439-1443	XBELCH	Head of statement type chain.	
5A4-5A6	1444-1446	X2STRM	Current 2nd text stream page reference.	
5A7-5A9	1447-1449	XTRFL	Text reference of first loop.	
5AA	1450	XNSRTSW	Switches for Type 2 text handling macros.	
			1... .... | XNSRT. Inserted table is on same page as table before.	
			.1.. .... | XNSRT. No 'LINK=' parameter.	
			..1. .... | XLINK. 'IND' specified.	
			...1 .... | XLINK. 'REF=' specified.	
			.... 1... | XNSRT. No third parameter specified.	
			Bits 5-7 | Not used.	
5AB	1451	XTXB0	XTXPG macro information byte.	
5AC-5AF	1452-1455	XCR	Output pointer.	XOUTARG (First text stream).
5B0-5B3	1456-1459	XBK	Last break point.	
5B4-5B7	1460-1463	XSP	Start of current page.	
5B8-5BB	1464-1467	XPGEND	End of current page.	
5BC-5BE	1468-1470	XCREF	Reference of current page.	
5BF	1471	XOUTST	Status of old pages.	

5C0-5C3	1472-1475	XCR2	Output pointer.	XOUTARG2 (Second text stream).	Text variables (cont'd.).
5C4-5C7	1476-1479	XBK2	Last break point.		
5C8-5CB	1480-1483	XSP2	Start of current page.		
5CC-5CF	1484-1487	XPGEND2	End of current page.		
5D0-5D2	1488-1490	XCREF2	Reference of current page.		
5D3	1491	XOUTST2	Status of old pages.		
5D4-5D7	1492-1495	XATERM	User end of dictionary chain routine.		Dictionary variables.
5D8	1496		Padding.		
5D9	1497	XCODBT	Dictionary type of reference.		
5DA-5DB	1498-1499		Dictionary reference used for XRFAB.		
5DC-5DD	1500-1501	XAGHEDA	Head of static array chain.		
5DE-5DF	1502-1503	XAGHEDS	Head of static structure chain.		
5E0-5E1	1504-1505	XDCRF	Dictionary reference in current call.		
5E2-5E3	1506-1507	XDCLTH	Dictionary entry length.		
5E4-5E5	1508-1509	XBH	Block header chain.		
5E6-5E7	1510-1511	XDFCH	Head of default chain.		
5E8-5E9	1512-1513	XBSCH	Based chain.		
5EA-5EB	1514-1515	XPICCH	Head of picture chain.		
5EC-5ED	1516-1517	XALIAS	Dictionary reference of alias strings.		
5EE-5EF	1518-1519	XFILCH	Head of file chain.		
5F0	1520	XSCNSW	Dictionary scan switch.		
5F1	1521	XCTLSV	Dictionary routine control code.		
5F2	1522	XDICSW	Directory routine calling switch.		
5F3-5F4	1523-1524	XRKDCH	Record/key descriptor chain		
5F5	1525	XDEVSW	Development switch.		
5F6-5F7	526-527	XNREF	Head of contextual declarations.		
5F8-5FB	1528-1531	XTCOF	Index slot address - XN5.		Statement trace information.
5FC-5FF	1532-1535	XTCEND	Trace table end address - XN9.		

600-607	1536-1543	XDPPSW	Abort PSW.	Dump routine information.
608-647	1544-1607	XDPRGS	Abort registers.	
648-64B	1608-1611	XSTGAD	Start of storage address.	
64C-64F	1612-1615	XSTGND	End of storage address.	
650-661	1616-1633	XDPOPT	Dump options.	
662-663	1634-1635	XDYSTS	Start of dynamic dump statement range.	
664-665	1636-1637	XDYSTE	End of dynamic dump statement range.	
666-667	1638-1639	XTTOP	Highest temporary storage base number.	
668-66B	1640-1643	XSTATSZ	Size of static storage.	Source program variables.
66C-66F	1644-1647	XCPSIZ	Size of constants pool.	
670-671	1648-1649	XSTAT	Current statement number.	
672-673	1650-1651	XNLAB	Total number of labels in program.	
674-675	1652-1653	XNTEMP	Total number of temporaries.	
676-677	1654-1655	XADCS	Storage allocated - estimate of number of program adcons.	
678-679	1656-1657	XSAADCS	Code generated - estimate of number of program adcons.	
67A-67B	1658-1659	XULAB	Number of user labels in source program.	
67C-67D	1660-1661	XBLKCT	Block count.	
67E	1662	XNML	Length of procedure name.	
67F-685	1663-1669	XNM	Procedure name.	
686-688	1670-1672	XCPREF	Constants pool text page reference.	
689	1673	XDECAR	Decimal arithmetic flag.	
68A-68B	1674-1675	XSAVOFF	Static address vector offset.	
68C-68D	1676-1677	XADCON1	First adcon offset in static.	
68E-68F	1678-1679	XNARG	Current argument list number.	
690-691	1680-1681	XSTAD	Number of static address adcons.	
694-697	1684-1687	XRUTAD	Address of RUT table.	
698-6E5	1688-1765	XLIBSTR	Bit vector indicating library subroutines called.	
6E6-6E7	1766-1767	XCOMSTR	Bit vector indicating compiler-generated subroutines called.	

6E8-6E9	1768-1769	XAAERR	Control phase terminal error.	Error message variables.
6EA	1770		Not used.	
6EB	1771	XACTS	Highest diagnostic severity to date.	
6EC-6ED	1772-1773	XMREM	Remaining space in error page.	
6EE-6EF	1774-1775	XMNUM	Numeric parameter for message.	XMESS. Total error message.
6F0-F1	1776-1777	XMDRF	Error message dictionary reference.	
6F2	1778	XMDTP	Error message dictionary type.	
6F3	1779	XERFLGS	Error and message flags.	
6F4-6F6	1780-1782	XMPRF	Error message page reference.	
6F7	1783		Not used.	
6F8-6FF	1784-1791	XN12	Save area for RE and RF.	
700-703	1792-1795	XN7	Length of trace table.	
704	1796	XSCLNG1	Source language flags.	
			1...	Check prefix.
			.1..	Allocate with attributes.
			..1.	Aggregates.
			...1	PLICOM required.
			.... 1...	Expressions in default specification in outer block.
			.... .1..	Main outer procedure.
			.... ..1.	Extended FLOAT in source.
			1	ILC outer block.
705	1797	XSCLNG2	Spare.	
706	1798	XOPPHS1	Optional phases flag.	
707	1799	XOPPHS2	Optional phases flag 2.	
708-70B	1800-1803	XBCHLNF	ID:PE total branch-table length. PE:SM offset in STATIC of first branch-table.	
70C-70E	1804-1806	XBCHREF	Anchor point for text page containing label constants in branch-tables.	
70F	1807		Spare.	
710-711	1808-1809	XENVCH	Environment chain header.	
712-719	1810-1817	XCATNM	Library name.	

71A-71B	1818-1819	XLIOCS	Chain head for LIOCS names (dictionary).	
71C-71D	1820-1821	XCNDCH	Condition chain head.	
71E-71F	1822-1823	XEXECH	External entry chain head.	
720-721	1824-1825	XLVLAB	Label variable range.	
722-771	1826-1905	XIFBF1	Input dataset buffers.	Buffers.
772-7C1	1906-1985	XIFBF2		
7C2-834	1986-2106	XPRBF1	Print dataset buffers.	
83B-8B3	2107-2227	XPRBF2		
8B4-923	2228-2339	XPHSCM	Communication area for consecutive phases.	
924-927	2340-2343	XNWTP	Number of new text page requests.	Page monitoring variables.
928-92B	2344-2347	XNWDP	Number of new dictionary page requests.	
92C-92F	2348-2351	XEXTP	Existing text page requests.	
930-933	2352-2355	XEXDP	Existing dictionary page requests.	
934-937	2356-2359	XEXTPI	I/O for existing text requests.	
938-93B	2360-2363	XEXDPI	I/O for existing dictionary requests.	
93C-93F	2364-2367	XRWSC	Spill candidate - read/write.	
940-943	2368-2371	XDSSC	Spill candidate - discard.	
944-947	2372-2375	XROSC	Spill candidate - read only.	
948-94B	2376-2379	XURRW	Number of unwritten read/write pages.	
94C-94F	2380-2383	XWRRO	Number of unwritten read-only pages.	
950-953	2384-2387	XNOPG	Number of existing pages.	
954-98F	2388-2447	XPHASE	Phase options (Dump and/or test phase identifier)	
9A0-A87	2464-2695	XPUNCH	Punch dataset.	
A88-AD8	2696-2776	XPFBF1	Punch dataset buffers.	
AD9-B2B	2777-2859	XPFBF2		

B2C-B2F	2860-2863	XCFILE	File parameter address.	Second spill file parameters.
B30-B33	2864-2867	XNWTA2	New track address.	
B34-B35	2868-2869	XRTKL2	Remainder track length.	
B36-B37	2870-2872	XNTC2	Number of tracks/cylinder.	
B38-B3B	2872-2875	XSAVTA2	Last track address in previous member of batch.	
B3C-B3F	2876-2879	XCRF	Current read file address.	
B40-B43	2880-2883	XCWF	Current write file address.	
B44-B47	2884-2887	XCSPC	Current spill candidate.	
B48	2888	XFILE	Current dataset.	
B49	2889		Work space.	
B4A-B4B	2890-2891	XCRFC	Current read file code.	
B4C-B4F	2892-2895		Not used.	
B50-BE7	2896-3047	XSPILL2	Second spill dataset.	
BE8-BEB	3048-3051	XMINTA2	Track address at start.	
BEC-BEF	3052-3055	XCSP	Current spill candidate.	

Figure 5.1. Communication Area - XCOMM

Bytes	Symbol	Meaning		
0-3	OCNFD	Forward chain field for the status chain.	Page header.	This part of the page space is not written onto the spill dataset at any time during compilation.
4-7	OCNBK	Backward chain field for the status chain.		
8	OSTAT .0.. 00.. .0.. 01.. .0.. 10.. .0.. 11.. .1.. 01..	Page Status: UNMOVABLE read/write. UNMOVABLE read only. MOVABLE read/write. MOVABLE read only. DISCARDED.		
9	ODCTP	Dictionary type if dict. page. OTXCN. Text page chain, if text.		Record.
10-11	ODCRF	Dictionary page number.		
12-15	OTKAD	Track address (TA).		
16-		1080, 1680, 3480, or 4040 bytes of processable data (directory, dictionary, text, or scratch). The first 32 bytes are taken up by an overflow page index table (see figure 5.3) during Type 2 text processing.		
		Four-byte page delimiter (X'99000000').		

Figure 5.2. Page-space format

Bytes	Symbol	Meaning	
0-3	OCNFD	Chain forward	
4-7	OCNBK	Chain back	
8-12	OSTAT ODCTP ODCRF	See figure 5.2 for meaning	Page header 1
12-15	OKTAD	Track address	
16-19		Pointer to page	
			Page header 2 (contents as above)
			Page header n

Note: The page header table is a copy of page header information held contiguously. For meaning of bit settings, see figure 5.2. The page header table is addressed from the page header chains starting in XCOMM.

Figure 5.2.1 Format of page header table

Bytes	Symbol	Meaning
0	IPHCOD	Type 2 text code byte (STPGE), indicating table type.
1	IPHFLG	Flag byte.
2-25	ISPILB	Eight three-byte fields to contain the TAs of overflow pages. A TA consists of cylinder number, track number, and record number within the track.
26-27	IOFSPA	Offset within this page at which free space commences.
28-31		Not used.

Figure 5.3. Format of overflow page index tables in Type-2 text

Both Type 1 and Type 2 text refer to items in the text stream by use of five-byte text references, and to dictionary entries by use of two-byte dictionary references. To access a dictionary entry, the two-byte reference is first converted to a page address and offset (as described below for a text reference). The conversion is described in section 2, "Method of Operation". The format of a five-byte text reference is shown below.

Bytes	Meaning	
0	Cylinder number.	Unique track address (TA), constituting the name of a compiler page.
1	Track number within the cylinder.	
2	Record number within the track.	
3-4	Offset within the page identified above.	

Figure 5.4. Five-byte text reference format

DICTIONARY ENTRIES

The four types of dictionary, with the main features of their entries, are shown in the following table.

Dictionary type	Alignment of entries	Format of entries	Length of entries	Created in phase
Names	5-byte	fixed	variable	GA onwards
Variables	40-byte	fixed	40 bytes	GA onwards
General	10-byte	variable	variable or fixed	GA onwards
Storage	40-byte	fixed	40 bytes	PE

Figure 5.5. Dictionary entry types

The format of a dictionary entry is indicated by the dictionary type and the entry's operand code byte.

The different types of dictionary entry are described in figures 5.6 to 5.28 as follows:

Figure 5.6. Format of names dictionary entries
Figure 5.7. Format of variables dictionary entries
Figure 5.8. Format of general dictionary block-header entries
Figure 5.9. Format of general dictionary entry-constant entries
Figure 5.10. Format of general dictionary parameter-descriptor entries
Figure 5.11. Format of general dictionary aggregate-table entries
Figure 5.12. Format of general dictionary picture-table entries
Figure 5.13. Format of general dictionary constant entries
Figure 5.14. Format of general dictionary file-constant entries
Figure 5.15. Format of general dictionary FCB entries
Figure 5.16. Format of FCB entry STREAM I/O block
Figure 5.17. Format of FCB entry RECORD I/O block
Figure 5.18. Format of general dictionary ENVB entries
Figure 5.19. Format of general dictionary DTF entries
Figure 5.20. Format of general dictionary record and key descriptor entries
Figure 5.21. Contents of a RECORD descriptor
Figure 5.22. Contents of a KEY descriptor
Figure 5.23. Format of general dictionary OCB entries
Figure 5.24. Single optimization entry for the whole program, in the general dictionary
Figure 5.25. Optimization entries for blocks, in the general dictionary
Figure 5.26. Format of general dictionary value-list entries
Figure 5.27. Format of general dictionary overflow entries
Figure 5.28. Format of storage dictionary entries

Bytes	Symbol	Meaning
0	YCDE	Operand code byte indicating type of indentifier (see figure 5.29).
1-2	YBHSH	Hash chain field.
3-4	YLC	Block level and count.

Figure 5.6. (Part 1 of 2). Format of names dictionary entries

Bytes	Symbol	Meaning	
5-6	YBCNG	Dictionary reference of the names dictionary entry for the containing structure.	YBBIF. Description of a built-in function (bif). These entries are created in Phase GI.
7	YBMNTP	Byte indicating whether the associated entry is in the variables or general dictionary.	
8-9	YBMNRF	Dictionary reference of the associated entry.	
10	YBCDL	Length of the name.	
11-	YBCD	Name.	

Figure 5.6. (Part 2 of 2). Format of names dictionary entries

Bytes	Symbol	Meaning	
0	YCDE	Operand code byte (X'80' to X'FF'). See figures 5.30 to 5.37.	
1-2	YSN	Statement number of declaration statement.	
3-4	YLC	Block level and count.	
5-6	YNMRF	Reference of names dictionary entry.	
7-9	YDED	Data element descriptor (DED). See figure 5.59.	
10-11	YPIC YPTAT	Reference of picture table entry, if picture. Reference of value list entry, if locator or label.	Offset of symbol table element.
12-13	YVPOS YAREA	Value of POS expression, if DEFINED string or picture. Reference of AREA, if OFFSET variable.	
14	YVALN	Object-time alignment.	
15-17	YVSIZ	Object-time size.	
18	YVNDIM	Object-time dimensionality.	
19	YVSEQ	Sequence number of this structure member.	
20	YVCTG	Sequence number of the containing structure.	
21	YVSLVL	Structure level.	
22-23	YDV YDSCR	Dictionary reference of an aggregate descriptor. Dictionary reference of a record or key descriptor.	
24-25	YVPTR YVDEF YVSIN	Dictionary reference of a base pointer. Dictionary reference of defined base. Offset of STATIC INITIAL information.	
26	YVPGE	Text page in which the	
27-28		variables declaration is held.	Offset of static locator.

Figure 5.7. (Part 1 of 2). Format of variables dictionary entries

Bytes	Symbol	Meaning
29	YV1FL	First flag byte:
	1...	Last member of a structure.
	.1..	Major structure containing adjustable extents.
	..1.	External.
	...1	Initialized.
	.... 1...	DEFINED, or argument to the ADDR function.
	.... .1..	Pre-GM: alignment is specified (i.e., not system default).
	.... .1..	Post-GM: the variable is in a check list.
	.... ..1.	Pre-OE: Connected parameter or apparently connected aggregate.
	.... ..1.	Post-OA: Parameter appears at every entry point to the procedure in which it is declared.
	1	Default entry for a default bif.
30	YV2FL	Second flag byte:
	.... 1...	Post PC: Variable has symbol table element.
	.... .1..	Post PC: Variable has symbol table.
	.... ..1.	Pre-GE: last descriptor in a parameter list.
	.... ..1.	Post-GE: the variable has a corresponding declaration expressions text stream statement.
	1	Post-PA: a skeleton locator exists in static.
		The remaining bits of the byte are used to indicate how the information represented by certain fields is referenced. The bits are related to fields in the entry as follows:
	X...	Bytes 12-13.
	.X..	Bytes 24-25.
	..X.	Bytes 31-32.
	...X	Bytes 10-11: this is used only on GA output, for a label variable.
	.... X...	Bytes 8-9: used only for adjustable string or AREA.
	.... .X..	Bytes 22-23.
	1111 11..	Phases GA-GM: a '1' bit in any of the positions shown indicates that the corresponding field contains the general dictionary reference of an overflow entry (figure 5.27) containing a text reference to the required information.
	0000 00..	Phases GA-GM: a '0' bit in any of the positions shown indicates that the text reference to the required information (if any) is YVPGE||field.
	1...	Post-GM: the text reference to the required information is YVPGE||field.
	0.00 0...	Post-GM: a '0' bit in any of the positions shown indicates that the required information is contained in the field.
31-32	YVOFF	Offset in YVPGE of a declaration expression of INITIAL information, or the symbol table offset.
33-34	YVLCTR	Code byte and data byte of the variables locator, if BASED.
35-36	YVDSCR	Descriptor's dictionary reference, if ENTRY || YVALTH. AREA length if default entry.
37-39		||

Figure 5.7. (Part 2 of 2). Format of variables dictionary entries

Bytes	Symbol	Meaning
0	YCDE	Operand code byte:
	0011 0000	PROCEDURE block.
	0011 0001	BEGIN block.
	0011 0010	ON block.
	0011 0011	ONB block.
1-2	YSN	Statement number. | Post PC: No. of static ONCBs in block.
3-4	YLC	Block level and count.
5-6	YHNXB	Dictionary reference of next entry in the block-header chain. | Post PC: Size of dynamic ONCB storage for block.
7	YHCOND	Code for on-unit (zero if the block is not an on-unit).
8		Not used.
9-10	YHCB	Dictionary reference of containing block's block-header entry.
11	YHFL	Flag byte, set by Phase GA:
	1...	There are parameters in the block.
	.1..	All parameter lists are the same.
	..1.	There is more than one entry point to the block.
	...1	There is more than one RETURN type in the block.
	.... 1...	REORDER option applies.
	.... .1..	MAIN option applies.
	.... ..1.	REENTRANT option applies.
	1	RECURSIVE option applies.
12-13	YPTAT	Dictionary reference of the optimization entry.
14-15	YHPOPT	Prefix options for this block.
16	YH2FL	Flag byte:
	1...	FILE condition in the block.
	.1..	Block contains CALL FORTRAN or COBOL.
	..1.	BEGIN block.
	...1	ON block.
	.... 1...	ON FILE condition in the block.
	.... .1..	On-unit with GO TO label constant.
	.... ..1.	CALL with TASK option.
17		Not used.
18-19	YHOPCH	Options changed in the block.
20-21	YENCH	Circular entry chain for internal entry-point entries.
22-23	YHSYLST	Offset of symbol table element list.
24-26	YBLSZ	Block size (excluding names).
27-29	YDIAG1	Number of entries in statement-number table.
30-32	YDIAG2	Number of entries in statement-number table.

Figure 5.8. Format of general dictionary block-header entries

Bytes	Symbol	Meaning
0	YCDE	Operand code byte. See figure 5.34.
1-2	YSN	Statement number.
3-4	YLC	Block level and count.
5-6	YNHRF	Reference of the corresponding names dictionary entry.
7-13		Information concerning the characteristics of the returned value (if any), as described for bytes 7-13 in figure 5.7, "Format of variables dictionary entries."
14-15	YECL	GSL(compiler prologue label). || Post PC: YEPSYM: Offset of symbol table.
16-17	YENCH	Entry-point chain field.
18-19	YEADC	Static storage offset of the adcon.
20	YENFP	Number of formal parameters in the following list.
21-	YEFP	Variable length list containing either variables dictionary entry reference for each parameter if this is an INTERNAL entry constant, or parameter descriptor entry reference for each parameter if this is an EXTERNAL entry constant.

Figure 5.9. Format of general dictionary entry-constant entries

Parameter Type	Format of Descriptor Entry
Variable	As for a variables dictionary entry, but the names dictionary is not used.
File constant	As for a file constant entry. See figure 5.14.
Entry constant	As for an entry constant. See figure 5.9.

Figure 5.10. Format of general dictionary parameter descriptor entries

Bytes	Symbol	Meaning
0	YCDE	Operand code byte (X'10').
1-2	YTCHN	Chain field - each entry is linked into one of five chains, mainly for commoning purposes.
Pre-GE		
3-7	YTAJEXT	Text reference to the declaration expressions text stream.
Phases GE-IA		
3-4	YTCMN	Chain head of aggregates to which this tables refers.
5	YTNMEM	Number of members in major structure.
6-7		Not used.
Post-IA		
3		Not used.
4-5	YTMJR	Text reference of major structure.
6-7	YTCNG	Text reference of containing structure member.
Post-PA		
3	YTKEFL	Flag byte:
	.... 1...	Aggregate descriptor in static.
4-5		Offset of descriptor in static.
8-9	YTNXT	Dictionary reference of the next member within this structure.
10	YTDM	Total number of dimensions.
11	YTXDM	Number of dimensions excluding inherited dimensions.
12	YTLVL	Structure level.

Figure 5.11. (Part 1 of 3). Format of general dictionary aggregate table entries

Bytes	Symbol	Meaning	
13	YTFLG	Flag byte or code byte of aggregate. See figure 5.30.	YTCDE: Code byte of aggregate.
14	YTALN	Alignment.	
15	YTHNG	Hang.	
16-19	YTROFF	Relative offset of this member of the structure.	
20-23	YTSZ	Size of the structure.	
	1...	Size adjustable.	
	.1..	Bounds specified by asterisks.	
	..1.	REFER bounds.	
24-25	YTADD	Reference of aggregate descriptor descriptor.	
26	YTSZFL	Flag byte:	YTSPA
	1...	Area.	
	.1..	Varying.	
	..1.	String.	
	...1	No-storage temporary operand.	
	.... 1...	COBOL.	
	.... .1..	FORTRAN.	
	.... ..1.	Unaligned BIT.	
	1	BIT.	
27	YTKEFL	Flag byte:	
	1...	Adjustable bounds present.	
	.1..	Not all bounds come from the same level.	
	..1.	All lower bounds = 1.	
	...1	Bounds specified by asterisks.	
	.... 1...	Used by Phase PA - storage for descriptor only, not locator.	
	.... .1..	Used by Phase KE.	
	.... ..1.	Used by Phase GE - connected non-controlled parameter.	
	.... ..1.	Used by Phase IQ - aggregate mapped, reinitialized by Phase PE.	
	.... ..1.	Used by Phase PI - locator/descriptor initialized.	
	1	Used by Phase PE - descriptor allocated storage (initialized by Phase GE).	
28-29	YTSPB	For an adjustable or unstructured array, size of descriptor. Otherwise, offset within descriptor of element entry.	YTSP
30-31	YTDEF	Dictionary reference of descriptor of aggregate on which this one is defined.	
32-35	YTRVO	Relative virtual origin.	
36-		Eight-byte descriptor for each dimension and/or adjustable string length. The format is described below.	

Figure 5.11. (Part 2 of 3). Format of general dictionary aggregate table entries

Bytes	Symbol	Meaning	
		FIXED EXTENT The multiplier is calculated during compilation, in Phase IQ.	
0-3	YTMULT	Multiplier used in calculating addresses of subscripted elements Address = relative virtual origin + $\sum_{i=1}^{n} Si*Mi$ where Si = value of ith subscript Mi = value of ith multiplier n = total number of dimensions	
4-5	YTLBD	Lower bound	YTBNDS. Collective name for bounds.
6-7	YTUBD	Upper bound	
		ADJUSTABLE EXTENT The multiplier is calculated at execution time.	
0	YTBFLG	Flag byte:	
	1...	Adjustable upper bound.	
	.1..	* upper bound.	
	..1.	REFER upper bound.	
	...1	Multiplier not known.	
	.... 1...	Adjustable lower bound.	
	.... .1..	* lower bound.	
	.... ..1.	REFER lower bound.	
	1		
1	YTBLV	Structure level from which this bound originates.	
2-3		Not used.	
4-5	YTLBD	Lower bound.	YTBNDS. Collective name for bounds.
6-7	YTUBD	Upper bound.	

Figure 5.11. (Part 3 of 3). Format of general dictionary aggregate table entries

Bytes	Symbol	Meaning
0	YCDE	Operand code byte (X'11').
1-2	YPCHN	Chain field, used for commoning purposes.
3	YPMFL	Mantissa flag, as required by the library.
4	YPEFL	Exponent flag, as required by the library.
5-6	YPOBLTH	Length of object-time representation of picture (not of datum).
7-9		DED. See figure 5.59.
10-11	YPICLTH	Picture length.
12-	YPICT	Picture.

Figure 5.12. Format of general dictionary picture table entries

Bytes	Symbol	Meaning	
0	YCDE	Operand code byte. See figure 5.31.	
1-2	YCCHN	Chain fields linking items of the same alignment class.	Pcst PA: Offset of constant in static.
3	YCFLG	Flag byte indicating whether or not the constant requires a descriptor or object-time DED.	
4	YCTYP	Type.	YCDED. Data element descriptor. See figure 5.59.
5	YCP	Precision.	
6	YCQ	Scale.	
7-8	YCLTH	Length of the constant in its present form.	
9-10	YCREP	Replication factor, if present.	
11	YCONS	Constant.	

Figure 5.13. Formats of general dictionary constant entries

Bytes	Symbol	Meaning
0	YCDE	Operand code byte (X'48').
1	YOFL	Optimization flag.
2-3	YOCHN	Chain.
4-5	YOSN	Select Statement Number.
6-9	YOWMX	Maximum WHEN expression value.
10-13	YOWMN	Minimum WHEN expression value.
14-15	YOWCC	WHEN clause count.
16-17	YOWEC	WHEN expression count.
18-19	YOWOEC	WHEN optimizable expression count.
20	YOTYP	SELECT data type.
21	YOP	SELECT precision.
22	YOQ	SELECT scale.
23	YOLEN	SELECT length.

Figure 5.13.1. Format of general dictionary SELECT optimization table

Bytes	Symbol	Meaning
0	YCDE	Operand code byte (X'0F').
1-2 3-4	YSN YLC	Statement number Block level and count. || Post PA: Offset of file in static
5-6	YNMREF	Reference of names dictionary entry.
7	YFLCD	Flag byte indicating type of constant.
8-9	YFCHN	General dictionary reference of the next entry in the file chain. The chain is headed by XFILCH in the communication area.
10-11	YFDCL	General dictionary reference of the FCB entry.
12-13	YFENV	General dictionary reference of the ENVB entry.
14-15	YFATL	Length of the following attribute list.
16-	YFATT	Attribute list, used in the construction of FCB and ENVB entries.

Figure 5.14. Format of general dictionary file constant entries

Bytes	Symbol	Meaning
0-10		As for a constant entry, with YCCHN pointing to the ENVB entry, or to the DTF entry if ENVB does not exist.
11-18	YCFST	Valid statement mask.
19-22	YCFAS	Reference of LIOCS module name in names dictionary.
23-26	YCFTM	Reference of LIOCS module name in names dictionary.
27-30	YCFNM	A (File name).
31-34	YCFEN	A (ENVB).
35-38	YCFDF	A (DTF).
39-42	YCFAF	Opened-file chain; object time only.
43-44	YCFTY	Fifth and sixth characters of transmitter.
45-46	YCFER	Error flag (X'47' if declaration error).
47-50	YCFAT	Attributes of the file.
51-58	YCFAG	Flag bytes.
59-60	YCFBK	Block size; Post Phase KM or object time.
61-62	YCFKL	Keylength-1; Post Phase KM or object time.
63-66	YCFRL	Record length; Post Phase KM or object time.
67-70	YCFREC	A (Current or hidden buffer); Post Phase KM or object time.
71-74	YCFIOA	A (Buffer space); Post Phase KM or object time.

Figure 5.15. (Part 1 of 2). Format of general dictionary FCB entries

Bytes	Symbol	Meaning
75-78	YCFIOL	L (Buffer space); Post Phase KM or object time.
79-86		Spare.
87-		The remainder of the FCB is either a STREAM I/O block (see figure 5.16) or a RECORD I/O block (see figure 5.17).

Figure 5.15. (Part 2 of 2). Format of general dictionary FCB entries

Bytes	Symbol	Meaning
87-114	YCSTRM	STREAM I/O block; unused at compile time.
115-116	YCFLS	Length of file name.
117-	YCFS	File name.

Figure 5.16. Format of FCB entry STREAM I/O block

Bytes	Symbol	Meaning
87-100	YCRECI	Part of RECORD I/O block which is unused at compile time.
101		Pointer to the DTF entry for the first file in associated chain.
102-127	YCRECI	Remainder of RECORD I/O block unused at compile time.
127-130	YCFAML	Address of LIOCS module.
131-132	YCFHSV	
	1...	READ file.
	.1..	PUNCH file.
	..1.	PRINT file.
133	YCFEGL	
	0000	File does not have ASSOCIATE option.
	1000	Function R,RP
	0100	P,RP
	0000	R,RW
	0001	W,RW
	0000	P,PW
	0001	W,PW
	1000	R,RPW
	0111	P,RPW
	1001	W,RPW
134	YCFEMT	Sixth character of error module name.
139-140	YCFLR	Length of file name.
141-	YCFR	File name.

Figure 5.17. Format of FCB entry RECORD I/O block

Bytes	Symbol	Meaning
0-10		As for a constant entry, with YCCHN pointing to the DTF entry.
11-14		Flag bytes (initially set to zero).
15 16-18		Code byte. Block size.
19 20-22		Code byte. Record length.
23 24-26		Code byte. KEYLOC value.

Figure 5.18. (Part 1 of 2). Format of general dictionary ENVB entries

Bytes	Symbol	Meaning
27 28-30		Code byte. Keylength.
31 32-34		Code byte. Index area size.
35 36-38		Code byte. Additional buffer size.

Figure 5.18. (Part 2 of 2). Format of general dictionary ENVB entries

Bytes	Symbol	Meaning	
0-10		As for a constant entry. See figure 5.13.	
11-12		NF = number of fields in the DTF which must be relocated to the DTF location in static.	
		DTF header table. One entry is made for each section of the DTF that is to be relocated. The entry can be one of two types, according to whether the section is constant or not.	
1 byte 3 bytes 4 bytes		Flag byte - X'01'. Length of a constant section of the DTF. 00 - Ignore this entry. 02 - Buffer entry-reserve this amount of space for buffers. 05 - Describes fullword field in body of DTF which is pointer to another DTF (dict. ref. in this entry).	Flag byte - X'0F'. Length and offset from the DTF origin of a DTF section that is not constant.
133+4*NF-		Skeleton DTF.	

Figure 5.19. Format of general dictionary DTF entries

Bytes	Symbol	Meaning
0	YCDE	Operand code byte:
	0001 0111	Record descriptor.
	0001 1000	Key descriptor.
1	YKL	Block level.
2	YKC	Block count.
3-4	YKCHN	Descriptor chain linking items of similar type (for commoning).
5-6	YKAUT	Offset of descriptor in AUTOMATIC storage.
7-8	YKAUTB	Base of descriptor in AUTOMATIC storage.
9-10	YKBOFF	Address vector offset in AUTOMATIC storage.
11	YKFLG	Flag byte:
	1...	The address of the 'length bytes' (see YKRLTH below) is required.
	.1..	The address of a VARYING string is required.
	..1.	The length field includes two length bytes.
	...1	The length field is the current length of the string.
	.... 1...	The descriptor will be in AUTOMATIC storage.
12-13	YKVAR	Dictionary reference of the variable corresponding to this descriptor.
14-15	YKOFF	Storage offset of this RECORD/KEY descriptor.
16-23	YKDESC	Descriptor body, containing object-time information concerning this RECORD/KEY. This descriptor may be processed by the constants analysis phase in the same way as other object-time constant information. The contents are described in the following two tables.

Figure 5.20. Format of general dictionary RECORD and KEY descriptor entries

Bytes	Symbol	Meaning
16-19	YKDAD	Address of the data to be written out (WRITE) OR Address of the area to which the data is to be read in (READ). OR Address of the area at which the buffer address is to be stored (LOCATE).
20	YKDFL	Flag byte:
	1...	INTO or FROM argument is a VARYING scalar string.
	.1..	INTO argument is a VARYING scalar bit string.
	..1.	INTO argument ends with a scalar AREA variable. This implies that the RECORD condition is not to be raised if the record is too short.
21-23	YKRLTH	Length, in bytes, of the data to be transmitted, including two 'length bytes' if it is a VARYING string. For output operations it is the current length. For input operations it is the maximum length.

Figure 5.21. Contents of a RECORD descriptor

Bytes	Symbol	Meaning
16-19	YKDAD	Address of the source key, excluding length bytes if VARYING. OR Address of where to put key, excluding length bytes if the target is varying.
20	YKDFL	Flag byte:
	1...	The KEYTO string is VARYING.
	.1..	The object-time descriptor will have an additional four bytes containing a region number.
21		Zero.
22-23	YKKLTH	Length, in bytes, of the KEY string, excluding the two length bytes if it is a VARYING string. For KEY or KEYFROM it is the current length. For KEYTO it is the maximum length, if the string is not VARYING.

Figure 5.22. Contents of a KEY descriptor

Bytes	Symbol	Meaning
0-10		As for a constant entry. See figure 5.13.
11-14	YDOATT	Bit vector indicating (compatible) attributes specified in the OPEN statement.
15-18	YDOCON	Bit vector indicating all file attributes which would be in conflict with the attributes specified above.
19-22	YDOENV	Pointer to OPEN environment options.

Figure 5.23. Format of general dictionary open control block (OCB) entries

Bytes	Symbol	Meaning	
0	YCDE	Operand code byte (X'78').	
1-256	YAWVL	Bit vector: which of the first 2048 variables have value list entries.	
257-288	YAIVLA	Bit vector: which of the first 256 variables are in value lists.	YAIVL
289-512	YAIVLB	Bit vector: which of the next 1792 variables are in value lists.	
513-544	YACONU	Bit vector: which of the first 256 variables are used in computational ON-units.	
545-576	YACONS	Bit vector: which of the first 256 variables are set in computational ON-units.	
577-608	YAIONU	Bit vector: which of the first 256 variables are used in I/O ON-units.	
609-640	YAIONS	Bit vector: which of the first 256 variables are set in I/O ON-units.	
641-896	YAGOBL	Bit vector: labels that are branched-to from on-units or from blocks called from on-units, and cause a branch out of a block.	

Figure 5.24. Single optimization entry for the whole program, in the general dictionary. It consists of eight bit vectors giving alias information for the whole program and variable usage information for ON-units. Its dictionary reference is held in XALIAS in the communication area (XCOMM).

Bytes	Symbol	Meaning
0	YCDE	Operand code byte (X'77').
1	YAFLAG	Flag byte: 1... .... Block is an ON-unit. .1.. .... Block is a computational ON-unit. ..1. .... Block is recursive.
2-3	YALEN	Total length of the last three bit vectors in this entry.
4-5	YALABL	Dictionary reference of the first label on the chain for this block.
6-7	YALABS	Number of labels in this block.
8-9	YALABL1	Ident number of first compiler-generated label in this block.
10-11	YALABS1	Number of compiler-generated labels for the block, (that can be assigned to label variables), existing at the end of the dictionary-build stage.
12-43	YABCB	Bit vector: which blocks are called from this block.
44-75	YAVUB	Bit vector: which of the first 256 variables are used in this block.
76-107	YAVSB	Bit vector: which of the first 256 variables are set in this block.
108-139	YAVUBCB	Bit vector: which of the first 256 variables are used in blocks called from this block.
140-171	YAVSBCB	Bit vector: which of the first 256 variables are set in blocks called from this block.
172-	YAXLAB	Three bit vectors, each containing a bit for every label constant in the program, rounded up to the next byte. The three bits corresponding to a label have the following meaning if on: Vector 1 bit: the label is external to this block and is branched to from within it. Vector 2 bit: the label is external to this block and labels a block called from within it. Vector 3 bit: control can be passed to this block by a go-to out of a contained block.

Figure 5.25. Optimization entries for blocks, in the general dictionary

Bytes	Symbol	Meaning
0	YCDE	Operand code byte (X'76').
1	YAFLAG	Flag byte (X'40') indicating that the owner of the list is a label variable, and that only a label value list is present.
2-3	YALEN	Length of the following bit vector.
4-	YALV	Label value list containing a bit for every label constant in the program. An on bit indicates that the owner of the list assumes the value of the label constant corresponding to the bit, at some point in the program.

Figure 5.26. (Part 1 of 4). Format of general dictionary value list entries - label variables

Bytes	Symbol	Meaning
0	YCDE	Operand code byte (X'76').
1	YAFLAG	Flag byte (X'80') indicating that the owner of the list is a parameter or DEFINED base, and that only a variables list is present.
2-3	YALEN	Length of the following bit vector.
4-35	YAP	Variables list containing a bit for each of the first 256 variables in the program. An on bit indicates that the owner of the list assumes the value of the variables corresponding to the bit, at some point in the program.

Figure 5.26. (Part 2 of 4). Format of general dictionary value list entries - parameter or DEFINED bases

Bytes	Symbol	Meaning
0	YCDE	Operand code byte (X'76').
1	YAFLAG	Flag byte (X'20') indicating that the owner of the list is an entry variable.
2-3	YALEN	Length of the following bit vector.
4-35	YAE	Block list containing a bit for each block in the program. If bit 0 is on, it indicates that the entry variable can transfer control to an external procedure. If any other bit is on, it indicates that the entry variable can transfer control to an entry point of the corresponding block.

Figure 5.26. (Part 3 of 4). Format of general dictionary value list entries - entry variables.

Bytes	Symbol	Meaning
0	YCDE	Operand code byte (X'76').
1	YAFLAG	Flag byte (X'E0)' indicating that the owner of the list is a locator, and that a label value list, a block value list, and a variables list are present.
2-3	YALEN	Total length of the following bit vectors.
4-35	YAP	Variables value list, as described in figure 5.26, Part 2.
36-67	YAPENT	Block value list, as described in figure 5.26, Part 3.
68	YAPLAB	Label value list, as described in figure 5.26, Part 1.

Figure 5.26. (Part 4 of 4). Format of general dictionary value list entries - locators

Bytes	Symbol	Meaning
0		Operand code byte (X'1F').
1		Total length of this dictionary entry.
2-		Data.

Figure 5.27. Format of general dictionary overflow entries

Bytes	Symbol	Meaning
0	YCDST	Operand code byte, as for the corresponding variables dictionary entry.
1	YLST	Block level.
2	YCST	Block count.
3	YSALN	Object-time alignment.
4	YFLGST	Flag byte.
5-6	YLOCB	Locator's base (in AUTOMATIC storage).
7-8	YLOCO	Locator's offset (in AUTOMATIC storage).
9-10	YSTB	Size of symbol table's external CSECT.
11-12	YSTO	Symbol table offset.
13-14	YDESCB	Descriptor's base.
15-16	YDESCO	Descriptor's offset.
17-20	YVOST	Relative virtual origin (AO-VO).
21-24	YELST	Offset of variable from region.
25-27	YSSIZ	Size of variable.
28-29	YSNMRF	Reference of names dictionary entry.
30-32	YDEDST	DED of variable.
33-34	YSBOFF	Offset of ADCON.
35-36	YSBSE	Base number.
37-38	YSHSK	Locator's offset (in STATIC storage).
39	YBTOFF	Bit offset.

Figure 5.28. Format of storage dictionary entries

OPERAND CODE BYTES

Operand code bytes are used to identify operand types, and an operand code byte is therefore the first byte of any six-byte reference (see "Six-byte References to Operands"). Also, all dictionary entries referenced directly in text commence with the same code byte as their corresponding six-byte references. Some entries (e.g., FCBs) are associated with operands in text only indirectly, by chaining to other dictionary entries, and have no corresponding six-byte references. In such cases, the 'operand code byte' does not classify an operand type, but only indicates a particular type of dictionary entry.

The figures included under the heading "Operand Code Bytes" are as follows:

Figure 5.29. Operand classifications
Figure 5.30. Variable operand classifications
Figure 5.31. Operand code bytes X'00' to X'0F'
Figure 5.32. Operand code bytes X'10' to X'1F'
Figure 5.33. Operand code bytes X'20' to X'2F'
Figure 5.34. Operand code bytes X'30' to X'3F'
Figure 5.35. Operand code bytes X'40' to X'4F'
Figure 5.36. Operand code bytes X'60' to X'6F'
Figure 5.37. Operand code bytes X'70' to X'7F'
Figure 5.38. Operand code bytes X'80' to X'FF'

The first hex digit of an operand code byte indicates the class of operand, as shown in figures 5.29 and 5.30. The second hex digit identifies the particular operand and/or dictionary entry type, as shown in figures 5.31 to 5.38.

First hex digit	Class of operand	Second hex digit. See figure no.
0	Constant.	5.31
1	Table or descriptor reference.	5.32
2	BIF.	5.33
3	Block header or entry point.	5.34
4		
5		
6	Temporary (six-byte reference format shown in figures 5.52 to 5.58).	5.36
7	Miscellaneous	5.37
8 to F	Variables, as described in figure 5.30.	5.38

Figure 5.29. Operand classifications

First hex digit	Type of variable	Six-byte ref. format. See figure no.	Dict. entry format. See figure no.
8	Undimensioned structure.	5.39	5.7
9	Dimensioned structure.	5.39	5.7
A	Undimensioned label, event, file, or task.	5.42 5.43	5.7
B	Dimensioned label, event, file, or task.	5.42 5.43	5.7
C	Undimensioned entry.	5.42 5.43	5.7
D	Dimensioned entry.	5.42 5.43	5.7
E	Undimensioned data.	5.40	5.7
F	Dimensioned data.	5.40	5.7

Figure 5.30. Variable operand classifications

Code (hex)	Type of operand and/or dictionary entry	Six-byte ref. format. See figure no.	Dict. entry format See figure no.
00	Source program constant.	5.41	5.13
01	Literal decimal integer constant in range 0 to ± 10**7.	5.44	
02	Literal binary integer constant in range 0 to ± 10**7.	5.44	
03	Literal character constant of length 1 to 4 bytes.	5.45	
04	Literal bit constant of length 1 to 31 bits.	5.45	
05	Literal FED constant of length 4 bytes.	5.45	
06	File control block - FCB (general dictionary, post-KL). Referred to by the file constant entry.		5.15
07	Define the file - DTF(general dictionary, post KL). Referred to by the ENVB entry.		5.19
08	Source program label constant.	5.48	5.13
09	Internal condition condition.	5.48	
0A	Generated statment label (GSL).	5.48	
0B	Local compiler label.	5.48	

Figure 5.31. (Part 1 of 2). Operand code bytes X'00' to X'0F'

Code (hex)	Type of operand and/or dictionary entry	Six-byte ref. format. See figure no.	Dict. entry format. See figure no.
0C	Compiler label.		
0D	External condition condition.	5.41	5.14
0E	Environment block - ENVB (general dictionary, post KL). Referred to by the FCB entry.		5.18
0F	File constant.	5.41	5.14

Figure 5.31. (Part 2 of 2). Operand code bytes X'00' to X'0F'

Code (hex)	Type of operand and/or dictionary entry	Six-byte ref. format. See figure no.	Dict. entry format. See figure no.
10	Adjustable descriptor reference.	5.49	
11	Picture table. Referred to by variables dictionary entry.	5.41	5.12
12	Loop data entry.		
13	Format element descriptor - FED.		
14	Data element descriptor - DED.		
15	Varying length string descriptor.		
16	Adjustable string descriptor.		
17	Record descriptor.		5.20
18	Key descriptor.		5.20
19	Aggregate descriptor.		5.11
1A	Structure offset field in an aggregate descriptor.	5.50	
1B	FED to be passed as an argument to a library routine.		
1C	DED to be passed as an argument to a library routine.		
1D	GET/PUT (no data list) argument - text only.		
1E	Aggregate descriptor descriptor.		
1F	Overflow entry.		5.27

Figure 5.32. Operand code bytes X'10' to X'1F'

Code (hex)	Type of operand	Six-byte ref. format. See figure no.	Dict. entry format. See figure no.
20	Arithmetic bif.		
21	Mathematical bif.		
22	String-handling bif.		
23	Array manipulation bif.		
24	Condition bif.		
25	Based storage bif.		
26	Multitasking bif.		
27	Miscellaneous bif.		
28	Arithmetic pseudovariable.		
29	Mathematical pseudovariable.		
2A	String-handling pseudovariable.		
2B	Array manipulation pseudovariable.		
2C	Condition pseudovariable.		
2D	Based storage pseudovariable.		
2E	Multitasking pseudovariable.		
2F	Library module.	5.47	

Figure 5.33. Operand code bytes X'20' to X'2F'

Code (hex)	Type of operand	Six-byte ref. format. See figure no.	Dict. entry format. See figure no.
30	PROCEDURE or BEGIN block header.		5.7
31	PLIDUMP, PLISORT, etc., entry point - irreducible.		5.7
32	FORTRAN entry point - irreducible.		5.7
33	COBOL entry point - irreducible.		5.7
34	External entry point - irreducible.	5.41	5.9
35	Internal entry point - irreducible.	5.41	5.9
36			
37			
38			
39			

Figure 5.34. (Part 1 of 2). Operand code bytes X'30' to X'3F'

Code (hex)	Type of operand	Six-byte ref. format. See figure no.	Dict. entry format. See figure no.
3A	FORTRAN entry point - reducible.		
3B	COBOL entry point - reducible.		
3C	External entry point - reducible function procedure.	5.41	5.9
3D	Internal entry point - reducible function procedure.	5.41	5.9
3E	Generic entry point.		
3F			

Figure 5.34. (Part 2 of 2). Operand code bytes X'30' to X'3F'

Code (hex)	Type of operand	Six-byte ref. format. See figure no.	Dict. entry format. See figure no.
40	Length of an adjustable non-varying string, or maximum length of an adjustable varying string.	5.58	
41	Controlled variable anchor.		
42	Controlled parameter anchor.		
43	Skeleton descriptor.		
44	Reference to slot in TCA.		
45	Actual origin of array.		
46			
47			
48			
49			
4A			
4B			
4C			
4D			
4E			
4F			

Figure 5.35. Operand code bytes X'40' to X'4F'

In the following table, "both" indicates that the temporary can exist both in a register and in storage.

Code (hex)	Type of temporary operand (Type-2 text only)			
60	both	local	If operand 1 or 2, the temp. is being used for the last time. If operand 3, the temp. is being created.	
61	storage			
62	both	global		
63	storage			
64	both	local	If operand 1 or 2, the temp. is alive. If operand 3, the temp. is being reset.	
65	storage			
66	both	global		
67	storage			
68	both	local	As for 60 to 63 above.	The temp. is either a global temp., dead at the end of current flow-unit, or a local controlled temp.
69	storage			
6A	both	global		
6B	storage			
6C	both	local	As for 64 to 67 above.	
6D	storage			
6E	both	global		
6F	storage			

Figure 5.36. Operand code bytes X'60' to X'6F'

Code (hex)	Operand type	Six-byte ref. format. See figure no.	Dictionary entry type	Dict. entry format. See figure no.
70				
71				
72				
73	Workspace			
74	Q-temp. (status alive)	5.51		
75	Q-temp. (status dead)	5.51		
76	VDA	5.46	Value list entry.	5.26

Figure 5.37. (Part 1 of 2). Operand code bytes X'70' to X'7F'

Code (hex)	Operand type	Six-byte ref. format. See figure no.	Dictionary entry type	Dict. entry format. See figure no.
77	Argument for lister.	5.46	Optimization entry for a block.	5.25
78	Base code for a BASED variable (QA output).	5.46	Optimization entry for the whole program.	5.24
79	Null operand.			
7A	Temp. address (QA output).	5.46		
7B	Constant address (QA output).	5.46		
7C	Variable address (QA output).	5.46		
7D	Base code (QA output).	5.46		
7E	Miscellaneous compiler-generated literal data.	5.46		
7F	Return value.	5.71	End-of-dictionary marker.	

Figure 5.37. (Part 2 of 2). Operand code bytes X'70' to X'7F'

Second hex digit	Storage characteristics of the variable
7	STATIC temporary (abnormal).
8	CONTROLLED parameter
9	CONTROLLED.
A	AUTOMATIC parameter.
B	DEFINED.
C	BASED.
D	STATIC.
E	AUTOMATIC.
F	AUTOMATIC temporary.

Figure 5.38. Operand code bytes X'80' to X'FF' (see figure 5.30)

SIX-BYTE REFERENCES TO OPERANDS

A 'six-byte reference' does not necessarily represent a single value. It may contain either literal data (e.g., a binary constant 10**7), or a dictionary reference or references, or both. After Phase GM, all operands in text are represented in this form. Six-byte references in Type 1 text are always preceded by text code byte DREF, to identify them as such.

The content of six-byte reference depends on the type of operand which it represents, and is indicated by the 'operand code byte' with which it commences (refer to previous sub-heading "Operand Code Bytes").

The different types of six-byte references, described in figures 5.39 to 5.58 are as follows:

Figure 5.39. Six-byte reference to a structure member
Figure 5.40. Six-byte reference to a data variable
Figure 5.41. Six-byte reference to a source program constant
Figure 5.42. Six-byte reference to an EVENT or TASK variable
Figure 5.43. Six-byte reference to a LABEL variable
Figure 5.44. Six-byte reference to a literal binary/decimal integer constant
Figure 5.45. Six-byte reference to a literal character/bit constant
Figure 5.46. Six-byte reference to a literal compiler-used constant
Figure 5.47. Six-byte reference to a library subroutine
Figure 5.48. Six-byte reference to a label constant
Figure 5.49. Six-byte reference to an adjustable aggregate extent
Figure 5.50. Six-byte reference to a structure offset field in an aggregate descriptor (BASED REFER structure member)
Figure 5.51. Six-byte reference to a non-string temporary operand
Figure 5.52. Six-byte reference to a non-string Q-temp.operand
Figure 5.53. Six-byte reference to an adjustable string temporary operand
Figure 5.54. Six-byte reference to a non-adjustable string temporary operand
Figure 5.55. Six-byte reference to a string Q-temp.operand (for accessing part of a string)
Figure 5.56. Six-byte reference to the maximum length of a non-adjustable string
Figure 5.57. Six-byte reference to the current length of a VARYING string
Figure 5.58. Six-byte reference to the maximum length of an adjustable string

Bytes	Symbol	Meaning
0	ZCDE	Operand code byte - refer to figure 5.29.
1	ZSEQ	Sequence number of member (1 for major structure).
2-3	ZSTR	Aggregate descriptor dictionary reference.
4-5	ZREF	Variables dictionary reference.

Figure 5.39. Six-byte reference to a structure member (refer to figure 5.68)

Bytes	Symbol	Meaning
0	ZCDE	Operand code byte - refer to figure 5.29.
1	ZDED	ZTYP - data type - refer to figure 5.59. || Base number, and offset in STATIC storage of the object-time DED/FED - used after Phase PA if this variable is passed to a library module for conversion.
2		ZP - Precision || ZLTH-contains PIC reference if PIC var.
3		ZQ - Scale
4-5	ZREF	Variable dictionary reference.

Figure 5.40. Six-byte reference to a data variable

Bytes	Symbol	Meaning
0	ZCDE	Operand code byte - refer to figure 5.31.
1	ZDED	ZTYP - Data type. (Refer to figure 5.59.)
2		ZP - Precision || ZLTH - String length || Not used.
3		ZQ - Scale.
4-5	ZREF	General dictionary reference. || Post PA - contains offset of constant in static.

Figure 5.41. Six-byte reference to a source program constant

Bytes	Symbol	Meaning
0	ZCDE	Operand code byte - refer to figure 5.29.
1	ZTYP	Flag byte indicating EVENT or TASK variable.
2-3		
4-5	ZREF	Variable dictionary reference.

Figure 5.42. Six-byte reference to an EVENT or TASK variable

Bytes	Symbol	Meaning
0	ZCDE	Operand code byte - refer to figure 5.29.
1		Flag byte indicating LABEL.
2-3	ZLBLST	General dictionary reference of value list entry, or zero.
4-5	ZREF	Variable dictionary entry. || Post PA - Contains offset in static, if required.

Figure 5.43. Six-byte reference to a LABEL variable

Bytes	Symbol	Meaning
0	ZCDE	Operand code byte - refer to figure 5.31.
1	ZCPL	Precision of original constant.
2-5	ZCONS	Binary representation of constant, right aligned.
Post-PA		
1-3		True compile-time DED of constant as allocated in storage.
4-5	ZREF	Contains offset in static, if required.

Figure 5.44. Six-byte reference to a literal binary/decimal integer constant

Bytes	Symbol	Meaning
0	ZCDE	Operand code byte - refer to figure 5.31.
1	ZCPL	Length of string.
2-5	ZCONS	Left-aligned string.
Post-PA		
1-3		True compile-time DED of constant as allocated in storage.
4-5	ZREF	Contains offset in static, if required.

Figure 5.45. Six-byte reference to a literal character/bit constant

Bytes	Meaning
0	Operand code byte (X'7E').
1	Not used.
2-5	Literal value or values - refer to the relevant text table for exact usage.

Figure 5.46. Six-byte reference to a literal compiler-generated constant

Bytes	Meaning
0	Operand code byte (X'2F').
1	Not used.
2-3	Entry-point number field.
4-5	Offset field.

Figure 5.47. Six-byte reference to a library subroutine

Bytes	Meaning
0	Operand code byte - refer to figure 5.31.
1	Branch mask, if conditional GOTO or BC.
2-3	Not used.
4-5	Label constant dictionary reference (general dictionary).

Figure 5.48. Six-byte reference to a label constant

Bytes	Meaning
0	Operand code byte (X'10').
1	Not used.
2-3	Offset in the aggregate descriptor at which the result of the extent expression is to be stored.
4-5	Aggregate table dictionary reference. (Refer to figure 5.11.)

Figure 5.49. Six-byte reference to an adjustable aggregate extent

Bytes	Meaning
0	Operand code byte (X'1A').
1	Not used.
2-3	Variables dictionary reference of the major structure.
4-5	Temporary descriptor number, i.e., RESDES number. (Refer to figure 5.11.)

Figure 5.50. Six-byte reference to a structure offset field in an aggregate descriptor (BASED/REFER structure member)

Bytes	Meaning
0	Operand code byte - refer to figure 5.36.
1-3	Data element descriptor - refer to figure 5.59.
4-5	Temporary-number.

Figure 5.51. Six-byte reference to a non-string temporary operand

Bytes	Meaning
0	Operand code byte - refer to figure 5.37.
1-3	Data element descriptor.
4-5	Q-temp.number.

Figure 5.52. Six-byte reference to a non-string Q-temp.operand

Bytes	Meaning
0	Operand code byte (X'60').
1	Data byte indicating adjustable string data - refer to figure 5.59.
2-3	Reference of a FIXED BIN (15,0) temporary holding the string length.
4-5	Reference of a FIXED BIN (31,0) temporary holding the string's address.

Figure 5.53. Six-byte reference to an adjustable string temporary operand

Bytes	Meaning
0	Operand code byte: X'60' if an address temporary is used to access the string, OR X'61' if the string is accessed directly (see below).
1	Data byte indicating non-adjustable string data - refer to figure 5.59.
2-3	Literal string-length.
4-5	Reference of a FIXED BIN (31,0) temporary holding the string's address, OR reference of a fixed-length string temporary.

Figure 5.54. Six-byte reference to a non-adjustable string temporary operand

Bytes	Meaning
0	Operand code byte (X'74').
1	Data byte indicating string data.
2-3	Literal string-length, or undefined.
4-5	Q-temp.number.

Figure 5.55. Six-byte reference to a string Q-temp.operand (for accessing part of a string)

Bytes	Meaning
0	Operand code byte (X'02'), indicating literal binary data.
1	Data byte (X'08'), indicating bit-string data.
2-3	Zero.
4-5	Length of string.

Figure 5.56. Six-byte reference to the maximum length of a non-adjustable string

Bytes	Meaning
0	Operand code byte of the string variable.
1-3	Data element descriptor (FIXED BIN (15,0)) - refer to figure 5.59.
4-5	Dictionary reference of the string variable.

Figure 5.57. Six-byte reference to the current length of a VARYING string

Bytes	Meaning
0	Operand code byte (X'40').
1	Data byte of the string variable - refer to figure 5.59
2	Operand code byte of the string variable.
3	Not used.
4-5	Dictionary reference of the string variable.

Figure 5.58. Six-byte reference to the maximum length of an adjustable string

COMPILE-TIME DATA ELEMENT DESCRIPTORS (DEDS)

Bytes	Symbol	Meaning
		PROGRAM CONTROL DATA
0	ZTYP (YDED or YCDED in dict.)	
	0000 1101	Label
	0000 1001	Event.
	0000 1111	File.
	0000 1011	Task.
	0000 1.10	Area (fixed).
	0100 1.10	Area (adjustable).
1-2		Not used.
		PROBLEM-DATA LOCATOR
0	ZTYP (YDED or YCDED in dict.)	
	1.0.	Aligned.
	1.1.	Unaligned.
	11.0 1100	Pointer.
	11.1 1100	Offset.
1-2		Not used.
		PROBLEM-DATA ARITHMETIC
0	ZTYP (YDED or YCDED in dict.)	
	1.0.	Aligned.
	1.1.	Unaligned.
	1... .0..	Decimal.
	1... .1..	Binary.
	1... ..0.	Fixed.
	1... ..1.	Float.
	10.0 1..0	Real arithmetic.
	11.0 1..1	Real part of a complex variable.
	11.1 1..1	Imaginary part of a complex variable.
	10.0 00.0	Numeric picture.
1	ZP	Precision.
2	ZQ	Scale.
		PROBLEM-DATA STRING
0	ZTYP (YDED or YCDED in dict.)	
	0.0. ...0	Aligned.
	0.1. ...0	Unaligned.
	00.. 1.00	Fixed.
	01.. 1.00	Adjustable.
	0..0 1.00	Non-varying.
	0..1 1.00	Varying.
	0... 1000	Bit string.
	0... 1100	Character string.
	00.0 0100	Alphameric picture.
1-2	ZLTH	Length of string.

Figure 5.59. Contents of compile-time data element descriptors

TYPE-1 TEXT FORMATS

This sub-section contains tables of the text code bytes used in Type-1 text, and format descriptions of the following data areas, in the figures listed:

Main Text Stream, on Output from Phase EA

Figure 5.60. General format of statements in Type-1 text, output from Phases EA to GE
Figure 5.61. Block chaining fields

Main Text Stream, on Output from Phase EE

Figure 5.62. PROC, ENTRY, BEGIN, and ONB statements, in deblocked position
Figure 5.63. CALL statement, used to replace BEGIN statement in inline position
Figure 5.64. Statement body format for ON statements

Main Text Stream, on Output from Phase GM

Figure 5.65. PROC, BEGIN, and ONB statements
Figure 5.66. RETURN statements
Figure 5.67. Array operands in Type-1 text
Figure 5.68. Structure operands in Type-1 text
Figure 5.69. Subroutine calls and function references in Type-1 text
Figure 5.70. Argument lists in Type-1 text

Dictionary Text Stream, on Output from Phase EE

Figure 5.71. Block headers
Figure 5.72. Statements in the dictionary text stream

Declaration Expressions Text Stream, on Output from Phase GE

Figure 5.73. Page sub-headers
Figure 5.74. Block headers
Figure 5.75. Locator qualifier statements
Figure 5.76. POSITION attribute statements
Figure 5.77. 'Simple defined' items, with base known at compile-time
Figure 5.78. 'Simple defined' items, with base known at prologue execution
Figure 5.79. 'Simple defined' items, with base not known until reference at execution time
Figure 5.80. iSUB-defined items
Figure 5.81. INITIAL assignment expressions
Figure 5.82. Adjustable-extent expressions
Figure 5.83. Adjustable string-length expressions for non-aggregate strings

Declaration Expressions Text Stream, on Output from Phase GM

Figure 5.84. INITIAL assignment expressions
Figure 5.85. MAP statements
Figure 5.86. Adjustable-extent expressions
Figure 5.87. Adjustable string-length expressions for non-aggregate strings

Text Code Bytes in Type-1 Text

Figure 5.88. Text code bytes in Type-1 text (X'00' to X'7F')
(X'80' to X'FF')
(X'D900' to X'D97F')
(X'D980' to X'D9FF')

Main Text Stream, on Output from Phase EA

The text is written in source program order, with a 17-byte block chaining field inserted before each block-heading (i.e., PROCEDURE BEGIN, or ON). Termination of a chain is indicated by X'FF' in the third byte of the text reference. For all on-units, the ON statement rather than the associated BEGIN statement (if present) is treated as the block-heading statement, at this stage.

Bytes	Symbol	Meaning	
0	ZSLN	SL if labeled, SN if not.	Standard statement header.
1	ZSTMT	Text code byte indicating statement type (refer to figure 5.88).	(This is later referred to as the standard header.)
2-3	ZSNO	Source statement number.	
4	ZPOPT1	Prefix options. 0 indicates conditions enabled for the statement or block.	
	X...	CHECK with/out list on/in block	
	.X..	ZERODIVIDE.	
	..X.	FIXEDOVERFLOW.	
	...X	SIZE.	
	.... X...	CONVERSION.	
	.... .X..	OVERFLOW.	
	.... ..X.	UNDERFLOW.	
	X	STRINGSIZE.	
5	ZPOPT2	Prefix Options. 0 indicates conditions enabled for the statement or block.	
	X...	STRINGRANGE	
	.X..	SUBSCRIPTRANGE	
	..X.	CHECK(without list) enabled for stmt.	
	...X	No CHECK or NOCHECK prefix on block.	
		Other features. 1 indicates feature exists. (Used by Phases ID and KE.)	
	.... X...	The block contains statements with prefix options differing from those applying to the rest of the block.	
	.... .X..	Defined arrays in the statement.	
	.... ..X.	Subscripts in the statement.	
	X	DO-loops in the statement.	

Figure 5.60. General format of statement header in Type-1 text, output frcm Phase EA to GE

Bytes	Meaning
0	Block level (level of nesting).
1	Block count (sequence of block in source program).
2-6	Text reference of the END statement for this block.
7-11	Text reference of the next block (i.e., the next chain field).
12-16	Text reference of the first ENTRY statement in this block, if any.

Figure 5.61. Block chaining fields

Main Text Stream, on Output from Phase EE

The text is written in deblocked order, with certain statements removed and written onto a separate text stream (the dictionary text stream). The general format of statements remains unchanged, with the following three exceptions:

Bytes	Symbol	Meaning
0-5		Standard header (SL,PROC,ENTRY,BEGIN or ONB) - refer to figure 5.60.
6	ZL	Block level.
7	ZC	Block count.
8-	ZBODY	Label List (refer to figure 5.60), Semicolon. Each statement of this type exists also in the dictionary text stream, where it contains the statement body (if any).

Figure 5.62. PROC, ENTRY, BEGIN, and ONB statements, in deblocked position

Bytes	Symbol	Meaning
0-5		Standard header (SN or SL, CALL) - refer to figure 5.60.
6-	ZBODYN	Label List, if SL statement. Statement Body, containing only one identifier, namely a generated statement label (GSL) labeling the associated BEGIN block heading statement (refer to figure 5.62). The GSL consists of X'FF' followed by a two-byte number equal to the block count. Semicolon.

Figure 5.63. CALL statement, used to replace BEGIN statement in inline position

Bytes	Meaning
0-1	Text code byte indicating type of ON condition.
2	Text code byte (SNAP/NOSNAP).
3	If an on-unit has been specified for this condition then this byte is zero. Otherwise it contains text code byte SYSTEM or NULL, and the follcwing four bytes are not used.
4	Text code byte (ONB).
5-7	GSL labeling the associated BEGIN block heading statement (refer to figure 5.63).
8	Filename, CHECK list, or nothing. Semicolon.

Figure 5.64. Statement body format for ON statements

Main Text Stream, on Output from Phase GM

Phase GM processes the text stream sequentially, modifying each statement as follows:

- In the label list, each 'length and name' (refer to figure 5.61.) is replaced by its general dictionary reference. This applies also to GSLs.
- In the statement body, each operand is altered from 'length and name' to a six-byte reference preceded by text code byte DREF to identify it as such. In the cases of array operands (figure 5.67.), structure operands (figure 5.68.), and subroutine calls and function references (figure 5.69.), additional information is held in text, in order to economize on dictionary calls.

Otherwise, the statement formats, with the exceptions of PROC, BEGIN, ONB, and RETURN statements (as shown in figures 5.65. and 5.66.), remain unchanged.

The three-byte data element descriptors (DEDs) contained in some six-byte references and dictionary entries, contain a 'data byte' indicating the data element type, and additional information for certain types of data, as shown in figure 5.59.

Bytes	Symbol	Meaning
0-5		Standard header (SL, PROC, BEGIN, or ONB) - refer to figure 5.61.
6-7	ZLC	Block level and count.
8-9	XBH	Dictionary reference of block header (general dictionary).
10-11	ZPCL	GSL (compiler prologue label).
12-		Statement body (if any). Semicolon.

Figure 5.65. PROC,BEGIN, and ONB statements

Bytes	Symbol	Meaning	
0-5		Standard header (SN, RETURN) - refer to figure 5.61.	
6		Text code byte (DREF).	
7		Block level.	Six-byte reference to a returned value.
8-10		Data element descriptor (DED) for the returned value - refer to figure 5.59.	
11-12		Dictionary reference of block header (general dictionary).	
13-		Statement body. Semicolon.	

Figure 5.66. RETURN statements

The following tables show the formats of operators for which additional information is held in Type-1 text.

Bytes	Meaning
0	Text code byte (DREF).
1-6	Six-byte reference to data variable (refer to figure 5.40).
7	Text code byte (ATRM), indicating that an ATR follows.
8-9	Aggregate table reference (general dictionary).

Figure 5.67. Array operands in Type-1 text

Bytes	Meaning	
0	Text code byte (DREF).	
1-6	Six-byte reference to structure member (refer to figure 5.39).	
7	Text code byte (STDMK) to mark beginning of structure descriptor.	
8	Code byte indicating type of variable (refer to figures 5.29. and 5.38).	This is repeated for each base element, and the total constitutes the structure descriptor, length N. (N = 5*number of base elements.)
9	Sequence number.	
10-12	DED for first base element (refer to figure 5.59) .	
N+8	Text code byte (ESTMK) to mark end of structure descriptor.	

Figure 5.68. Structure operands in Type-1 text

Bytes	Meaning
0-5	Standard header (SN or SL, CALL) - refer to figure 5.60.
6	Text code byte (DREF).
7-12	Six-byte reference to function or subroutine.
13	Text code byte (ANO), indicating that the next two bytes contain the number of arguments or argument-expressions in the argument list.
14-15	Number of items in argument list (refer to figure 5.70).

Figure 5.69. Subroutine calls and functions in Type-1 text

The argument list is treated normally but, before each argument or argument expression, a reference to the corresponding parameter descriptor entry (general dictionary) is inserted, unless this statement calls an EXTERNAL procedure for which no ENTRY declaration has been made. Each parameter descriptor reference is preceded by the text code byte PARD to indicate its presence.

Argument list item	Comments
Text code byte (PARD). Text code byte (DREF). Six-byte reference to parameter descriptor. Additional information if aggregate (refer to figures 5.67 and 5.68).	Present before each argument or argument expression if information is available.
Text code byte (PARD). Six-byte reference to argument operand. Additional information if aggregate (refer to figures 5.67 and 5.68).	
Operator. Text code byte (DREF). Six-byte reference to argument operand. Additional information if aggregate (refer to figures 5.67 and 5.68).	Repeated for each further operand in the argument expression.
as above.	Repeated for each further argument or expression in the list.
);	

Figure 5.70. Argument lists in Type-1 text

Dictionary Text Stream, On Output From Phase EE

Bytes	Meaning
0	Text code byte (PROC, BEGIN, or ONB).
1-2	Block level and count.
3-7	Text reference of next block header.
8-12	Text reference of last DTF statement in this block.
13-17	Text reference of last item (PROC, BEGIN, or ONB) on the entry-point chain for this block.
18-22	Text reference of last DCL statement in this block.
23-27	Text reference of last ALLOCATE statement in this block.

Figure 5.71. Block headers

Bytes	Symbol	Meaning			
0-5		Standard header - refer to figure 5.60.			
6-7	ZLC	Block level and count.			
8-12	ZPBCHN	Text reference of the next item on the entry point, DCL, DFT, or ALLOC chain for this block. The chains are back-pointing. Each entry-point chain is headed by a PROC, BEGIN, or ONB statement. The remaining items in entry chains are ENTRY statements.			
		PROC or ENTRY	BEGIN or ONB	DCL or DFT	ALLCCATE
13-	ZPBLBL	Label list containing all entry names for this statement. Statement body. Semicolon.	Label list containing the GSL only (refer to figure 5.63) Semicolon.	Statement body. Semicolon.	Label list Statement body. Semicolcn.

Figure 5.72. Statements in the dictionary text stream

Declaration Expressions Text Stream, on Output from Phase GE

This is a second Type-1 text stream, on separate text pages, containing the following declaration expression types:

- Array bounds and adjustable string length expressions.
- INITIAL assignments.
- Locator qualifier expressions.
- DEFINED item expressions.

Bytes	Symbol	Meaning	
0	DPHCDE	Text code byte (SN2).	
1-3	DPHPREV	Track address of previous page.	Chain field.
4-5	DPHCHN	Offset in previous page.	

Figure 5.73. Page sub-headers

Bytes	Symbol	Meaning
0	DBHCDE	Text code byte (SN2).
1		Text code byte (PROC/BEGIN).
2-3	DBHLC	Block level and count.
4		Semicolon.

Figure 5.74. Block headers

Bytes	Symbol	Meaning
0	DBASE	Text code byte (SN2).
1		Text code byte (PTS/OFFA).
2-3	DBASDR	Dictionary reference of base (PTS statement), or AREA to which the OFFSET attribute applies (OFFA statement).
4-		Qualifier expression. Semicolon.

Figure 5.75. Locator qualifier statements

Bytes	Symbol	Meaning
0	POSDEST	Text code byte (SN2)
1		Text code byte (POSX).
2-3	POSDR	Dictionary reference of the DEFINED item.
4-		POSITION expression. Semicolon.

Figure 5.76. POSITION attribute statements

Bytes	Symbol	Meaning
0	DM1	Text code byte (SN2).
1	DM2	Text code byte (X'52').
2-3	DMAPDR	Dictionary reference of DEFINED item.
4-		Expression. Semicolon.

Figure 5.77. 'Simple defined' items, with base known at compile time

Bytes	Symbol	Meaning
0	DEDASN	Text code byte (SN2).
1		Text code byte (LDSAN).
2-3	DASNDR	Dictionary reference of the DEFINED item.
4-		Expression. Semicolon.

Figure 5.78. 'Simple defined' items, with base known at prologue execution

Bytes	Symbol	Meaning
0	DEDPTS	Text code byte (SN2).
1		Text code byte (PTSD).
2-3	DPTSDR	Dictionary reference of the DEFINED item.
4-		Expression. Semicolon.

Figure 5.79. "Simple defined" item, with base not known until reference at execution time

Bytes	Symbol	Meaning
0	DEDSUBS	Text code byte (SN2).
1		Text code byte (DSUBS).
2-3	DSUBSDR	Dictionary reference of the DEFINED item.
4-		Expression. Semicolon.

Figure 5.80. iSUB-defined items

Bytes	Symbol	Meaning
0	DINCD	Text code byte (SN2).
1		Text code byte (INASSN).
2-3	DINVAR	Dictionary reference of the variable.
4		Text code byte (ASSN/CALL).
5-		Expression. Semicolon.

Figure 5.81. INITIAL assignment expressions

Bytes	Symbol	Meaning	
0	DECDE	Text code byte (SN2).	
1	DETYPE	Text code byte (AGGASSN).	
2		Text code byte (DREF).	
3	DEDV	Dictionary code byte (X'10') indicating that this is a reference to an aggregate table offset.	Six-byte reference to an adjustable extent.
4			
5-6	DEOFFS	Offset in object-time aggregate descriptor at which the result of the expression is to be stored.	
7-8	DEAGRF	Dictionary reference of the aggregate table for this variable.	
9		Text code byte (ASSN) indicating that the result of the following expression is to be assigned to the field indicated by the above six-byte reference.	
10-11	DECHN	Chain field, used for commoning purposes.	
12-13	DEXPNO	Expression number.	
14-15	DEXPL	Expression length.	
16-		Expression. Semicolon.	

Figure 5.82. Adjustable extent expressions

Bytes	Symbol	Meaning
0	DESTRCD	Text code byte (SN2).
1	DESTRL	Text code byte (STRL).
2-3	DESTRVR	Dictionary reference of the variables dictionary entry.
4	DESTRASN	Text code byte (ASSN).
5-		Expression. Semicolon.

Figure 5.83. Adjustable string-length expressions for non-aggregate strings

Bytes	Meaning	
0	Text code byte (SN2), indicating that this is a 'second file' statement.	
1	Text code byte (INASSN).	
2-6	Five-byte chain field to be used by Phase IA.	
7	Text code byte (DREF).	
8-13	Six-byte reference to the variable.	
14	Text code byte (ATRM).	Not used if the variable is not in an aggregate.
15-16	Aggregate table reference.	
17	Text code byte (ASSN-CALL).	
18-	Expression. Semicolon.	

Figure 5.84. INITIAL assignment expressions

Bytes	Meaning	
0	Text code byte (SN2).	
1	Text code byte (X'52').	
2-6	Five-byte chain field to be used by Phase IA.	
7	Text code byte (DREF).	
8	Operand code byte (X'10').	Six-byte reference.
9-10	Dictionary reference of next item on this. alignment chain.	
11	Alignment of the aggregate.	
12-13	Dictionary reference of the last aggregate mapped.	
14	Semicolon.	

Figure 5.85. MAP statements

Bytes	Meaning	
0-9	As described in figure 5.82.	
10-	Expression. Semicolon.	Six-byte reference to another aggregate extent to enable the extent referenced in bytes 3-8 to be inherited. Expression. Semicolon.

Figure 5.86. Adjustable extent expressions

Bytes	Meaning
0	Text code byte (SN2).
1	Text code byte (STRL).
2	Text code byte (DREF).
3-8	Six-byte reference to the string variable.
9	Text code byte (ASSN).
10	Expression. Semicolon.

Figure 5.87. Adjustable string-length expressions for non-aggregate strings

Text Code Bytes In Type-1 Text

First hex digit								Second hex digit
0	1	2	3	4	5	6	7	
0	G	W	COMMA			CONCAT	ALLOC	0
1	H	X	QUOTE	FMTCOL		DIVIDE	FREE	1
2	I	Y	POINT	FMTE	MAP	MULT	WAIT	2
3	J	Z	COLON	FMTF	DOEQ	PLUS	DELY	3
4	K	@	SCOLON	FMTC		MINUS	EXIT	4
5	L	$	PCENT	FMTA		POWER	STOP	5
6	M	ARK	SLPASN	FMTB	MASSN	NOT	DSPLY	6
7	N	SPECIAL	RPAR	FMTP	ASSN	PPLUS	SIGNAL	7
8	O	LOQ	SCOLON2	FMTX	GE	PMINUS	REVERT	8
9	P	ROQ	ATRM	FMTR	LT	LPAR	NULL	9
A	Q	CAH1	STDMK	FMTPGE	EQ	TO	SASSN	A
B	R	CAH2	ESTMK	FMTSKP	NE	BY	SBASSN	B
C	S	RPAR2	ANO	FMTLIN	LE	THEN	SOASSN	C
D	T	JUMP	PARD	SELECT	GT	WHILE *	RETURN	D
E	U		PCASSN	SWHEN	AND	DOWHYL	GOTO	E
F	V	SCLN3	OFFA	FILE	OR	ELSE	GOOB	F

* also WHYL

Figure 5.88. (Part 1 of 4). Text code bytes in Type-1 text (X'00' to X'7F')

First hex digit								Second hex digit
8	9	A	B	C	D	E	F	
CALL	PROC	CHCKC	NOCHK	SKIP	CV	AGGASSN	BCF	0
IF	ENTRY	FETCH	ATTR	LINE	SYST	INASSN	LDASN	1
FORMAT	BEGIN	RLEASE	PGSIZE	PAGEO	SNAP	STRL	POSX	2
READ	DO		TITLE	COPY	NOSNAP	ENDL	SINT	3
WRITE	ITDO		LNSIZE	TASKO	ISUB	SDO		4
UNLOCK	END		INTO	EVENTO	ARRX	SEXP	ENDSEL	5
DELETE	ENDDO		FROM	PRIOR	PSV		SOTHER	6
REWRITE	ENDIT		SET	REPLY	FNCT	SSPLUS		7
LOCATE	ENDPRG	UNTIL	KEYO	FMTLST	SBAR	SPMIN		8
GET	ENDPGE	GARG	NOLOCK	IN	SLVL	SLPAR	PTSF	9
PUT	FLWUNT	CFORM	IGNORE	KEYFM	PTS	STO		A
OPEN	ONB		FILO	KEYTO	DSUBS	SBY	SN2	B
CLOSE			LIST	SETA		STHEN	SN	C
ONS		LEAVE	EDIT	BIFZ		DOREPT	SL	D
DCL	ENTMP		DATA		DREF	EEMK	GSN	E
SPOSX	DOTMP		STRING		ERRORM	SELSE	GSL	F

Figure 5.88. (Part 2 of 4). Text code bytes in Type-1 text (X'80 ' to X'FF')

First hex digit								Second hex digit
0	1	2	3	4	5	6	7	
LIT		DEC	IRRED	BASED	BUFF	MAIN		0
FL DC IM		BIN	RED	REFER	UNBUFF	REENTRNT		1
FL DC RL		FLOAT	RECUR	DRFISUB	EXCL	SECONDRY		2
FL BN JM		FIXED	VARBLE	INITVAR	HEYED	ORDER		3
FL BN RL		REAL	CONDA	INIT	STREAM	REORDER		4
FX DC IM		COMPLEX		LIKE	RECORD	COBOL		5
FX DC RL		PREC(1)		DEF	BACK	FORTRAN		6
FX BN IM		PREC(2)	RETURNS	UNAL	SEQ			7
FX BN RL		VARYING	ENTRY	AL	DIRECT	ALL		8
INTEG		PICT NUM	GENERIC	PTR	PRINT	RANGE		9
iSUB		PICT CHAR	BUILT-IN	OFFSET	ENV	VALUE		A
CHARC		CHAR	EXT	AREA	INPUT	DESCRIP		B
BITC		BIT	INT	TASK	OUTPUT		CSP1	C
		DIMS	AUTO	EVENT	UPDATE		CSP2	D
		LABEL	STATIC	POS			CSP3	E
		OPTIONS	CTL	FILE			CSP4	F

Figure 5.88. (Part 3 of 4). Text code bytes in Type-1 text (X'D900' to X'D97F')

First hex digit								Second hex digit
8	9	A	B	C	D	E	F	
OFL	NAME	NOZDIV						0
UFL	ENDFILE	NOFOFL						1
ZDIV	RECORDC	NOSUBRG						2
FOFL	KEYC	NOSTRG						3
SUBRG	PAGEC	NOSIZE						4
STRG	CSP7	NOCONV						5
SIZE	CSP8	CSP15						6
CONV	CSP9	CSP16						7
ERROR	CSP10	CSP17						8
FINISH	CSP11	CSP18						9
AREAC	CSP12	NOCHCK						A
CSP5	CSP13							B
CSP6	CSP14							C
COND	CHECKC							D
TRANSMIT	NOOFL							E
UNDEF	NOUFL							F

Figure 5.88. (Part 4 of 4). Text code bytes in Type-1 text (X'D980' TO X'D9FF')

TYPE-2 TEXT FORMATS

The Type-2 text stream is created by Phase II, which inserts overflow page index tables, and breaks down each Type-1 text statement into a statement header table and a number of general usage tables. These general usage tables are referred to as Type-2 text tables, the others in the Type-2 text stream being referred to by their specific names, e.g., overflow page index tables, etc. Tables may be inserted into and deleted from text throughout Type-2 text processing. The following general types of table may exist in Type-2 text:

- Overflow page index tables - Occupy the first 32 bytes of the processable-data region in each page (see figure 5.2), throughout Type-2 text processing.
- Statement header tables - One table is generated by Phase II for each Type-1 text statement found. See figure 5.90.
- Type-2 text tables - Generated and deleted during Type-2 text processing (Phase II to SD). The format of Type-2 text tables is shown in figure 5.92. The different types are shown in figure 5.96.
- Flow-unit header tables - Inserted into text by Phase OE, for optimization purposes. See figure 5.98.
- Hash tables - Inserted into text by Phase OE, for optimization purposes. See figure 5.100.

The following figures are included under the heading "Type-2 Text":

First hex digit								Second hex digit
0	1	2	3	4	5	6	7	
		CGSR		ASSNEX	DO3	CONCAT	ALLOC	0
		LLAD	RESDES		PSV2	DIVIDE	FREE	1
		VDA			MAP	MULT	WAIT	2
		SHIFT				PLUS	DELAY	3
		LADDR			DO1	MINUS	EXIT	4
					DO2	POWER	STOP	5
			OFFS		MASSN	NOT	DSPLAY	6
		COMP			ASSN	PPLUS	SIGNAL	7
		BGE			GE	PMINUS	REVERT	8
		BLT			LT		NULL	9
		BEQ	GEN		EQ	DINC		A
		BNEQ			NE	SCI		B
		BLE	CONV		LE			C
		BGT	NDX		GT	WHILE	RETURN	D
DROSS		BCB			AND	DOWHYL	GOTO	E
MGT	GHOST	BC	BADDR		OR		GOOB	F

Figure 5.89. (Part 1 of 2). Operator code bytes (IOP1) in Type-2 text (X'00' to X'7F')

First hex digit								Second hex digit
8	9	A	B	C	D	E	F	
CALL	PROC	CHECKC	NOCHK				SUBS1	0
IF	ENTRY						SUBS	1
	BEGIN					DEL	DEL	2
READ		CHANE					SINIT	3
WRITE	ITDO	TRT					ALIST	4
UNLOCK		ONCAD		EVENTO				5
DELETE		ALOBIF			PSV			6
REWRITE	ENDIT	ACCUM		REPLY	FNCT		BIF	7
LOCATE	ENDPRG			FMTLST			KONST	8
GET	ENDPGE						PTSAT	9
PUT	FLWUNT				PTS		PINIT	A
OPEN1	ONB	FITE			DSUBS		GSL2	B
CLOSE	STPG	FIT			ARG		SN	C
ONS	HASH	AID			DLIST		SL	D
	MOVE	ENDAID	DATAE	FORME	DSUBS1		GSN	E
	PEND	IASSN		EVO	DSUBSL		GSL	F

Figure 5.89. (Part 2 of 2). Operator code bytes (IOP1) in Type-2 text (X'80' tc 'FF')

Bytes	Symbol	Meaning
0	ISLN	Text code byte (SN or SL).
1	ISTMT	Text code byte, indicating statement type (as for Type-1 text, see figure 5.89).
2-3	IPOPT	Prefix options on the statement. (As for ZPOPT in Type-1 text, see figure 5.60.).
4-5	ILABL	General dictionary reference of the statement label.
6-10	ITCHN	Text reference: first statement-type chain field.
11-15	ITCHN2	Text reference: second statement-type chain field.
16-17	ILC	Block level and count.

Figure 5.90. (Part 1 of 2). Statement header tables in Type-2 text

Bytes	Symbol	Meaning
18-19	IFUNO	Flow-unit number.
20-21	ISNO	Statement number.
22	ISF	Flag byte: 1... .... Phase OE is to create a new flow unit. .1.. .... No epilogue is required for this END statement. ..1. .... The GSL appears at the end of a do-loop. ...1 .... The GSL represents an entry point. .... 1... The GSL delimits a 'black box' EDIT I/O. .... .1.. The GSL indicates an abnormal termination point, for example termination of stream I/O statements.
23-27	ITCHN3	Text reference: DO statement chain field/SELECT level and count.
28-29		Not used.
30-31	IFCHN	Forward chain field, as for the Type-2 text table.

Figure 5.90. (Part 2 of 2). Statement header tables in Type-2 text

Bytes	Symbol	Meaning
0	IOP1	Text code byte - PROC (X'90')/ENTRY (X'91').
1	IOP2	Second byte of operator (IOP2) - refer to PROC/ENTRY in figure 5.96.
2-3	IPLAB2	Apparent entry label.
4-5	IBH	Dictionary reference of block header entry.
6-7	IEH	Dictionary reference of entry-point entry.
8-11	IDSASZ	Size of variable part of dynamic storage area (DSA).
12-13	IPLAB1	GSL (compiler prologue label) - real entry label.
14-15	IPBSE	Procedure box number.
16-17	ILC	Block level and count.
18-19	HSKSZ	Size of housekeeping storage.
20-21		Not used.
22-23	IONOFF	Offset of first ON cell.
24-25	IST2	See figure 5.93.
26-27	IONND	Number of cells.
28-29	IPAMS	Size of the parameter area.
30-31		Not used.

Figure 5.91. Format of PROC/BEGIN/ONB/ENTRY tables in Type-2 text

Bytes	Symbol	Meaning	
0	IOP1	Text code byte indicating type of table. See figure 5.89.	Operator.
1	IOP2	Second byte of operator, subdividing within type. See the particular table (figure 5.96) for usage. IOP2 may be overlaid by IGEN2 (see figure 5.94) during optimization.	
2-7	IOPND1	Operand 1.	Usage of these operand fields depends on the text table type, which is indicated by the operator. Details of usage are shown in figure 5.96.
8-13	IOPND2	Operand 2.	
14-19	IOPND3	Operand 3.	
20-21	ISTNO	Statement number.	
22-31		General area, usage of which depends on the stage of compilation (see figure 5.93).	

Figure 5.92. Format of Type-2 text tables

	Statement processing and optimization		Storage allocation		Register assignment		
Bytes	Symbol	Meaning	Symbol	Meaning	Symbol	Bits	Meaning
22	IHCHN	Hash chain field (used during optimization).	IBA1	Base number for operand 1.	INDXA	0-3	
						4-7	Index register number for operand 1, if required.
23					INDXB	0-3	Index register number for operand 2, if required.
						4-7	Index register number for operand 3, if required.
24	IST2	Status flag byte, set during statement processing, and used in later stages for selecting work-register independent instructions from skeletons. IST2 is set according to the type of table and its usage of operands. See figure 5.96.					
25	IFLAG1	The IFLAG fields relate to the corresponding operands, and are used to enable processing of operands by common routines, where possible. Their bits are used as follows:					
	1...	A DED is required for this operand at execution time.					
	.1..	This operand does not reference the constants pool.					
	..1.	Conversion is required for this operand.					
	...1	Symbol table required (ARG tables only).					
	.... 1...	Locator required.					
	.... .1..	No conversion required for constant.					
	.... ..1.	No locator required.					
	1	Float conversion required for arithmetic constant.					
	1	Static ONCB required (ONS tables only).					
	1	FETCH control block required (ARG tables only).					
		IFLAG1 may be overlaid by IGEN27 (see figure 5.95) during register assignment.					

Figure 5.93. (Part 1 of 2). Format and usage of the general area of Type-2 text tables

	Statement processing and optimization		Storage allocation		Register assignment		
Bytes	Symbol	Meaning	Symbol	Meaning	Symbol	Bits	Meaning
26	IFLAG2	Flag byte for operand 2 (see IFLAG1).	IBA2	Base number for operand 2.	IREG1	0-3	Work register (operand 1).
						4-7	Base register (operand 1).
27	IFLAG3	Flag byte for operand 3 (see IFLAG1).			IREG2	0-3	Work register (operand 2).
						4-7	Base register (operand 2).
28	IBCHN	Text table back-chain field.	IBA3	Base number for operand 3.	IREG3	0-3	Work register (operand 3).
						4-7	Base register (operand 3).
29					IST1	0-7	Status flag byte concerning register usage.
30	IFCHN	Text table forward-chain field.					
31							

Figure 5.93. (Part 2 of 2). Format and usage of the general area of Type-2 text tables

Bits	Meaning if on
0	The loop control variable is set.
1	The loop has non-constant bounds.
2	The table is to be ignored.
3-7	Not affected.

Figure 5.94. Use of the IGEN2 byte

Bits	Meaning if on
0	Execution-time DED required for operand 1.
1	Execution-time DED required for operand 2.
2	Execution-time DED required for operand 3.
3-7	Not affected.

Figure 5.95. Use of the IGEN27 byte

Figure 5.96 shows the main features of the operands used in the various types of Type-2 text tables, and indicates the uses of those text tables. An attempt has been made to indicate the phases that create and delete (or replace) the text tables. However, some types of text tables are created by a number of phases; in such cases, only the first phase that generates them may be shown.

Operator		Operand 1	Operand 2	Operand 3	Comments	Phase	
IOP1	IOP2					Cr	D1
ACCUM A7	00	Variable.			Accumulator for size of VDA.	IA	IQ
		Variable or temp.		Index temp.	Accumulator for variable incremented in a loop.	OE	QE
AID AD		Iteration. factor.	Null.	Array.	The IASSN table initializes the next operand 2 elements of the array in operand 3 with the initial value in operand 1. If more than one level of iteration factor occurs in the source, then AID and ENDAID tables mark the limits of the iteration factors.	II	KE PA
ALIST F4	00	Number of arguments in the list.	Post-PA: Offset of arg. list in static	argument list identification number	Start of an argument list for a procedure or library call.	II	SD
	01				ALIST for function call.		
ALLOC 70	00	IN option specification.	SET option specification.	ALLOCATE variable specification.	ALLOCATE (CONTROLLED/ BASED variable).	II	
ALOBIF A6	00	Controlled variable.	Null.	Fixed-binary target.	ALLOCATION bif.	KK	SD
AND 5E	00	Fixed-length bit string.	Fixed-length bit string.	Result (<2049 bits).	Bit string ANDing operations: Op3 = Op1&Op2. These tables generate the same code as MOVE tables with corresponding IOP2 (with NC instead of MVC instructions). Operands 1. and 2 are converted to bit string form if necessary.	II	SC
	03	Fixed-length bit string.	Fixed length bit string.	Result (>2048 bits).			
	05	Null.	Immediate field.	Target.			
	0A	adjustable and/or VARYING bit string.	Fixed-length bit string (<2049 bits).	Result.			
	0B	Short (<2049 bits) VARYING bit string.	Longer VARYING bit string.				
	0C	Adjustable or short (<2049 bits) VARYING bit string.	Short or shorter VARYING bit string.				

Figure 5.96. (Part 1 of 32). Usage of operands in Type-2 text tables

Operator		Operand 1	Operand 2	Operand 3	Comments	Phase	
IOP1	IOP2					Cr	Dl
ARG DC	00	Argument	Position in argument list	Offset of list element.	Argument table, deleted by PA if the argument is STATIC.	II	SD
	01	Argument.	Position in argument list	Argument list identification number.	Argument table for a new temporary argument.	KQ	
	02	Operand.			Operand not referenced in source, requiring a locator for DATA-directed I/O.	PC	PA
	03	Controlled argument.	Position in argument. list.	argument list identification number.	argument list used for uments to controlled uments to controlled parameters.		
ASSN 57	00	Source - binary (p1,q1).		Target - binary (p3,q3).	Binary assignment, with p1=p3, q1=q3.	II	
	01	Source - binary (p1,q1).		Target - binary (p3,q3).	Binary assignment, with q1<=q3. IST2 is used as follows: .... ..1. (p1-q1)> (p3-q3). ....1 SIZE is on.		SQ
	02	Source - binary (p1,q1).	Rounding constant.	Target - binary (p3,q3).	Binary assignment, with q1>q3. IST2 is used as shown for IOP2=X'01'.		
	03	Source - float (p1).		Target - float (p3).	Float assignment, with p1¬=p3. IST2: ....1 p3>p1.		

Figure 5.96. (Part 2 of 32). Usage of operands in Type-2 text tables

Operator		Operand 1	Operand 2	Operand 3	Comments	Phase	
IOP1	IOP2					Cr	D1
ASSN 57 (contd)	04	Source - decimal (p1,q1).		Target - decimal (p3,q3).	Decimal assignment with p1=p3 and q1=q3.	II	
	05	Source - label constant.	Invocation pointer.	Target - label variable.	Label assignment.		SQ
	06	Source - float (p1).		Target - float (p3).	Float assignment with p1=p3.		
	08	Source - binary (63).		Target - binary (31).	Binary assignment, with doubleword opl. IST2: ....1 SIZE is on.	II	
	09	Source - label variable.		Target - label variable.	Label assignment.		
	0A	Source - pointer,		Target - pointer.	Pointer assignment.		
	0B	Source - binary.		Target - binary.	Binary assignment, with no test for negative.		
	0C	Source.		Target.	Target=Source-1.		
	0D	Source - entry variable.		Target - entry variable.	Entry assignment.		
	0E	Source - external entry point.		Target - entry variable			
	0F	Source- internal entry point.		Target - entry variable.			
	1B	Source.		Target.	Produces an MVZ on 1 byte.		
	1C	Source.		Target.	Produces an MVN on 1 byte.		
	1D	Source.		Target.	Produces a ZAP (size not possible).		

Figure 5.96. (Part 3 of 32). Usage of operands in Type-2 text tables

Operator		Operand 1	Operand 2	Operand 3	Comments	Phase	
IOP1	IOP2					Cr	Dl
ASSN 57 (contd)					Decimal assignment The following notation is used in the ten tables described below: n=q3-q1. L1=length of op1. L3=(n+1)/2. t=(n+1)/2. IST2 is used as follows: 1... .... SIZE is on. .1.. .... L3>L1+t. ..1. .... p3 is even. Bit 3 Always off. Bits 4-7 Used to indicate that certain tables are to be treated as extensions of ASSN. IOP2 depends on the values of n, L1, L3, and t, as follows:		
	10				n=0.		
	11				n>0 & even, L3>t.		
	12				n>0 & even, L3<=t.		
	13				n>0 & odd, L3>t.		
	14				n>0 & odd, L3<=t.		
	15				n<0 & even, L1>-t.		
	16				n<0 & even, L1<=-t.		
	17				n<0 & odd, L1<-t.		
	18				n<0 & odd, L1<=-t.		
	19				Both operands refer to the same variable so no code is generated.		
ASSNEX 40	00	Source.		Target.	Produces STC.		SQ
	08	Source.		Target.	Extended float assignment. (applies to OS version of compiler only.)		

Figure 5.96. (Part 4 of 32). Usage of operands in Type-2 text tables

Operator		Operand 1	Operand 2	Operand 3	Comments	Phase	
IOP1	IOP2					Cr	C1
BADDR 3F	00	Offset base number.		Storage base number.	Generates addressing code.	QI	
	01	Pointer.		Unique base number.	Sets up a 'base' for a pointer.	PI	
	02	Offset base number.	Offset of adcon.	Unique base number.	Sets up a 'base' for temporary storage.		
	03		Variable.		Loads the base for an outer procedure.	QA	
	04	Offset of DSA back chain.		Offset in temporary storage of base.	This table is changed to ASSN in Phase QA.	QI	QA
	05	Offset of anchor, or var.DR if OS.	Null.	Null.	Sets pointer to the latest allocation of controlled variable.	PI	SA
	06	Offset in parameter list.	Null.	Null.	Sets pointer to the item which is a parameter.		QA
	07	Offset of temp storage in outer DSA.			Sets up base of temp. in an outer block. Changed to LA(02) table by Phase QA.		
	08	Base number.	Offset in descriptor.	Base number.	Changed to MINUS table by Phase QA.		
	09	Base number.	Offset in descriptor.	Base number.	Changed to PLUS table by Phase QA.		
	0A	Offset in parameter list or '47'...file DR			Loads base for a cont-rolled parameter or the FCB of a file parameter/ file variable. (Applies to OS version of compiler only.)		SA
	0B	Parameter DR.			Loads base for ccnt-rolled parameter anchor slot. (Applies to OS version of compiler only.)		
	0C				Loads PRV base. (Applies to OS version of compiler only.)	QA	
	0D	'48'...file DR			Loads base for file constant FCB. (Applies to OS version of compiler only.)	PI	

Figure 5.96. (Part 5 of 32). Usage of operands in Type-2 text tables

Operator		Operand 1	Operand 2	Operand 3	Comments	Phase	
IOP1	IOP2					Cr	Dl
BC 2F		Logical operations. Comparison operations are represented by EQ, NE, GE, GT, LE, and LT text tables generated by Phase II. If these tables form part of an IF statement, they are replaced by BC/BCB tables by Phase KA; otherwise, they are replaced by BC/BCB tables by either Phase KV or Phase OC. (These tables may in turn be replaced by BEQ, etc., tables by Phase SQ during code generation.) The basic form of BC/BCB tables is as follows:					
		X(p1,q1).	Y(p2,q2).	Branch mask (BEQ, BNE, BCE, BGT, BLE, BLT) and label.	Compares X and Y, and branches according to resulting condition code.		
		The value of IOP2 for BC/BCB tables varies according to the operand types as follows:					
	00				Binary, with q1=q2.		SQ
	01				Binary, with q1¬=q2. Rearrangement requirements and IST2 are as for the PLUS X'01' table.		
	02				Float, with p1=p2.		
	03				Decimal, with operands arranged so that q1<=q2, and IST2 bit 0 indicating whether SIZE is on.		
	04				Float, with p2¬=p2. Rearrangement requirements are as for the PLUS X'04' table.		SQ
	05				Binary - BC table only. Ignored by statement processing, and shift is suppressed even if q1¬=q2.		
	06				As above, for decimal operands.		
	07				Special BC table. Operand 1 contains an immediate field, operand 2 is the 'returns byte' in the DSA.		SD
	08				Extended float. (Applies to OS version of compiler only.)	I	SQ

Figure 5.96. (Part 6 of 32). Usage of operands in Type-2 text tables

Operator		Operand 1	Operand 2	Operand 3	Comments	Phase	
IOP1	IOP2					Cr	D1
BCB 2E	00	String.	Length to be tested.		Tests a short (<=256 bytes) fixed-length bit string for 'all-0'.		SC
	02	String.	Test mask.		Generates TM op1,op2.		
	03	String.	Length to be tested.		Tests a long (>1536 bytes) fixed-length bit string for 'all-0'.		
BEGIN 92					See PROC table.		
BEQ 2A					See BC table.	SQ	SQ
BGE 28					See BC table.		
BGT 2D					See BC table.		
BIF F7	n	Built-in function argument.	Built-in function argument.	Built-in function result.	Operand 3 will contain 'Null' if more than 1 table is needed for the bif. n identifies the bif.	II	OX
BLE 2C					See BC table.	SQ	SQ
BLT 29					See BC table.		
BNEQ 2B					See BC table.		
BXLE 6A	00			Label.	Generates BXLE op3.	QA	SA
	07			Label.	Generates BXH op3.		
	08			Label.	Generates BXH op3.		

CALL 80	00	Argument list number. Post-PA: Address of argument list.	Entry point.		Calls an entry point to a block.	II	SA
	01	Argument list number. Post-PA: Address of argument list.	Library entry point number.		Calls a library subroutine.	KQ	
	02	Parameter list.	CGSR entry point.	Special registers for call.	Calls a compiler-generated subroutine.		
	03	Source.	CGSR entry point number.	Target.	Calls a string CGSR (preceded by KONST 02).	OC OX	

Figure 5.96. (Part 7 of 32). Usage of operands in Type-2 text tables

Operator		Operand 1	Operand 2	Operand 3	Comments	Phase	
IOP1	IOP2					Cr	Dl
CALL 80	04	Argument list number. Post-PA: Address of argument list.	Offset from Reg.12 of a slot in the TCA.		Call to a routine in the TCA.		SA
(contd)	05	Apparent entry label.	Prologue label.		Call to common section of prologue code.		
	07		Entry point.		Load address of entry point.		
CGSR 20	00		CGSR.		Start of a CGSR.	KQ	SA
	01		CGSR.		End of a CGSR.		
CHANE A3	00		Number of dynamic ONCBs.	First dynamic ONCB.	Chains a dynamic ONCB (got with the DSA).	PI	
	01		Number of dynamic ONCBs.	First dynamic ONCB.	Chains a dynamic ONCB (got with the VDA).		
CHECKC A0	00		Offset of ONCB.	Check parameter.	CHECK parameter on a statement.	II	
	01		Offset of ONCB.	Check parameter.	CHECK parameter on a block.		
	02		Offset of ONCB.	File DR.		KV	
	03		Offset of ONCB.	Condition DR.			
CLOSE 8C	00	File.	Environment options.	Null.	CLOSE statement (last file in statement).	II	KL
	80	File	Environment options.	Null.	CLOSE statement (not last file in statement).		
COMP 27	00	String (non-varying).	Length for comparison.	String (non-varying).	Compares short (<=256 bytes) strings.	OX	SC
	01	String (non-varying).	Length for comparison.	Comparison byte (byte 5).	Compares remainder of a string with blank or zero.		
KONST F8	00	Mask to be ANDed with op1.	Mask to be ANDed with op3.		Compares two bit strings (<=8 bytes) after ANDing out any spare bits in the byte. (COMP X'02' is always preceded by KONST X'00').		
COMP 27	02	Bit string.		Bit string.			

Figure 5.96. (Part 8 of 32). Usage of operands in Type-2 text tables

Operator		Operand 1	Operand 2	Operand 3	Comments	Phase	
IOP1	IOP2					Cr	D1
COMP 27 (contd)	03	String (non-varying).	Length for comparison.	String (non-varying).	Compares long (>256 bytes) strings.	OX	SC
	05	Character string, of length <=op3-1 (i.e., bytes 1 onward of op3).	Bytes: 0 Content: Code; 1-2 Unused; 3 Mask; 4-5 1.op3-1	Character string, length <=256 bytes.	Compares op1 with op3, or compares op1 or op3 with mask in op2.		
	06		Bytes: 0 Content: Code; 1-2 Unused; 3 Mask; 4-5 1.op3-1	Character string, length >256 bytes.	As above, for a character string of length >256 bytes.		
CONCAT 60	00	String, converted if necessary.	String, converted if necessary.	Result.	Concatenation operation.	II	KV OC
CONV 3C	00	Source.		Target.	Compiler-generated conversion.		
KONST F8	07	Reference to the DED of the source.		Reference to the DED of the target.	Contains the necessary references for CCNV X'01' operands which require object-time DEDs.	PC	QA
CONV 3C	01	Source.	Library entry point number.	Target.	Conversion via the library. This table is preceded by a KONST X'07' table after Phase PC.	KX	SD
CONV 3C	02	Source. binary.	Workspace.	Target - float.	Binary to Float conversion. IST2 is used as follows: .... .1.. Zero flcat register Bits 6-7 Number of AER-ADR instructions.		
	03	Source - float.	Constant.	Target - binary.	Float to binary conversion. IST2 is used as follows: .... .1.. SIZE is on. Bits 6-7 Number of HER-HDR instructions.		

Figure 5.96. (Part 9 of 32). Usage of operands in Type-2 text tables

Operator		Operand 1	Operand 2	Operand 3	Comments	Phase	
IOP1	IOP2					Cr	D1
CONV 3C (contd)	04	Source.	Workspace.	Target.	Fixed decimal to float conversion. IST2 is used as follows: .... short target ....1 long target	KX	SD
	05	Source.	Workspace.	Target.	Long/short float to fixed decimal conversion.		
	06	Source.		Target.	Fixed binary to fixed decimal conversion.		
	07	Source.		Target.	Fixed decimal to fixed binary conversion.		
	08	Source.		Target.	Character(1) to decimal conversion.		
	09	Source.	Constant.	Target.	Character(1) to float conversion.		
	0A	Source.	Constant.	Target.	Character(1) to fixed binary conversion.		
KONST F8	02	Pattern for target digits.		Length of pattern.			
CONV 3C	0C	Source.	Q.temp.(offset in target).	Target.	Fixed decimal to character(n) where n is a compile-time integer. (CONV(0C) is always preceded by KONST(02)).	KX	SD
KONST F8	02	Pattern for target digits.		Length of pattern.			
CONV 3C	0D	Source.	Q.temp.(offset in target).	Target.	(CONV(0D) is always preceded by KONST(02)).	KX	SD
CONV 3C	0E	Source.		Target.	Bit string to character string conversion.		
	0F	Binary workspace.	Workspace.	Target.	Fixed binary to bit conversion.		
	10	Source.	Constant.	Workspace.	Float to binary conversion.		
	11	Source.	Workspace.	Binary workspace.	Bit string to fixed binary conversion.		
	12 14	Source.	Q.temp.	Workspace or target.	Fixed decimal to fixed picture conversion.		
	15	Source.	Q.temp.	Target.	Fixed picture to fixed decimal conversion.		

Figure 5.96. (Part 10 of 32). Usage of operands in Type-2 text tables

Operator		Operand 1	Operand 2	Operand 3	Comments	Phase	
IOP1	IOP2					Cr	Dl
DATAE BE	00	Null	Null	Element.	Data element (always followed by 5 NULL tables).	II	KQ
DEL E2	00	XMESG parameter	XMESG parameter	XMESG parameter	Tables containing errcr message information (for message generaticn by Phase KA).		KA
DELAY 73		Null.	Null.	Delay expression.			KT
DELETE 85					See the last part of this figure (Applies to OS version of compiler only.)		
DINC 6A	00	Loop control variable.	Increment.	Loop control variable.	Increments DO-LOOP control variables prior to branch-back.	KI	QA
DIVIDE 61	00	Divisor - binary (p1,q1).	Dividend - binary (p2,q2).	Quotient - binary (p3,q3).	IST2: ....1 p1=15.	II	SQ
	01	Divisor - float (p1).	Dividend - float (p2).	Quotient - float (p3).	Float division, with p1=p2. IST2: ....1 p3>p1.		
	02	Divisor - float (p1).	Dividend - float (p2).	Quotient - float (p3).	Float division, with p1¬=p2. IST2: ....1 p1<p2.		
	03	Divisor - decimal.	Dividend - decimal.	Quotient - decimal.	IST2 bits 5 and 7 are always set, to trigger generation of a DP instruction.		
	04	Divisor - binary (p1,q1).	Dividend - binary (p2,q2).	Remainder - binary (p3,q3).	IST2: ....1 q1=15.		SQ
	05	Divisor - decimal.	Dividend - decimal.	Remainder - decimal.	IST2 bits 5 and 7 are always set, to trigger generation of a DP instruction.		
	06	Divisor - decimal.	Dividend - decimal.	Quotient - decimal.	Generates a DP instruction only.		
	08	Divisor - ext.float.	Dividend - ext.float.	Quotient - ext.float.	Generates an interrupt which is detected by the library, which then calls the floating point divide routine. (Applies to OS version of the compiler only.)	QI	SQ

Figure 5.96. (Part 11 of 32). Usage of operands in Type-2 text tables

Operator		Operand 1	Operand 2	Operand 3	Comments	Phase	
IOP2	IOP2					Cr	Dl
DLIST DD	00	Base array.	Defined array	Null.	Table indicating start of a DEFINED iSUB subscript list.	II	KE
DO1 54	00	Null	GSL at end of the do-loop	Loop control variable.	First table for iterative DO statement.		
DO2 55	00	TO specification.	BY specification.	GSL of first statement in the do-loop.	Second table for an iterative DO statement.		
DO3 50	00	GSL at end of a DO-loop.			End DO-loop marker		
DOWHYL 6E	00	WHILE specification	GSL of first statement in the do-loop.		DO WHILE case (no TO or BY expression)		
DROSS 0E		Not a true text table; used to force alignment of text tables during code generation.				SA	SK
DSPLAY 76		Null.	Null.	Display expression.		II	KT
DSUBS DB	00		Subscripted array information.		Intermediate subscript in an iSUB DEFINED subscript list.		KE
DSUBSL DF	00		Subscripted array information.		Last subscript in an iSUB DEFINED subscript list.		
DSUBS1 DE	00	Subscript. See SUBS1 for full details.	Subscripted array information.	Index temporary operand.	First subscript in an iSUB DEFINED subscript list.		
ENDAID AE		Null.	Null.	Array.	See AID table.		KE PA
ENDIT 97	00			Temporary control variable.	End of array expansion do-loop.		
ENDPAGE 99	00	Null.	Null.	Null.	'End of page' marker.		
ENTRY 91	00	Dictionary reference of real entry point.	Flag for PL/I, COBOI, cr FCRTRAN	Apparent entry point.	Indicates a real secondary entry point in the prologue code.	KT	SA
ENDPROG 98	00	Null.	Null.	Null.	'End of program' table	II	
EQ 5A		X(p1,q1).	Y(p2,q2).	Result of comparison of X and Y.	See BC table.		

Figure 5.96. (Part 12 of 32). Usage of operands in Type-2 text tables

Operator		Operand 1	Operand 2	Operand 3	Comments	Phase	
IOP1	IOP2					Cr	Dl
EVO CF	00	Length of COBOL structure (if COBOL file).		Event variable.	Follows all RECORD I/O tables except LOCATE. See last part of this figure.	II	KM KT
EXIT 74		Null.	Null	Null.			KT
FIT AC	00	Iteration factor or expression.			Format iteration.	KQ	
FITE AB	00				Format iteration end.		
FLWUNT 9A					Flow-unit header; see figures 5.98 and 5.99.	CE	PA
FMTLIST C8	00				Start of format list.	II	KQ
FNCT D7	00	Function name.	Argument list	Result.	Function reference.		KK
FORME CE	Type	W, P, or label or null.	d, or null.	P, S, or null.	Format element (exact contents of operands depends on format item). IOP2 shows type of format item.		KQ
FREE 71	00	IN option (or null).	Pointer qualifying generation of BASED variable (or null).	Variable to be freed.	Operand 2 is null for CTL variables.		IQ
GE 58					See EQ table.		
GEN 3A	00	L, string.			Generates a literal string up to 29 bytes long. First byte = length.	KQ OX	SA
	01	L, string.			Generates a literal HEX constant.		
	02				Generates a 4-byte adcon for library module.		
GET 89	00	FILE (filename) or STRING (stringname).	SKIP specification			II	KQ

Figure 5.96. (Part 13 of 32). Usage of operands in Type-2 text tables

Operator		Operand 1	Operand 2	Operand 3	Comments	Phase	
IOP1	IOP2					Cr	Dl
GHOST 1F	00	Base variable to which Q.temp. in preceding table refers.	Base variable to which Q.temp. in preceding table refers.	Base variable to which Q.temp. in preceding table refers.	Passes used set info. Table ignored by all phases except OM.	KM KQ OE	PA
GOOB 7F	00			Label constant	General case (via TCA).	II	SA
	01			Label variable.	Always done via a TCA.		
	02			Label constant.	Special case (in-line code).		
GOTO 7E	00			Label constant.			
	01	GSL or null.	GSL or null.	Label variable or temporary label variable.	If op3 is a temporary, then op1 and op2 contain its first and last values.	KQ	
	02			Label constant.	BXLE required.	QA	
	03			Branch mask and label number.	Conditional branch table, to generate BC branch mask, label	KQ OC OX	
	05	Variable, used to set the condition code.		Branch mask and label number.	Conditional branch table, to generate LTR OP1,OP1 BC branchmask, label		SA
	06	Temporary variable.		Label constant.	Generates BCT instruction to reduce op1 by 1 and branch to label if op1¬=0.	KI	
	07	Variable, used to set the condition code.	Test-mask.	Branch mask and label number.	Conditional branch table, to generate TM testmask, op1 BC branchmask, label	KE	
	08	Variable, used to set the condition code.		Branch mask and label number.	Conditional branch table, to generate LTER OP1,OP1 BC branchmask,label.	KA	
	09	Link register number (Rn).		Label constant.	Generates BAL,Rn,label.		
	0A	Link register number (Rn).		Label variable.	Generates BAL, Rn,labvar.		

Figure 5.96. (Part 14 of 32). Usage of operands in Type-2 text tables

Operator		Operand 1	Operand 2	Operand 3	Comments	Phase	
IOP1	IOP2					Cr	Dl
GSL FF	00	Label number.			GSL table.	KQ	
	01	Label number.		Base register number (Rn).	GSL table, specifying Rn as base register after branch.		
	02				ENTRY statement GSL.		
	04	Label number.			Position of label must be switched to follow the 'next' SN table.	KA	SA
GSL2 FB	00				Produces no code. Used as separator to avoid commoning of labels.		
GSN FE	00				Generated statement header. See table 5.90.	II	
GT 5D	00				See EQ table.		
HASH 9D	00				Follows flow-unit header or page header (see fig.5.100).	OE	KK PC
IASSN AF	00	Source	Iteration factor result.	Array target.	Array initial assignment.	II	KE PA
IF 81	00	IF expression.	GSL at end of THEN clause.		If op1 is true, then branch to op2.		
ITDO 94	00			Temporary control variable.	First table of an array expansion do-loop.		SA
KONST F8	00	Constant.	Constant or null.	Constant or null.	Table for storing up to 3 constants.	KX	
	01	Q.temp. number.			Used to flag temps or Q.temp.s as 'dead'.	PI	QA
	02				Used to extend a following table. Its operands are checked for register residence, and bases and flags are set.		
	03	Register number.	Register number or null.	Register number or null.	Reserves up to three registers.	KM	QA

Figure 5.96. (Part 15 of 32). Usage of operands in Type-2 text tables

Operator		Operand 1	Operand 2	Operand 3	Comments	Phase	
IOP1	IOP2					Cr	Dl
KONST F8 (contd)	04	Register number.	Register number or null.	Register number or null.	Frees up to three registers.	KM	QA
	05	Register number.			Indicates that op1 is to be used as the base register, instead of register 2.		
	06	Register number.			Indicates that op1 is to be dropped as the base register (revert to register 2).		
	07				See CONV X'01' table.	PC	QA
	08	Parameter base number.	Null	Null	Optimization only. Appears before end of which has had parameter addressing code moved out by Phase QI. Ensures that the base is in the same register at the end of the lcop as at the head.	QI	
	09	Register number.			Indicates that a register is reserved for the next table.		
	0A	Primary function.					
		Uncommoned constant, or null.	Uncommoned constant, or null.	Uncommoned constant, or null.	Up to three constant operands.	OC	
		Secondary function.			Optimization only. This is a marker table to inform Phase QA that a flow unit contains a branch table follcwed by moved out code. Data cannot be carried forward in registers to any connection other than the first.	QI	QA
	0B	Null.	Null.	Register temporary operand.	Clears a register (by SR instruction) and gives it the name specified in op3.		SA
	0C		Loop number.	Loop control variable.	Optimization only. This is a dummy store of the loop control variable at the head of an optimized loop. It is changed to a store by Phase QE, if Phase QA decides it is necessary.	QA	QE

Figure 5.96. (Part 16 of 32). Usage of operands in Type-2 text tables

Operator		Operand 1	Operand 2	Operand 3	Comments	Phase	
IOP1	IOP2					Cr	Dl
KONST F8 (contd)	0D				Phase QA gets work registers for operand1 and operand2. This table precedes MAP(00) and MAP(04) tables.		SA
	0E	Accumulator		X'7E' | X'79'	Marks end of accumulator. Used by Phase QA to reset register status. ICDE3 is X'7E' if accumulator is an array element.	QI	QA
	0F		Mask.		Generates AND with operand1 as source and operand2 as mask.		SA
	10	Parameter base number.	Bit vector offset.	storage offset.	Optimization only. This is a dummy store of a parameter base at the head of a loop out of which Phase QI has moved parameter addressing code. It is changed to a store by Phase QE if Phase QA finds it necessary, i.e. if a load of the base is required.	QI	QE
	11	Storage offset.	Bit vector offset.	Parameter base number	Optimization only. This is a dummy load of a parameter base inside a loop out of which Phase QI has moved the parameter addressing code. It is changed to a load by Phase QA if the parameter base has been lost from a register, and the bit corresponding to the value in IREF2 is set in a vector to indicate to Phase QE that the KONST(10) table at the head is to be changed to a store.		QA
	12	X'79' | X'7E'	Temporary register storage offset.	Global temporary register storage offset.	ICDE1 is X'7E' if temporary register storage occurs at the end of temporary storage, X'79' otherwise.		

Figure 5.96. (Part 17 of 32). Usage of operands in Type-2 text tables

Operator		Operand 1	Operand 2	Operand 3	Comments	Phase	
IOP1	IOP2					Cr	D1
KONST F8 (contd)	13	Operand1 of CALL(03) table.	Null.	Operand3 of CALL(03) table.	Phase QI splits CALL(03).op1.op2.op3 into (KONST(13).op1.null.op3 (CALL(03).null.op2.null	QI	SA
	14	1st register number.	2nd register number.	3rd register number.	Marks a register in use but does not clear its contents (currently only used for register1 in stream I/O).		QA
	20	Constant.	Reference of pseudo-constants pool entry for a FED.	Target.	Convert non-decimal/ integer constants into correct FED positions.	KQ	PA
LADDR 24	00	Null.	Bump constant (<4096).	Register number.	Assigns constant op2 to op3.		SA
	01	Source.	Bump constant (<4096).	Target.	Bumps op1 by op2 and assigns it to op3.		
	02	Source.		Target.	Loads the address of a variable into a register or storage.		SD
	04	Variable or temporary.	Null.		Loads bit address of aligned bit string.	PI	
	05	Variable or temporary			Loads bit address of unaligned bit string.		
	06	Variable or temporary.	Null.		Loads bit address of unaligned bit string where offset is unknown at compile-time.		
	07	String address temporary.	Null.		Loads bit address of aligned string address temporary.		
LE 5C	00				See EQ table.		
LLAD 21	00		Library entry point.		Loads address of a Library entry point into register 15.	KQ	SA
	19 1A				See last part of this figure.	II	KM
LOCATE 88	00	ABNORMAL LOCATE return label.		SET parameter.	Second table in a LOCATE statement. (Always generated)		
LT 59	00				See EQ table.		

Figure 5.96. (Part 18 of 32). Usage of operands in Type-2 text tables

Operator		Operand 1	Operand 2	Operand 3	Comments	Phase	
IOP1	IOP2					Cr	D1
MAP 52	00	Aggregate to be mapped.	REFER flag (byte 2) = zero.	Size temporary (FIXED BIN fullword).	Automatic adjustable, controlled, or based aggregate-mapping table.	IA	IQ
		Item to be mapped.	REFER information: Bytes: Content 0: X'7E' 1: Flag (X'80') 2-3: ATR for start of mapping. 4-5: ATR for end of mapping	Size temporary, as above.	Map table for BASED-REFER (map on reference) items.		
		Null, indicating that the size temp has already been set up.	REFER flag (byte 2) = zero.	Size temporary (FIXED BIN fullword).	Map table for allocation of a controlled non-aggregate string.		
		String to be mapped.	REFER flag (byte 2) = zero.	Size temporary as above.	Map table for automatic adjustable non-aggregate string.		
	01	Locator.	Descriptor.	Descriptor descriptor.	Initialize locator for a descriptor.	IQ	SC
	02	Locator.			Initialize aggregate locator for adjustable extents.		
	03	Locator.	Descriptor.	Null.	Initialize locator for controlled string.		
	04	Descriptor.	Element length.	Null.	Map dynamic FORTRAN array.		
	05	Temporary pointing at storage for structure.	Storage temporary containing hang and length.	Temporary to be set to point at structure.	Relocate automatic adjustable structure.		
	06	Temporary operand.			Temporary operand holding length of allocated variable in LOCATE statement.		KM
	07	Temporary pointing at Storage for structure.	Storage temporary containing hang and length.	Temporary to be set to point at structure.	Relocate adjustable structure.		SC

Figure 5.96. (Part 19 of 32). Usage of operands in Type-2 text tables

Operator		Operand 1	Operand 2	Operand 3	Comments	Phase	
IOP1	IOP2					Cr	Dl
MASSN 56	00	Reference to current or maximum string length.	Reference to current or maximum string length.	Current length of a varying string.	Compare op1 and op2 and assign the smaller value to op3.	OC	SC
	01	Reference to length of a string.		Register temp target.	Rounds value of op1 up to the nearest multiple of eight.		
	02	Adjustable string length or variable.		Register temp target.	Converts a byte length to a bit length.		
	03	Constant or variable.	String length	String.	Sets argument registers for the HIGH and LOW functions.	OX	
	04	Mask			Generates an MVI for the HIGH and LOW functions.		
	06	Bit offset.	TEMPLOC+7.	Index temp.	Generates code to add the bit offset of RVO to the index temp. Sets up bit offset in TEMPLOC+7, and converts index to bytes.	PI	
	07	Negative offset.		Next available base number.	Sets up addressing code to address array with V.O. outside scope of the DSA or STATIC.		
	08	Descriptor or locator reference.	Null.	Register temporary.	Clears operand3 and assigns 1st byte of operand1 to operand3.		
	09	Variable to be rounded.	Mask.	Increment factor.	Rounds operand1 according to mask in operand2.	IQ	
	0A	File variable.	Offset within FCB.	Target.	Assigns a halfword field from the FCB of a file variable or parameter to operand3.	KK	
MGT 0F	Not a true text table, but a variable-length marker preceding text tables that have not been replaced by object code.					SA	

Figure 5.96. (Part 20 of 32). Usage of operands in Type-2 text tables

Operator		Operand 1	Operand 2	Operand 3	Comments	Phase	
IOP1	IOP2					Cr	Dl
MINUS 64	00	Binary (p1,q1).	Binary (p2,q2).	Result - binary (p3,q3).	Binary subtraction, with q1=q2.	II	SQ
	01	Binary (p1,q1).	Binary (p2,q2).	Result - binary (p3,q3).	Binary subtraction, with one of the following scale requirements: • IST2 bit 7 off - q1<q2 and p1+(q2-q1) <=15. • IST2 bit 7 on - q1>q2 and p2-(q2-q1) >15.		
	02	Binary (p1,q1).	Binary (p2,q2).	Result - binary (p3,q3).	Binary subtraction, with one of the following scale requirements: • IST2 bit 7 off - q1>q2 and p2-(q2-q1) <=15. • IST2 bit 7 on - q1<q2 and p1+(q2-q1) >15.		
	03	Float (p1).	Float (p2).	Result - float (p3).	p1=p2. IST2 bit 7 on indicates p3>p1.		
	04	Decimal (p1,q1).	Decimal (p2,q2).	Result - decimal (p3,q3).	Operands 1 and 2 are arranged so that q1<=q2. IST2 is used as follows: 1... .... SIZE is on. .... ..1. SUBTRACT PACKED (SP) flag - always on. ....1 q1=q2=q3, and y=T. Operands are reordered so that T=T=X.		
	05	Float (p1).	Float (p2).	Result.	p1<p2.		SQ
	06	Float (p1).	Float (p2).	Result.	p1>p2.		
	08	Extended float.	Extended float.	Extended float.	Extended float subtraction. (Applies to OS version of compiler only).	QI	

Figure 5.96. (Part 21 of 32). Usage of operands in Type-2 text tables

Operator		Operand 1	Operand 2	Operand 3	Comments	Phase	
IOP1	IOP2					Cr	Dl
MOVE 9E	00	Fixed length string.	Length of move (<257 bytes).	Fixed length string.	Short, fixed length string move.	OC	SC
	01	Offset of argument from start of ALIST (R1).		Parameter.	Argument list to parameter list string move.		
	02	Immediate field (byte 5).		Offset of return byte in DSA.	Sets the return flag in the DSA.		
	03	Fixed length string.	Length of move (>1536 bytes).	Fixed length string move.	Long fixed length		
KONST F8	00	op4=target length.	op5=target length-3.	op6=1.	Adjustable and/or VARYING source to short fixed length target. (MOVE X'04' is always preceded by KONST X'00').		
MOVE 9E	04	Adjustable and/or VAR CHAR.		Short fixed length CHAR.			
MOVE 9E	05	Q-temp: start of padding+1.	Byte 3: Padding character Byte 4-5: Length of padding-1 (<257).	Q-temp: start of padding.	Padding move for short (<257) fixed length string target.		
	06		Byte 3: Padding character Byte 4-5: Length of padding-1 (>1536).	Q-temp: start of padding.	Padding move for long (>1536) fixed length string target.		
	07	Short (<257) VARYING CHAR.		Longer VARYING CHAR.	VARYING (max1) CHAR tc VARYING (max2) CHAR with 257>max1<max2.		
	08	Adjustable CHAR or short VARYING CHAR.	Constant (1).	Short VARYING CHAR.	Adjustable to short VARYING or short VARYING to shorter VARYING.		
KONST F8	00	op4=length of shorter string.			String move from source to target, via workspace if they overlay; otherwise directly. (MOVE X'09' is always preceded by KONST X'00').		
MOVE 9E	09	Short fixed-length string.	Workspace.	Short fixed-length string.			

Figure 5.96. (Part 22 of 32). Usage of operands in Type-2 text tables

Operator		Operand 1	Operand 2	Operand 3	Comments	Phase	
IOP1	IOP2					CR	D1
KONST F8	00	Op4=smallest number of lists with same number of bytes as target.	Op5=number of bytes required for target-2	Op6=7.	Adjustable or VARYING source to short fixed length target. (MOVE X'0A' is always preceded by KONST X'00').	OC	
MOVE 9E (contd)	0A	Adjustable or VARYING BIT.	Table of masks for NC instruction.	Short fixed-length BIT.			
MOVE 9E	0B	Short (<2049) VARYING BIT.		Longer VARYING BIT.	VARYING (max1) BIT to VARYING (max2) BIT with 2049>max1<max2.		
	0C	Adjustable BIT or short VARYING BIT.	Q-temp: end of target.	Short VARYING BIT.	Adjustable to short VARYING or short VARYING to shorter VARYING.		
	0D	Immediate field (byte 5).		Q-temp: end of target.	ANDs out spare bits at end of target when not multiple of 8 bits.		
	0E		Length of padding (>1 and <9 bytes)	Q-temps: start of padding.	Bit string padding table to generate an XC instruction.		
	10	VARYING string.			Clears top bits of a VARYING string.	OX	SC
	14	Temporary.	Shift constant	String temporary.	Short fixed-length bit string concatenation.	KV	
	17	ARG1.	ARG2.	String temporary.	Short VARYING bit-string concatenation.		
MULT 62	00	Multiplicand-binary (p1,q1).	Multiplier - binary (p2,q2).	Product - binary (p3,q3).	T2 is used as follows: .... ..1. op1=op2, so MR R3,R2 nct MR R3, R3+1 is generated. ....1 p1+p2+1<=15, so result is in odd register of a pair, and SLDA is omitted.	II	SQ
	01	Multiplicand - halfword binary.	Multiplier - halfword binary.	Product - halfword binary.		QA	

Figure 5.96. (Part 23 of 32). Usage of operands in Type-2 text tables

Operator		Operand 1	Operand 2	Operand 3	Comments	Phase	
IOP1	IOP2					Cr	Dl
MULT 62 (contd)	02	Multiplicand - float (p1).	Multiplier - float (p2).	Product - float (p3).	p1=p2.	II	SQ
	03	Multiplicand- decimal.	Multiplier - decimal.	Product - decimal.	IST2 is used as follows: .... .1.. Always on - triggers MP and ZAP code generation. ....1 No overflow is specified ZAP code generation is supp- ressed.		
	04	Multiplicand - float (p1).	Multiplier - float (p2).	Product - float (p3).	p1¬=p2, with operands arranged so that p1<p2.		SQ
	05	Multiplicand and product.	Multiplier.		Multiplies in situ. extending the op1 field with zero to accomm- odate the product length.		
	08	Extended float.	Extended float.	Extended float.	Extended float multiplication. (Applies to OS version of compiler only.)	QI	SQ
NDX 3D	00	Index temp- orary (dead).	Indexed variable or Q-temp if BASED.	Q-temp (alive).	Used in evaluating sub- scripted array expression	II	QA
NE 5B	00				See EQ table		
NOCHK B0	00		Offset of ONCB.		NOCHECK parameter on a statement.	II	SA
	01		Offset of ONCB.		NOCHECK parameter on a block.		
NOT 66	00 03 05 0A 0B 0C	OP1	OP2	OP3 (target).	Op3=¬op1. The operand types, and the code skeletons and bit strips (except for X'0A') used in code generation, are the same as for the corres- ponding MOVE tables.		SC

Figure 5.96. (Part 24 of 32). Usage of operands in Type-2 text tables

Operator		Operand 1	Operand 2	Operand 3	Comments	Phase	
IOP1	IOP2					Cr	Dl
NULL 79	00	Null.	Null.	Null.	Table containing no useful information e.g., a deleted text table.	II	SA
OFFS 36	00	Offset number	Array reference.	Q-temp representing array element.	These two tables are used to reference offset (op1) of op2. If table is used when associated with SUBS/NDX tables then IOP2=X'00'. Otherwise IOP2=X'01'.		KE
	01	Offset number reference	Array	Q-temp representing array element.			
	02	Offset numbers	Array reference.	Q-temp representing array element.			
	03	Offset.		Q-temps.	Offset table with no dictionary reference - supplied by preceding index table.		
	04	Offset.	Offsetted item.	Q-temp.	Allows same index to be associated with different offsets.		
	05	Offset number	Array reference.	Q-temp. representing array element.			
	06			Address temp.	Tells phase OC no VDA wanted for S.A.T.		
	07	Temporary operand.	Offsetted item.	Q-temp.	Has specific register in operand2 or a temp. in operand2. Either register or register containing temp. to be used as a base at Q-temp. print.		
ONB 9B	00				ON BEGIN block header. See PROC table.	II	
ONCAD A5	00	Flag and Offset from ONCA.	Null.	Address temporary.	ONCHAR and ONCOUNT bifs.	KK	SD
	01	Flag and offset from ONCA.	Length temporary.	Address temporary.	DATAFIELD, ONFILE, ONKEY, and ONSOURCE bifs, and ONSOURCE psv.		
	02	Flag and offset from ONCA.	Null.	Address temporary.	ONCHAR psv.		

Figure 5.96. (Part 25 of 32). Usage of operands in Type-2 text tables

Operator		Operand 1	Operand 2	Operand 3	Comments	Phase	
IOP1	IOP2					Cr	D1
ONS 8D	00	File name if a file is involved.	Condition name and description of ON-unit	ONB block reference (or 0).	ON statement. Body of statement follows ONB text table.	II	SA
OPEN1 8B	00	File name.	Dictionary reference of overflow entry (attributes on OPEN).	TITLE expression result.	First text table for OPEN statement.		KL
	01	LINESIZE expression result.	PAGESIZE expression result.		Second text table for OPEN statement (for PRINT dataset).		
OR 5F	00	Op1.	Op2.	Op3(target)	Op3=Op1|Op2.		SC
	03 0A 0B 0C				The operand types and bit strips (except for on) used in code generation are the same as for the corresponding MOVE tables.		
	05		Immediate field, F (byte 3).	Op3 (target).	Generates OI Op3,F.		
PEND 9F	00	Compiler label.			Prologue end label.	II	SA
	01				End of DSA code.		
	02	Compiler label.			Start of initialization code.		
PINIT FA	00	Variable DR.			Output comment for variable initialization.		
PLUS 63	01	Binary (p1,q1).	Binary (p2,q2).	Sum - binary (p3,q3).	Binary addition, with q1¬=q2. IST2 bit 7 on indicates that q1<q2 and p1+(q1-q2)>15, and that therefore a double register is needed. Otherwise a single register is needed and q1<q2. Operands are rearranged if required.		SQ
	02	Float (p1).	Float (p2).	Sum - float (p3).	Float addition, with p1=p2. IST2 bit 7 is on if p3>p1.		

Figure 5.96. (Part 26 of 32). Usage of operands in Type-2 text tables

Operator		Operand 1	Operand 2	Operand 3	Comments	Phase	
IOP1	IOP2					Cr	Dl
PLUS 63 (contd)	03	Decimal (p1,q1).	Decimal (p2,q2).	Sum - decimal (p3,q3).	If q1=q2=q3 and op3=op2, then the operands are rearranged so that the operation is op3=op3+op1. Otherwise the operands are arranged so that q1<=q2. IST2 is used as follows: 1... .... SIZE is on. ....1 ADD PACKED (AP) flag - always on.	II	
	04	Float (p1).	Float (p2).	Sum - float (p3).	Float addition, with p1¬=p2. Operands are arranged so that p1<p2.		SQ
	05	Float (p1).	Float (p2).	Sum - float (p3).	Unnormalized float addition, with p1=p2. IST2 bit 7 is on if p3>p1.		
	06	Binary.	Binary.	Sum - binary.	Binary addition, with no shifting.		
	08	Extended float.	Extended float.	Extended float.	Extended float addition (applies to OS version of compiler only).	QI	
PMINUS 68	00	Source.		Target.	Prefix minus table with IOP2 as shown for the ASSN table.	II	
POWER 65	00		Exponent.	Result.	Generates code to perform the operation op3=op1**op2.		
PPLUS 67	00	Source.		Target.	Prefix plus table, with IOP2 as shown for the ASSN table.		SA
PROC 90 BEGIN 92 ONB 9B	ILC				Real entry point to a procedure. See figure 5.91.		

Figure 5.96. (Part 27 of 32). Usage of operands in Type-2 text tables

Operator		Operand 1	Operand 2	Operand 3	Comments	Phase	
IOP1	IOP2					Cr	D1
PSV D6	00	Pseudo-variable argument.	Second pseudo-variable argument.	String address temporary. The string address is null for SUBSTR with 3 args since 2 PSV tables exist, the second of which contains the string address temp.	Pseudovariable table used for STRING pseudo-variables only.	II	KV
PSV2 51	00	Expression result to be assigned to the pseudo-variable.	Null.	Pseudo-variable argument.	Pseudovariable table used for non-string pseudovariables, e.g., STATUS.		OX
PTS DA	00	Pointer.	Pointer.	Pointer temporary (may be qualified).	Used when op2 is itself a locator. Becomes BADDR in PI.		PI
PTSAT F9	00	Pointer.	Based variable	Q-temp. (alive).	Pointer text table Becomes BADDR in PI.		
PUT 8A	00	File or STRING.	SKIP (expression).	LINE (expression).			
READ 83	00				See last part of this figure.		KM
REPLY C7	00	Null.	Null.	Reply expression.		KT	II
RESDES 31	00	Based variable (REFER structure).		Descriptor temporary	Used to reserve space for a descriptor.	IA ID	
RETURN 7D	00	No of blocks to be epilogued.	Dict. ref. of last block to be epilogued.	RETURN expression result.	Operand 3 is null if nothing is returned.	II	
	01	No of block to be epilogued.	Dict. ref. of last block to be epilogued	RETURN expression result.	Return for interlanguage encompassing block.	IM	SA
REVERT 78	00	File reference if file condition.		Condition name and description of ON-unit.	REVERT statement.	II	
REWRITE 87	00				See last part of this figure.		KM

Figure 5.96. (Part 28 of 32). Usage of operands in Type-2 text tables

Operator		Operand 1	Operand 2	Operand 3	Comments	Phase	
IOP1	IOP2					Cr	D1
SCI 6B	00	Comparand.	Increment.	Loop control variable.	Increments a comparand at the head of an iterative do-loop. IOP2 00 for inner loops 01 for outer loops for loops with multiple specifications.	II	SQ
SHIFT 23	00	Operand to be shifted.	Byte 2 Meaning Type of shift. 5 Amount.	Target.	Not used if SIZE is possible.		SA
SIGNAL 77	00	Null.	Null.	Signal type.			
SINIT F3	00	Initializing constant.		Static variable to be initialized.	Static initial assignment		PA
SL FD	00				See figure 5.90.		SA
SN FC	00				See figure 5.90.		
STOP 75	00	Null.	Null.	Null.			KT
STPG 9C	00	Non-standard format containing overflow page reference.			Overflow page index table. See figure 5.3.		KA
SUBS F1	00	Subscript.	Array V-ref.	Index temporary.	Any subscript other than the first in a list. IST2 is as for the MULT X'00' table.	II	KE
	01	Subscript.	Literal binary constant.	Index temporary.	Any subscript other than the first in a list. GEN2 may be used for the last subscript in a list (see figure 5.94).	KE	QA

Figure 5.96. (Part 29 of 32). Usage of operands in Type-2 text tables

Operator		Operand 1	Operand 2	Operand 3	Comments	Phase	
IOP1	IOP2					Cr	Dl
SUBS1 F0	00	Subscript.	Array information: Byte / Meaning 0-1 / 2-3 / A.T. ref 4-5 / V-ref.	Index temporary.	First subscript in a list. The content of operand 2 after Phase KE is shown in figure 5.97. IST2 is set by Phase KE as for a MULT table and the table is changed to MULT by Phase QA.	II	QA
KONST F8	00	Translate table.			TRANSLATE bif. (TRT X'00' is always preceded by KONST X'00').	OC	SC
TRT A4	00	Source - non-varying string <=256 bytes.	Length.	Target - non-varying string.			
TRT A4	01	Source - varying string <=256 bytes.	Translate table.	Halfword binary target.	VERIFY bif.	OX	
	02	Source string.	Length (source) if adjustable, otherwise length (source)-1.	256 byte string (table).	Generates a translate and test table for a TRANSLATE bif.		
KONST F8	00	Translate table.			TRANSLATE bif. (TRT X'03' is always preceded by KONST X'00').	OC	
TRT A4	03	Source - non-varying string >256 bytes.	Length.	Target - non-varying string.			
KONST F8	00	Length (POS) if adjustable source string, otherwise length(POS)-1			Generates a translate and test table for a TRANSLATE bif. (TRT X'05' is always preceded by KONST X'00').		
TRT A4	05	String variable (POS).	String variable (replace).	256 byte string (table).			
KONST F8	00	Length of the source string.			VERIFY bif. (TRT X'06' is always preceded by KONST X'00').	OX	
TRT A4	06	Source - non-varying string <=256 bytes.	Translate table.	Halfword binary target.			

Figure 5.96. (Part 30 of 32). Usage of operands in Type-2 text tables

Operator		Operand 1	Operand 2	Operand 3	Comments	Phase	
IOP1	IOP2					Cr	Dl
TRT A4 (contd)	07	Source - varying string <=256 bytes.	Translate table.	Target - varying string longer than source.	TRANSLATE bif.	OC	SC
KONST F8	00	Translate table.			TRANSLATE bif. (TRT X'08' is always preceded by KONST X'00').		
TRT A4	08	Source - adjustable string<=256 bytes or varying string <=256 bytes.	Constant (1).	Target - varying string <=256 bytes or varying string shorter than source.			
KONST F8	00	Length of shorter string.	Translate table.		TRANSLATE bif. (TRT X'09' is always preceded by KONST X'00').		
TRT A4	09	Source string.	Temporary workspace, if source and target overlap.	Target string.			
KONST F8	00	Translate table.			VERIFY bif. (TRT X'0A' is always preceded by KONST X'00').	OX	
TRT A4	0A	Source - non-varying string>256 bytes	Length of source string.	Halfword binary target.			
TRT A4	0B	Source-fixed length.	Source-fixed length<257.	Target.	Generates code for INDEX bif.		
	0C	Source - any length, adjustable, varying, or fixed.	Source <256. May be varying.	Target.			
	0D	Source - any length, varying.	Source - fixed <256.	Target.			
	0E	String temporary.			Generates code to create a translate table.	OC	
VDA 22	00	Size of storage needed (absolute value or variable).	Rounding constant for doubleword alignment of VDAs.	Target for address, if needed.	Get VDA table. IST2 is set, by ORing with SETPNAB, if the VDA must be kept until termination of the block.	II	SA
	02				Free VDA table. Only used if VDA to be freed before end of statement.		

Figure 5.96. (Part 31 of 32). Usage of operands in Type-2 text tables

Operator		Operand 1	Operand 2	Operand 3	Comments	Phase	
IOP1	IOP2					Cr	Dl
WAIT 72	00	Null.	Null.	Event expression.		II	KT
WHILE 6D	00	WHILE expression.	GSL of end of do-loop.		WHILE clause.		
WRITE 84	00				See last part of this figure		
Record I/O text tables The general format of record I/O tables (listed below) is as follows: File. Key (variable, constant, or expression). Variable or IGNORE expression. The options permitted with each type of statement are shown against the corresponding values of IOP2. All the text tables except LOCATE are followed by an EVO table.							
UNLOCK 85	00		KEY		(Applies to OS version of compiler only.)	II	
READ 83	01			SET.			KM
	02		KEYTO.	SET.			
	03		KEY.	SET.			
	04			INTO.			
	05			INTO.	EVENT.		
	06		KEYTO.	INTO.			
	07		KEYTO.	INTO.	EVENT.		
	08		KEY.	INTO.			
	09		KEY.	INTO.	EVENT.		
	0A		KEY.	INTO.	UNLOCK) OS		
	0B		KEY.	INTO.	EVENT. UNLOCK) only.		
	0C			IGNORE.			
	0D			IGNORE.	EVENT.		
WRITE 84	0E			FROM.			
	0F			FROM.			
	10		KEYFROM.	FROM.			
	11		KEYFROM.	FROM.	EVENT.		
REWRITE 87	12			FROM.			
	13			FROM.	EVENT.		
	14		KEY.	FROM.			
	15		KEY.	FROM.	EVENT.		
DELETE 85	16				EVENT. (Applies to OS version of compiler only.)		
	17		KEY.				
	18		KEY.		EVENT.		
LOCATE 88	19			SET.	Always followed by a LOCATE 00 table.		
	1A		KEYFROM.	SET.			

Figure 5.96. (Part 32 of 32). Usage of operands in Type-2 text tables

Type of array	Contents of Operand 2			
	Byte 0	Byte 1	Bytes 2-3	Bytes 4-5
Adjustable non-based.	Code byte (variable).	Data byte.	Offset.	ATR.
Adjustable based.	Code temporary (Q-temp).	Data byte.	Offset.	QT number.
Non-adjustable.	Code byte (constant).	Precision.	Literal binary constant.	

Figure 5.97. Content of Operand 2 of a SUBS1 table after Phase KE

Bytes	Symbol	Meaning
0	IFOM	Text code byte (FLWUNT).
1	IFOFL	First flag byte: 1... .... Drop through from this unit to the next in the stream. .1.. .... The first label (see below) exists. ..1. .... The second label (see below) exists. ...1 .... The unit ends at a CALL statement. .... 1... The unit ends at a RETURN statement. .... .1.. This is the last flow unit in the block. .... ..1. The flow unit starts an iterative do-loop.
2	IFOFL2	Second flag byte: 1... .... The unit starts at block entry statement (PROC,BEGIN,ENTRY)
3		Not used.

		Label constant	Label variable temporary	Label variable or pointer	First label descriptor
4		IFOC1 - operand code byte.	IFOC1 - operand code byte.	IFOC1 - operand code byte.	
5-6		IFORF1 - label number.	IFORF1 - label number of first compiler label.	IFORF1 - dictionary reference of the variable.	
7		IFOLV1 - block level.	IFOCL1 - number of compiler labels.	IFOLF1 - flag byte (X'80' if	
8		IFOCT1 - block count.		GOOB, X'00' if GOTO).	
9-13		Second label descriptor - as above, with symbolic names ending in 2.			
14-18	IFOUNC	Flow unit chain field.			
19-23	IFOBCH	Block chain field.			
24-26		Not used.			
27-28	IFOPH1	Two-byte reference of last hash table in preceding flow-unit.			
29	IFOCDE	For first flow unit in block: block count.		Otherwise: operand code byte of label on this flow unit.	
30-31	IFOLAB	For first flow unit in block: the number of the first label used in a GOTO statement in the block.		Otherwise: dictionary reference of label on this flow unit.	
32		Not used.			

Figure 5.98. (Part 1 of 2). Format of flow unit headers from Phase OE to Phase OI

Bytes	Symbol	Meaning
33-35	IFOLPU	Track address of the last page in the flow unit.
36-67	IFOVU	Bit vector indicating which of the first 256 variables are used in this flow unit.
68-99	IFOVS	Bit vector indicating which of the first 256 variables are set in this flow unit.
100-131	IFOBEN	Bit vector indicating EITHER which of the first 256 variables are busy on entry to the flow unit, (Phase OE), OR which of the first 256 variables are set between the head and back dominator of the flow unit.
132-163	IFOBEX	Bit vector indicating EITHER which of the first 256 variables are busy on exit from the flow unit, OR which of the first 256 variables are set more than once in the flow unit (Phase OE).

Figure 5.98. (Part 2 of 2). Format of flow unit headers from Phase OE to Phase OI

Bytes	Symbol	Meaning
0	IFOM	Text code byte (FLWUNT).
1	IFOF1	First flag byte:
	1...	The first connection (see below) exists.
	.1..	The second connection (see below) exists.
	..1.	First connection is by 'drop through'.
	...1	The unit ends at a CALL statement.
	.... 1...	There are no more connections. (Depends on second connection existing.)
	.... .1..	
	.... ..1.	The unit can never be executed.
	1	This flow unit has been optimized.
2	IFOF2	Second flag byte:
	1...	The flow unit is always executed in a loop.
	.1..	The flow unit is in a non-looping part of the program.
	..1.	The flow unit is the first in a loop (loop entry).
	...1	The flow unit is the last in a loop.
	.... 1...	All flow units on chain back to back dominator are in a loop.
	.... .1..	
	.... ..1.	The flow unit is a back target.
	1	Re-ordering is permitted in this block.
3	IFOF3	Third flag byte:
	1...	The label in this flow unit can be gone to from an ON-unit or is a stream I/O abnormal termination label.
	.1..	The first flow unit in a block.
	..1.	The first flow unit in a DO group.
	...1	The last flow unit in a DO group.
	.... 1...	The flow unit header is to be retained.
	.... .1..	The loop contains a hidden branch (e.g., as a result of ENDFILE) and the loop control variable must be stored.
4	IFOFUN	Flow unit number.
5	IFOBD	Back dominator number (if flow unit is in a loop).
6	IFOLC	Loop chain field.
7	IFOBTN	Back target number (if flow unit is not in a loop).
8	IFOCN1	First connection: flow unit to which control passes on exit.
9	IFOCN2	Second connection: alternative to the first connection.
10-11	Not used.	

Figure 5.99. (Part 1 of 2). Format of flow unit headers after Phase OI

Bytes	Symbol	Meaning
12-13	IFOLDE	Dictionary reference to loop data entry.
14-18	IFOLCR	Text reference of the next loop in the loop chain.
19-23	IFOBDR	Text reference of the back dominator.
24-28	IFOBTNR	Text reference of the back target.
29		Not used.
30-31	IFOPC	Number of back connectors.
32	IFODN	Loop depth number.
33-35	IFOLPU	Track address of the last page in the flow unit.

Figure 5.99. (Part 2 of 2). Format of flow unit headers after Phase OI

Bytes	Symbol	Meaning
0	IOP1	Text code byte (HASH).
1	IOP2	Second byte of operator.
2-23	IHASHC	Eleven two-byte hash-chain fields.
24-26	IBPCH	Chain field pointing back to the previous page.
27		Not used.
28-29	IBCHN	Head of chain back through text tables in this flow unit or page.
30-31	IFCHN	Head of chain forwards through text tables in this flow unit or page.

Figure 5.100. Format of hash tables in Type 2 text

PSEUDO CONSTANTS POOL (PCP)

A pseudo constants pool entry consists of a six-byte heading field, followed by a 'constant' field from which the object-time constant will eventually be constructed (during storage allocation). In some cases, e.g., literal source-program constants with no replication factor, the 'constant' part of the PCP entry is identical to the final object-time constants pool entry.

The general format of PCP entries is as follows:

Bytes	Symbol	Meaning
0	PCPFGL	PCP flag byte.
1	PCPCDE	PCP code byte, indicating type of entry.
2-3	PCPCPL	Length of constants pool entry (i.e., expanded length).
4-5	PCPDRF	Dictionary reference of variables dictionary entry.
		OR
	PCPREP	Replication factor if the PCP entry is for a literal constant.
6-	PCPENT	Constant (not necessarily in its final form).

Figure 5.101. General format of pseudo constants pool entries

The PCP is constructed by Phases PC and PA. The types of entry made by each of these phases are described in Figures 5.103 to 5.123, as follows:

Figure 5.103. Format of object-time arithmetic DEDs
Figure 5.104. Format of non-pictured arithmetic FEDs (E or F format)
Figure 5.105. Format of pictured arithmetic FEDs
Figure 5.106. Format of non-pictured string DEDs and FEDs
Figure 5.107. Format of pictured string DEDs and FEDs
Figure 5.108. Format of C-format FEDs
FIGURE 5.109. Format of carriage-control FEDs
Figure 5.110. Format of program control data DEDs
Figure 5.111. Code bytes in object-time DEDs and FEDs
Figure 5.112. Flag bytes in object-time DEDs and FEDs
Figure 5.113. Format of symbol table list element entries
Figure 5.114. Format of long symbol tables
Figure 5.115. Format of short symbol tables
Figure 5.116. Format of string locator/descriptors
Figure 5.117. Format of string descriptors
Figure 5.118. Format of aggregate locators
Figure 5.119. Format of area locator/descriptors
Figure 5.120. Format of array descriptors
Figure 5.121. Format of structure descriptors
Figure 5.122. Format of descriptor descriptors for structures
Figure 5.123. Format of descriptor descriptors for base elements of aggregates

Output to the PCP from Phase PC

Object-time DEDs and FEDs which are required to be passed as arguments to library routines. The formats of these entries are shown in figures 5.103 to 5.110. The values of the DED code bytes are shown in figure 5.111, and the meanings of the flag bytes (DEDFLG) in figure 5.112.

Symbol table lists: A symbol table list ('symbol table vector') is created for each block containing declarations of variables which are involved in DATA-directed I/O or CHECK condition statements. Blocks with a lower block count which do not contain any such declarations are nevertheless represented with dummy lists, for chaining purposes. A dummy list is similar to the last two elements of a normal list and contains a list delimiter (X'00000000') and a pointer to the first element of the list of the containing block. The lists are built in block-count order. The format of a symbol table list element entry is shown in figure 5.113.

Symbol tables are created for variables that are known within a block containing either a 'GET/PUT DATA;' STATEMENT, OR A 'SIGNAL CHECK;' statement (for those variables for which the CHECK condition is enabled). Each symbol table is referenced (at compile-time by a two-byte offset; at object-time by a four-byte address) by its corresponding element in the symbol table list for the block in which the variable is declared.

The formats of symbol tables are shown in figures 5.114 and 5.115.

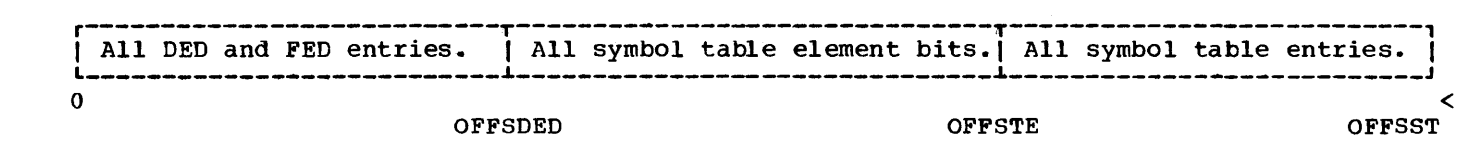


Figure 5.102. Contents of the PCP after Phase PC

Bytes	Symbol	Meaning	
0	DEDLKP	DED code byte, indicating type of DED. See figure 5.111.	
1	DEDFLG	Flag byte, indicating the attributes of the item:	
	..0.	Fixed constant.	
	..1.	Float constant.	
	...0	Non-extended float.	
	...1	Extended float.	
	.... 0...	Fixed picture.	
	.... 1...	Float picture.	
	.... .0..	Declared binary.	
	.... .1..	Declared decimal.	
	.... ..0.	Short precision.	
	.... ..1.	Long precision.	
	0	Real.	
	1	Complex.	
2	DEDPRS	Precision.	
3	DEDSCL	Scale.	
4	PADPLN	Length of picture.	This part is present only if the DED is pictured.
5	PADDLN	Length of data.	
6	PADMFL	Mantissa flag byte:	
	01.. ...0	Drifting s in subfield.	
	0.1. ...0	Drifting + in subfield.	
	0..1 ...0	Drifting - in subfield.	
	0... 1..0	Drifting $ in subfield.	
	0... .1.0	Total suppression in subfield.	
	0... ..10	* in subfield.	
7	PADEFL	Exponent flag byte - as for PADMFL.	
8-	PADPIC	Picutre specification.	

Figure 5.103. Format of object-time arithmetic DEDs

Bytes	Symbol	Meaning
0	DEDLKUP	DED code byte, indicating type of FED. See figure 5.111.
1	DEDFLG	Flag byte, indicating the attributes of the field. See DEDFLG in figure 5.103.
2-3		Field width.
4		Precision.
5		Scale.

Figure 5.104. Format of non-pictured arithmetic FEDs (E- or F-format)

Bytes	Symbol	Meaning	
0	DEDLKUP	DED code byte, indicating type of FED. See figure 5.111.	
1	DEDFLG	Flag byte. See figure 5.112.	
2-3	PDFPL2	Field width.	
4	PDFLK2	DED code byte.	This part of the FED is identical to the corresponding arithmetic DED. See figure 5.103.
5	PDFFL2	Flag byte.	
6	PDFPRS	Precision.	
7	PDFSCL	Scale	
8	PDFPLN	Length of picture.	
9	PDFDLN	Length of data.	
10	PDFMFL	Mantissa flag.	
11	PDFEFL	Exponent flag.	
12-	PDFPIC	Picture specification.	

Figure 5.105. Format of pictured arithmetic FEDs

Bytes	Symbol	Meaning
0	DEDLKUP	DED code byte, indicating type of DED/FED. See figure 5.111.
1	DEDFLG	Flag byte. See figure 5.112.
2-3	DEDLNG	String length (if DED), or field width (if FED).

Figure 5.106. Format of non-pictured string DEDs and FEDs

Bytes	Symbol	Meaning
0	DEDLKUP	DED code byte, indicating type of DED/FED. See figure 5.108.
1	DEDFLG	Flag byte. See figure 5.109.
2-3	PSTPIL	Length, or field width, with insert characters.
4-5	PSTPNL	Length, or field width, without insert characters.
6-	PSTPIC	Picture specification.

Figure 5.107. Format of pictured string DEDs and FEDs

Bytes	Symbol	Meaning
0	DEDLKUP	DED code byte (X'5C'), indicating type of FED.
1	DEDFLG	Flag byte. See figure 5.112.
2-3		Field width.
4-		FED for real part (bit 7 of the flag byte = '1'B). FED for imaginary part (bit 7 of the flag byte = '1'B).

Figure 5.108. Format of C-format FEDs

Bytes	Symbol	Meaning
0	DEDLKUP	DED code byte, indicating type of FED. See figure 5.111.
1	DEDFLG	Not used.
2-3		This field is omitted for a PAGE FED, and used as follows for other carriage-control FEDs: width of item - X format item column number - COL format item number of lines to be skipped - SKIP format item line number - LINE format item

Figure 5.109. Format of carriage-control FEDs

Bytes	Symbol	Meaning
0	DEDLKUP	DED code byte, indicating type of DED. See figure 5.111.
1	DEDFLG	Flag byte. See figure 5.112.

Figure 5.110. Format of program control data DEDs

Hex value	Data type (problem data DED)
00	FIXED BINARY.
04	FIXED DECIMAL.
08	FLOAT.
0C	Free decimal (internal form).
10	FIXED PICTURE BINARY.
14	FIXED PICTURE DECIMAL.
18	FLOAT PICTURE BINARY.
1C	FLOAT PICTURE DECIMAL.
20	Non-VARYING CHAR.
24	Non-VARYING BIT.
28	VARYING CHAR.
2C	VARYING BIT.
30	CHAR PICTURE.
40	BINARY constant.
44	DECIMAL constant.
48	BIT constant.

Hex value	Data type (FED and program control data DED)
50	F or E format.
54	Arithmetic P format.
58	Character A, B, or P format.
5C	C format.
60	X format.
64	COL format.
68	SKIP format.
6C	LINE format.
70	PAGE format.
80	LABEL.
84	ENTRY.
88	AREA.
8C	TASK.
90	OFFSET.
94	POINTER.
98	FILE.
9C	EVENT.

Figure 5.111. Code bytes in object-time DEDs and FEDs

Bit setting	Meaning
00..	A-format item.
01..	B-format item.
10..	Character picture format item.
..0.	Fixed constant.
..1.	Float constant
...1	Extended float.
.... 0...	F-format item/fixed picture.
.... 1...	E-format item/float picture.
.... .0..	Declared binary.
.... .1..	Declared decimal.
.... 11..	Character.
.... ..0.	Short precision.
.... ..1.	Long precision.
....0	Real/length specified (A or B format)/unaligned bit string.
....1	Complex/no length specified (A or B format/aligned bit string.

Figure 5.112. Flag bytes in object-time DEDs and FEDs

Bytes	Symbol	Meaning
0	STEFLG	Flag byte:
	0000 0000	Not the last entry for a block.
	1000 0000	Last entry in the list for the first block.
	1100 0000	Last entry in the list for a block other than the first.
1		Not used (zero).
2-3	STEOFF	Pointer to a symbol table entry for an identifier, OR zero (to indicate end of list for block) if this is the last-but-one OR pointer to the first entry for the preceding block, if this is the last entry for a block.

Figure 5.113. Format of symbol table list element entries

Bytes	Symbol	Meaning
0	SYTFL1	Flag byte 1:
	000.	Identifier is STATIC.
	100.	Identifier is AUTOMATIC.
	010.	Identifier is CONTROLLED (non-parameter).
	001.	Identifier is BASED.
	011.	Identifier is DEFINED.
	101.	Identifier is non-CONTROLLED parameter.
	111.	Identifier is CONTROLLED parameter.
	...0	Identifier is INTERNAL.
	...1	Identifier is EXTERNAL.
	.... 1...	Identifier may appear in a CHECK list (this bit must be cn if the identifier is EXTERNAL).
	.... .1..	Field A (SYTFLA) refers to data.
	.... .0..	Field A (SYTFLA) refers to locator (this bit is zero for CONTROLLED parameter).
	.... ..1.	Identifier is a member of a structure.
	1	This is a long symbol table.
	0	This is a short symbol table. See figure 5.115.
1	SYTFL2	Flag byte 2:
	1...	Field A (SYTFLA) addresses code.
	0...	Field A (SYTFLA) does not address code.
	.1..	The item is dynamically CHECKED currently (this bit can only be set if the table is not in STATIC).
	..1.	The word preceding this table (at object-time) contains the value used to establish ON CHECK for this variable.
	..0.	This word does not exist.
	.... .1..	Variable is based; bits 0, 1, 2, 5 of SYTFL1, bit 0 of SYTFL2, level #, and flags all refer to declared pointer.
	.... ..1.	Variable is based and SYSTFLB contains descriptor's address in static storage.
	.... ..0.	If variable is based, SYTFLB contains descriptor offset in DSA.
	...0 0000	Reserved bits, always set to zero.
2	SYTDIM	Dimensionability (zero if non-array).
3	SYTLVL	Block level, if AUTOMATIC.
4-7	SYTADD	Address of the associated DED.
8-11	SYTFLA	Field A - address of data or locator.

Figure 5.114. (Part 1 of 2). Format of long symbol tables

Bytes	Symbol	Meaning
12-15	SYTFLB	Field B - address needed for BASED.
16-17	SYTIDL	Length of identifier's name.
18-	SYTBCD	Fully qualified identifier name.

Figure 5.114. (Part 2 of 2). Format of long symbol tables

Bytes	Symbol	Meaning
0	SYTFL1	Flag byte 1. See SYTFLG1 in figure 5.114.
1	SYTFL2	Flag byte 2. See SYTFLG2 figure 5.114.
2-3	SYTIDL	Length of identifier's name.
4-	SYTBCD	Fully qualified identifier name.

Figure 5.115. Format of short symbol tables

Output to the PCP from Phase PA

Entries for constants (source-program, compiler-generated, address, and label) and I/O control blocks (FCB, record descriptor, key descriptor, ENVB, and DTF) are constructed from the corresponding general dictionary entries. In these cases, the 'constant' part of the entry, PCPENT, (see figure 5.101) is the same as the 'constant' part of the dictionary entry. The formats of PCP entries for static locators, static descriptors, and descriptor descriptors are shown in figures 5.116 to 5.123.

Bytes	Meaning
0-3	Byte address of the string.
4-7	String descriptor, as described below.

Figure 5.116. Format of string locator/descriptors

Bytes	Meaning
0-1	Allocated length of the string.
2	Flag byte: X'00' - fixed length string. X'80' - VARYING string.
3	For UNALIGNED BIT string, the last three bits of this byte contain the bit offset of the string from the byte address.

Figure 5.117. Format of string descriptors

Bytes	Meaning
0-3	Bytes address of the aggregate.
4-7	Address of the aggregate descriptor. (See figures 5.121 and 5.122.)

Figure 5.118. Format of aggregate locators

Bytes	Meaning
0-3	Byte address of the AREA.
4-7	Allocated length of the AREA. This is the area descriptor, and is concatenated to the end of tne array descriptor for an array of AREAs.

Figure 5.119. Format of area locator/descriptors

Bytes	Meaning
0-3	Byte address of the start of the array, relative to its 'virtual origin'.
4-11	Extent descriptor for the first dimension. See figure 5.11.
12-	Extent descriptors for all other dimensions of the array.

Figure 5.120. Format of array descriptors

Bytes	Meaning	
0-3	Byte address of the element relative to the start of the structure descriptor	This is repeated for each base element of the structure, in order of declaration. The whole makes up the structure descriptor.
4-7	Descriptor for the element.	

Figure 5.121. Format of structure descriptors

Bytes	Bits	Meaning	
0-1	0-1	Zero, indicating that this a structure	This is repeated for each identifier declared in the aggregate.
	2-15	Offset, from the start of the descriptor, of this entry in the descriptor. The offset is in multiples of four bytes.	
2	0-7	Logical level of the identifier in the structure.	
3	0	'1'B indicates AREA.	
	1	'1'B indicates BIT string.	
	2-7	Real dimensionality of the identifier.	

Figure 5.122. Format of descriptor descriptors for structures

Bytes	Bits	Meaning	
0	0	'1'B indicates that this is a base element,	This is repeated for each identifier declared in the aggregate.
	1	'1'B indicates that this is the last element in the structure.	
	2-7	Alignment stringency for the element, as follows: 0 bit 7 byte 15 halfword 31 word 63 doubleword	
1	0-7	Length in bytes of the item. This is set to zero for AREA or string identifiers.	
2-3		As for bytes 2 and 3 in figures 5.122.	

Figure 5.123. Format of descriptor descriptors for base elements of aggregates

EXTENDED CODE FORMATS

During the code generation stage, the text stream contains a mixture of markers, Type-2 text tables, generated code, and padding bytes. Unprocessed Type-2 text tables are always preceded by special four-byte markers (inserted by Phase SA), and are word-aligned. All other items in the text steam are byte-aligned.

Bytes	Meaning	
0	X'0F'.	Marker.
1	X'00'.	
2	X'00'.	
3	X'00'.	
4-	Text table.	

0	X'0F'.	Marker.
1	TYP. Type of marker, indicating the contents of the remainder of the marker. See figure 5.125 for details.	
2	TXT. Length of marker + associated code.	
3	COD. Length of generated code associated with this marker, or zero.	
4-	Remainder of marker depends on value of TYP.	

0	Code byte.	Generated code contains instructions whose length is indicated by the first two bits of the operation code.
1-	Remainder of instruction, 1, 3, or 5 bytes in length.	

	X'0E'. Padding byte.

Figure 5.124. Components of extended code

Marker Type	TYP	TXT	COD	Bytes 4 & 5	Byte 6 (see Notes 2 & 4)	Byte 7	Other bytes
Label	1	8	0	Dictionary reference, or compiler-label number.	label flags.	-	
Branches - unconditional or conditional. (See Note 3)	2 3	14 14	4 4	Dictionary reference, or compiler-label.	label flags.	-	Bytes 8-15 are X'0E', byte 16 is the operation code, byte 17 is the operation mask.
End of text table marker.	4	4	0	Used for intermediate processing within code production phases.			
Prologue load address.	5	14	4	Dictionary reference, or compiler-label number.	label flags.	-	
Procedure.	7	8	0	Dictionary reference of block.	label flags.	-	
Begin.	8	8	0	Dictionary reference of block.	label flags.	-	
Prologue.	9	8	0	Compiler label number of start of prologue.	label flags.	-	
Procedure.	A	8	0	Compiler label number of procedure base.	label flags.	-	
Statement number.	B	8	0	Statement number.		-	
On block.	C	8	0	Dictionary reference of block.	label flags.	-	
Block end.	D	A	0			-	Bytes 8 and 9 contain a NOOP 0707.
No listing information - N=number of bytes of code following.	10	8+N	N	Value N is repeated in this half-word.		-	Bytes 8 to 8+N-1 contain the code.
Internal text table label.	11	8	0	Internal label number.		-	

Figure 5.125. (Part 1 of 3). Markers inserted in extended code by code generation phases

Marker Type	TYP	TXT	COD	Bytes 4 & 5	Byte 6 (see Notes 2 & 4)	Byte 7	Other bytes
Internal text table branches; unconditional or conditional.	12 13	8 8	4 4	Internal label number of branch label.		-	Byte 8 is the operation code. Byte 9 is the branch mask.
Program end.	14	8	0	Size of program.			
Statement change.	15	8	0	Statement number.		-	
BXLE and BXH branches. See Note 3.	16	1A	4	Dictionary reference or compiler-label number.	label flags.	-	Bytes 8-15 are X'0B', byte 16 is the operation code, byte 17 is the branch mask.
Call.	17	8	0	Dictionary reference of entry label.	label flags.	-	
Constant.	18	8+N	N	N repeated.	con-stant flags.	-	
Constant requiring relocation.	19	8+N	N	N repeated.	con-stant flags.	Reloc-ation CSECT.	
Real entry point.	1A	8	0	Dictionary reference of entry.	label flags.	-	
Start compiler-generated subroutine.	1B	6	0	Number of in-line module.		-	
Loop initialization.	1C	8	0			-	
'Code moved' marker	1D	9	0	Statement number from where code came.		-	7, 8 - offset of statement marker.
'Code command' marker.	1E	8	0			-	
'Code reordered' marker.	1F	8	0			-	
'Number region' marker.	20	8	0	Register for program base.		-	

Figure 5.125. (Part 2 of 3). Markers inserted in extended code by code generation phases

Marker Type	TYP	TXT	COD	Bytes 4 & 5	Byte 7 (see Notes 2 & 4)		Other bytes
End compiler-generated sub-routine.	21	4	0	Number of sub-routine.		-	
End of program CSECT.	22	8	0			-	
Load adcon.	23	C	0	Library entry name.		-	
'Code revert' marker.	24	9	0	Statement number re-verted to.		-	7,8 - offset of statement marker.
'Common ended' marker.	25	8	0			-	
Pseudo register relocation.	26	8	0	Dictionary reference of controlled variable.		-	

Figure 5.125. (Part 3 of 3). Markers inserted in extended code by code generation phases

Notes:

1. Byte 0 is omitted because it is always X'0F'. The values of Type Code, TXT, and COD bytes are shown in hexadecimal.

2. The label flags are set as follows:

   Bit

   0 Programmer-supplied label.
   1 Compiler-generated label.
   2 Programmer-supplied subscripted label.

3. If an RR instruction is involved, the values shown for TXT and COD are halved.

4. The constant flags are set as follows:

   Bit

   0 Character constant.
   1 F type constant.
   2 H type constant.
   3 X type constant.
   4 A type constant.
   7 Static-initial constant.

PREPROCESSOR DICTIONARY ENTRIES

During the first preprocessor scan of text, entries are made in the general dictionary for all identifiers involved in compile-time statements. The format is shown below in figure 5.126.

During the replacement activity of the second scan, IVB (identifier value block) entries holding character string values for variables and intermediate text are made in the variables dictionary. The format of these entries is shown in figure 5.127.

Bytes	Symbol	Meaning
0		Length of the identifier's name. This is zero if the identifier is a constant.
1		Compile-time procedure number of the procedure in which the identifier is declared. Non-procedural text is procedure number 1.
2-3		Hash chain field, containing the dictionary reference of the next entry in the chain. The end of a chain is indicated by zero in this field.
4		Flag byte indicating various attributes of the identifier:
	1...	Fixed decimal.
	.1..	Character.
	..1.	Bit.
	...1	Entry.
	.... 1...	Label.
	.... .1..	INCLUDE identifier.
	.... ..1.	Iterative DO.
	1	Constant.
5-9	VALUE	Field containing either a literal value, or a pointer to an IVB entry.
	NNN..	Five-digit (three-byte) packed decimal number in the leftmost three bytes, if the identifier is FIXED.
	.DD..	Two-byte dictionary reference to an IVB entry in the variables dictionary, if the identifier is CHAR or BIT.
	TTTTT	Five-byte text reference to the text containing the procedure or label, if the identifier is PROCEDURE or LABEL.
	TTTTT	Five-byte text reference to the start of the included text, if this is an INCLUDE entry.
	.DD..	Two-byte dictionary reference to another entry for the variable, derived from its declaration in the non-procedural text. This is used in the case of an entry made for a variable which is used but not declared in compile-time procedure, and the entry's type is indicated by an 'indirect reference bit' in byte 17.
10-11	COUNT	For a PROCEDURE entry, this field contains the number of parameters for the procedure. For an INCLUDE entry, it contains the initial line number assigned to the included text after it has been included. For a parameter to a procedure, it contains the position of the parameter.

Figure 5.126. (Part 1 of 2). Preprocessor general dictionary entries

Bytes	Symbol	Meaning
12-15		During the first scan, this field is used to join labels and simple variables by forward and backward chains, using dictionary references as pointers. This enables a final check to be made for undeclared items. The information is not required by the second scan, which uses this field as follows: For a LABEL, the field contains two dictionary references; that for the containing iterative DO statement, and that for the containing INCLUDE statement. For an INCLUDE entry, the dictionary references of the containing INCLUDE statement is in bytes 14 and 15.
16		Flag byte:
	1...	Special entry.
	.1..	DECLARE encountered in the first scan.
	..1.	Procedure body encountered in first scan.
	...1	Parameter.
	.... 1...	Procedure called during second scan of a compile-time text.
	.... ..1.	Identifier used on left-hand side of an argument.
	1	Activated variable.
17		Flag byte:
	1...	Procedure 'in use' bit, for recursive checking.
	.1..	Indirect reference bit (see bytes 5-9).
	..1.	Undefined or multiply-defined variable.
	.... 1...	Built-in function bit. (Also used for 'explicitly set' bit.)
	.... ..1.	Activated with NORESCAN.
18-		Name item, in EBCDIC.

Figure 5.126. (Part 2 of 2). Preprocessor general dictionary entries

Bytes	Symbol	Meaning
0		Back-up character i.e., last character in the IVB preceding this one in the chain, if such an IVB exists.
1-35		Main data part of the IVB.
36		Length of the IVB data (maximum 35) in bits 2-7. Bit 0 = '1' if the IVB is the last in the chain.
37		Not used.
38-39		Chain field, containing the dictionary reference of the next IVB entry in the chain. If bit 0 of byte 36 = '1', then this field is not used.

Figure 5.127. Identifier value block (IVB) entries in the preprocesscr variables dictionary

Section 6: Diagnostic Aids

INTRODUCTION

This section contains information that is intended to assist in the recognition, identificiation, and location of errors in the compiler program. In particular it describes how various aids can be made available.

(Aids for diagnosing errors in source programs and object programs are described in the publications: DOS PL/I Optimizing Compiler: Programmer's Guide and, DOS PL/I Optimizing Compiler, Execution Logic.

USE OF THE COMPILER DUMP OPTION

The compiler dump phase (Phase AI) can be used to provide a printed dump of the contents of various data areas used during a compilation. In general, the printed dump shows the hexadecimal values of the data area contents. In order to obtain a dump, the DUMP option must be specified in the *PROCESS statement.

A dump can be obtained after the execution of one or more processing phases, when it is referred to as an interphase dump, or in the case of a program-check interrupt causing abnormal termination of the execution of a phase, when it is referred to as an abort dump (In the case of an abort dump, the number of the statement being processed at the time of the abnormal termination can be found by examination of the XSTAT field (offset X '670') of the communication area (see figure 5.1., part 10).)

The contents of the following data areas can be dumped:

Registers (not applicable to interphase dumps)

Communications area (XCOMM)

Phase working storage (XSTG)

Text pages

Dictionary pages

In addition to these data areas, a dump can also contain the following additional information:

- Page header information - the page headers of all pages in main storage at the time of the dump, plus the first 16-bytes of the page data.
- Page monitoring variables - information about the number of various page-handling operations performed during the compilation up to the time of the dump.
- Special areas of phase working storage - if a compiler abort dump is provided during execution of Phase QA, a formatted listing of the contents of the current register usage table is provided (in addition to this data being dumped in unformatted form as part of the contents of XSTG).

The compiler dump phase can be invoked by inclusion of the keyword, DUMP, or its abbreviation, DU, in the *PROCESS statement. The option can be specified with or without a value list. The formats, and the facilities provided, are described in the following paragraphs.

Note: If the facilities provided by the compiler dump phase are required during compilation of one or more external procedures that are members of a batched job or job-step, the DUMP option must be specified in the *PROCESS statement of the first member of the batch as well as in the *PROCESS statement(s) of the batch member(s) for which the facilities are required. This is because the dump phase remains in main storage throughout compilation and allowance for it must be made during initialization of the compiler partition. The dump phase requires approximately 16K bytes of main storage.

USE OF THE DUMP OPTION WITHOUT A VALUE LIST (COMPILER ABORT DUMP)

The format of the *PROCESS statement when the DUMP option is used without a value list is as follows:

```
*[ ]PROCESS [options,] DUMP|DU [options]
```

Commas between options can be omitted, provided that the options are separated by at least one blank.

If the DUMP option is specified without a value list, a dump will only be provided in the case of abnormal termination. Such a dump will have the header:

```
PL/I OPTIMIZER - DUMP ROUTINE
```

and the sub-header:

```
DUMP ON COMPILER ABORT DURING PHASE xx..
```

The dump will include the data described below.

1. Page header information. This section contains information about each of the data pages resident in main storage at the time of the dump. This information consists of:

   a. a three-byte field showing the address of the page.

   b. four 4-byte fields showing the contents of the page header (see part 1 of figure 5.2).

   c. four 4-byte fields showing the contents of the first 16 bytes of processable data in the page.

2. Registers. The contents of the general registers are printed.

3. Communications area. The contents of XCOMM are printed.

4. Page monitoring variables. Information about the number of various page-handling operations performed up to the time of the dump is listed as follows:

   ```
   No. of new text page requests.
   No. of DISCARDED text pages.
   No. of new dictionary page requests.
   No. of existing text page requests.
   No. of existing dictionary page requests.
   No. of existing text pages requiring I/O operations.
   No. of dictionary pages requiring I/O operations.
   No. of I/O operations satisfied by SPILLABLE pages.
   No. of I/O operations satisfied by USABLE pages.
   No. of I/O operations satisfied by DISCARDED pages.
  *No.  of READ/WRITE pages not written on.
  *No.  of READ ONLY pages not written on.
   Total No. of I/O operations.
   ```

   Note: The values given for items marked * have no significance in the published version of the compiler owing to the fact that more I/O operations are required to obtain them. Modification and reassembly of Phase AA is required to obtain significant values.

5. Phase working storage - XSTG. A subheading is printed to show the start and end addresses of XSTG for the phase. The contents of XSTG are listed as lines of eight 4-byte fields, each line preceded by a three-byte address indicating the offset from the origin of XSTG of the first item in the line.

6. Dictionary sections. When the DUMP option is specified without a value list, a formatted dump of the dictionary sections is provided only if the terminated phase uses the dictionary. The heading, "DUMP OF DICTIONARY", is followed by sub-headings

for each section of the dictionary, in the order: general dictionary, names dictionary, variables dictionary, and storage dictionary.

Following each section heading, the contents of the whole of that dictionary section are listed. The left-hand columns show the reference of each entry. Other columns show the contents of each entry, starting with the code byte which identifies the type of the entry. The fields within the entry are separated by blanks, and each column representing a field has a heading indicating the symbolic name of the field. Where a dictionary section contains a number of different types of entry, a new line of headings is printed each time a new type of entry is encountered. Where a type of entry has no fixed format, the heading idnciates the type of entry, e.g., Overflow entry. Each names dictionary entry shows the EBCDIC representation of the name itself.

If it is impossible for Phase AI to provide a formatted dump, or to complete the printing of a formatted dump, a message:

"*ERROR IN SCAN. UNFORMATTED DUMP FOLLOWS".

is printed. The start and end addresses of the unformatted dump are then printed. Each line of an unformatted dump starts at an address that is a multiple of X'20' and, in the case of an interrupted formatted dump, will start at the appropriate address preceding the page currently being dumped. Thus an unformatted dump may contain data already printed in formatted image, the page header, and a few bytes of data contained in main storage preceding the start of the current page. When dumping of a particular dictionary is completed, Phase AI will attempt to resume formatted dumping of any remaining dictionary sections.

7. Dump of text. When the DUMP option is specified without a value list, a formatted dump of text areas is printed. The text areas that are dumped vary according to the phase in which the interrupt occurs.

   If the interrupt occurs in a phase that processes sequential text, the contents of the current input text page are dumped. If such a phase reads more than one text stream, the current input page that is dumped depends upon the page number set in the XSIPT field of the communications area at the time of the interrupt.

   The output pages that are dumped also vary from phase to phase. For any phase, the contents of each page in the main text stream are printed, in formatted image where possible. For phases that output more than one stream of text pages, the contents of the chain of text pages anchored in the X2STRM field of XCOMM are also printed. If the format of text contained in this second output stream is similar to that used in the main text stream, these pages are also dumped in formatted image; otherwise, they are dumped unformatted. The contents of a page used as a work page, cr as a phase-to-phase information page, are also dumped if the track address of the page is stored in the XSCRCH field of XCOMM. At the end of the dumps of output text pages, the contents of the current error message page are printed. The dump of the contents of the main text stream is preceded by the header: "OUTPUT TEXT STREAM". Other output text streams are preceded by identifying headers.

   The dump of each text page is preceded by a sub-header showing the start address of the page. Note that a location address may be identical to that for other pages, as pages that are not in main storage are read into page spaces to maintain the sequence of the text stream dump.

   The formatting of text page dumps varies from phase to phase. If the text output from a phase is sequential, the text page number is printed at the head of the page contents, and the offset from the origin of the page is printed at the beginning of each line of text. If the text output from the phase is non-sequential, i.e., Type 2 text output where the logical sequence is maintained by chains to and from text tables in overflow pages, an overflow page index table is printed at the head of each main stream text page, and the full text reference, i.e., page number and offset, is printed at the beginning of each line of text.

   In a dump of Type-1 text, a new print line is started for each statement, identified by a statement header beginning with X'FC' or X'FD'. Print lines are not of fixed-length. Within each statement, operands, operators, etc., are separated by blanks. Six-byte references to operands are identified by a preceding X'DE'.

In a dump of Type-2 text output, each print line contains one 32-byte text table. The EBCDIC representation of the IOP1 field is printed to identify the type of each text table. Within each text table, the various fields are separated by blanks. At the top of each column, the various fields are identified by descriptive headings. Where the text is non-sequential, values in the FWD chain field indicate the overflow page numbers referred to in the overflow page index table printed at the head of each page.

If Phase AI is unable to print a formatted dump of the contents of text pages, or to continue a partly printed formatted dump, an unformatted dump is printed. In such a case, the page number of each page is printed, and also the start and end address of processable data within that page. The text is printed in lines of 32 bytes, each line being divided into four-byte groups. Each line is preceded by the start address of the line. For non-sequential text, the dump does not show the contents of overflow pages. If a formatted dump changes to an unformatted dump, no later reversion is made to the formatted image, as in the case of a dump of dictionary sections.

USE OF THE DUMP OPTION WITH A VALUE LIST (INTERPHASE DUMP OR UNFORMATTED COMPILER ABORT DUMP)

When the DUMP option is used with a value list, the format of the *PROCESS statement is as follows:

```
*[ ]PROCESS [options,]DUMP|DU(value-list)[,options]
```

A variety of arguments can be specified in the value list, the format being:

```
(value list) = ([C][E][S][H][F][U][2]
                [,T[F]{(W)      }]
                      {(C)      }
                      {(x[,y])  }
                [,D[F]{(W)                                  }]
                      {(T1[,T2][,T3][,T4,Ratcd[,1]])        }
                ,(P1[,P2].....[,P3 - P4]))
```

The value list format is free form in that the groups of arguments may be entered in any order, and the commas are optional.

The key characters that can be specified in the first argument indicate:

C	Communication area (XCOMM).
E	Current diagnostic message page.
S	Phase working storage (XSTG).
H	Page header information.
U	Register usage table (Phase QA interphase dump only).
F	Label trace table or execution trace bit string (not available without reassembly of the compiler phases. See "Use of the XBUG Macro").
2	Second text stream.

Each of these key characters is optional. If specified, they may be written in any order.

The next argument, T, specifies that a dump of text data is required. It can be qualified by 'F', which indicates that a formatted dump of the specified text areas is required. If T or TF is specified, it must be followed by one of the three qualifications which indicate the area of text to be dumped:

(W) = whole of text output from the phase, i.e., main output text stream, second output text stream (if applicable), and scratch text page (if applicable).

(C) = dump of current output text page.

(x) = dump of the output text representing source statement number x.

(x,y)= dump of the output text representing source statements in the range of source statement numbers x to y inclusive.

The next argument, D, specifies that a dump of dictionary data is required. It can be qualified by 'F', which indicates that a formatted dump of the specified dictionary areas is required. If D or DF is specified, it must be followed by one of the two qualifications which indicate the dictionary area to be dumped:

(W) = the whole of the dictionary existing at the time, i.e., names, general, variables, and storage dictionaries.

(Tx) = type of dictionary section, i.e., N, G, V, or S. If more than one dictionary section is specified, the qualification can be written in any order.

(Tx,Rabcd)= one dictionary entry, reference abcd (hexadecimal), in the dictionary section Tx is to be dumped. Thus DF(V,R0004) indicates that a formatted dump of the fifth entry in the variables dictionary is required.

If a particular entry in either the names or general dictionary is specified, the entry reference must be followed by the length (decimal bytes) of that entry if an unformatted dump is specified. Thus D(N,0003,14) specifies that an unformatted dump of the fourteen-byte entry at reference 003 in the names dictionary is required.

The next argument, (Pn), specifies the name of the phase after which an interphase dump is required. The name is in the form of the last two characters of the phase name, e.g., GA, KX, PI, etc. One or more phases can be specified, and a dump of the data areas specified in preceding arguments will be printed on completion of execution of the phase or phases. If a dump is required after each of a successive sequence of phases, the first and last phases in the sequence can be specified with an intervening '-' character. Thus, (GA-GM) specifies that a dump is required after execution of the phases GA, GI, GE, and GM. Note that the range (xx-yy) specifies only phases whose names lie alphabetically between xx and yy, not phases which lie logically between them. If a sequence of phases is specified, no additional phases can be added to the end of the list. Thus, (EA,GA-GM,II) will result in dumps being printed after phases EA, GA, GI, GE, and GM, but not after Phase II. However, as the list of phases need not be written in any particular sequence, this situation can be overcome by placing the sequential range of phases last in the range list, i.e., (EA,II,GA-GM).

Regardless of the phases for which interphase dumps are specified, a compiler abort dump will be printed in case of abnormal termination caused by a program-check interrupt during any phase.

Note: If an unformatted dump of text and/or dictionary data is required in the case of abnormal termination of execution of the compiler, the DUMP option must be specified with a value list in which the arguments T and/or D are written without the qualification F. In this case, the syntax of the option must be completed by inclusion of a phase list. If an interphase dump is not required, the item in the phase list should be the name of the last compiler phase or a fictitious phase name (e.g., ZZ).

USE OF REGISTERS IN THE COMPILER PROGRAM

The symbolic names applied to general registers used by the compiler, and the ccnventions applied to register usage by phases and macros, are shown below:

Register Number	Symbolic Name	Use Within the Compiler
0	R0 or RO	General work register.
1	R1 or RI	Input text register (base register specified in a USING statement for DSECTs which described general input text formats).
2	R2	Transfer vector, index register and general work register.
3	R3	Output text register and general work register (register used as current output pointer by output macro and specified as the base register in a USING statement for a DSECT which describes the output formats).
4	R4	First dictionary pointer and general work register (register used as absolute dictionary entry pointer. Specified as the base register in a USING statement for a DSECT which describes all dictionary entry formats).
5	R5	Second dictionary pointer and general work register (register used as absolute dictionary entry pointer. Specified as the base register in a USING statement for a DSECT which describes duplicates of all dictionary entry formats, so that dictionary entries may be compared and accessed simultaneously).
6	R6	General work register or module base register.
7	R7	General work register or base register.
8	R8	General work register or base register.
9	R9	General work register or base register.
10	RA	Storage DSECT base register.
11	RB	Parameter register or general work register.
12	RC	Parameter register or general work register.
13	RD	Communications area and control phase base register (specified as the base register in a USING statement for a DSECT which describes the communications region).
14	RE	Return register used by control phase and all macros. Must be preserved on entry to all subroutines in user allocated storage.
15	RF	Branch register. Contains an identification code on entry to the control phase which specifies the required function. General work register.

USE OF COMPILER DEBUGGING MACROS

A number of macros can be used to provide facilities for testing and debugging compiler phases. These facilities are not immediately available, but require modificatcn and reassembly of the particular phase or module to be tested. They are provided primarily for use by development and maintenance programmers. The following paragraphs describe the facilities that can be made avaialble by the macros listed below.

XBUG — Used to specify label trace or execution trace requirements, and to enable the debugging facilities of other macros to be made available.

XLAB — Used to indicate a trace checkpoint if the label trace facility is specified.

XTRSW — Used to limit the area of a module in which label trace or execution trace code is generated.

XLABTAB — Used to generate a table which relates checkpoints in the compiler program to bit positions in the execution trace bit string.

XDYDP — Used to obtain a printed dump of the contents of various data areas during execution of a phase.

USE OF THE XBUG MACRO

The XBUG macro is used to set the debugging level for a compiler module, and thus to control the generation of debugging code at module assembly time. Each module cf the compiler contains one or more XBUG macro statements. One XBUG macro statement precedes all other compiler-macro statements in a module. XBUG macro statements can alsc appear at other positions in the module. The format of the XBUG macro statement is:

blank	XBUG	[BGL = 0|L|E][,TRACE = n]

BGL = 0 (Suppression of Debugging Code)

In the published version of the compiler, each XBUG macro statement will appear with the operand BGL=0, or without operands, in which case BGL=0 is applied by default. In these cases, the generation of debugging code at module assembly time is suppressed. Debugging code can only be generated as a result of chaining the value to BGL=L or BGL=E, and reassembling the module.

Note: Alternatively, the debugging code may be suppressed in some phases by entering the XBUG macro statement as a comment, i.e., with an * in column 1. Thus any value of BGL and/or TRACE may be inserted. To enable the statement, the * should be deleted.

CAUTION

Use of the XBUG macro with a specified debugging level other than zero can cause the generation, at module assembly time, of debugging code within the containing module. The generation of such code can cause an increase of approximately 30% in the storage requirement for the module. If the debugging macro facilities are used, attention should be given to storage management considerations. Use of the debugging macros should be controlled so that debugging code is generated only in areas of the program where the facilities are specifically required.

BGL=L (The Label Trace Facility)

If BGL=L is specified in an XBUG macro statement, a label trace table can be specified in any invocation of the compiler dump phase, either by use of the DUMP option or by invocation of the XDYDP macro. The label trace table will contain the labels on all labeled XLAB macro statements and most labeled compiler-macro statements, listed in the order in which the statements are executed. (Use of the XLAB macro is explained in a later paragraph.)

The number of labeled statements for which a trace is maintained and printed can be specified in the TRACE operand of the XBUG macro statement. The format of the cperand is TRACE=n, where n is the number of labeled statements for which a trace is to be maintained. If the TRACE operand is not specified when BGL has a non-zero value, the operand is applied by default with a value of 50. The value of n defines the number of label checkpoints that are maintained in the trace, and that are printed in the dump at the time of interrupt or at the end of execution of the phase.

The label trace table can only be printed if the DUMP option is specified. The table is automatically included in a formatted compiler abort dump if BGL=L is specified in each invocation of the XBUG macro. If the table is required in an interphase dump, the value list of the DUMP option must include the argument 'F'. If the table is required in a dynamic dump produced by invocation of the XDYDP macro (described in later paragraphs), the invoking macro statement must contain the operand 'F'.

Checkpoints for Label Trace: Most of the compiler macros contain code that enables a labeled macro statement to be used as a checkpoint for the label trace facility. If it is necessary to provide additional checkpoints, uniquely labeled XLAB macro statements should be inserted at appropriate places in the compiler program. When the trace facility is enabled, each statement that is used is followed by a comment:

```
**CHECKPOINT FOR LABEL TRACE**
```

CAUTION

Use of the label trace facility causes generation of additional code requiring 20 bytes of storage (12 bytes if the NS option to the XLAB macro is used) for each checkpoint used. When storage requirements are critical, use of the label trace facility should be confined to specific areas of code by use of the XTRSW macro.

BGL=E (Execution Trace Facility)

If BGL=E is specified in an XBUG macro statement, an execution trace can be specified in any invocation of the compiler dump phase, either by use of the DUMP option or by invocation of the XDYDP macro. The execution trace consists of a bit vector in which bits are set to indicate monitored compiler statements that have been executed. The execution trace can be used to determine the areas of compiler code that are executed.

The monitored statements, for which bits are allocated in the bit vector, include labeled compiler macro statements, and conditional macro statements (even though they may not be labeled). In the case of conditional macros, a bit position is allocated for the fall-through path; any branching path is usually labeled. In the phase listing, each monitored statement is followed by a comment which indicates the relevant bit in the label trace, e.g.,

```
**EXECUTION TRACE,POSITION 83,BYTE 10,BIT 2**
```

For ease of reference to the execution trace printed by the compiler dump phase, a consolidated list of the comments can be produced at the end of the phase listing by use of the XLABTAB macro statement at the end of all code using execution trace. The format of the statement is:

```
XLABTAB
```

The consolidated list shows any applicable labels.

The XBUG macro argument, TRACE=n, does not apply to the execution trace facility. However, the areas of code in which statements are monitored, and hence the amount of tracing code that is generated, can be controlled by use of XTRSW macro statements.

Control of Tracing Code Generation. (The XTRSW Macro)

In order to control the amount of tracing code that is generated when using the label trace or execution trace facilities, XTRSW macro statements can be used to limit the trace facility to particular areas of a compiler module. The macro is invoked by a statement with a single non-optional operand, ON or OFF. Use of such statements is illustrated below:

```
-
-
-          }trace on.
-
XTRSW OFF
-
-          }trace suppressed.
-
XTRSW ON
-
-          }trace on.
-
-
```

USE OF THE XDYDP MACRO (DYNAMIC DUMPING FACILITY)

The XDYDP macro can be used to complement the dumping facilities provided by the DUMP option. Each invocation of the XDYDP macro causes the generation of code to prcduce a printed dump of the contents of various data areas at the time of invocation. Thus, a dump can be obtained at any time during the execution of a phase without terminating compilation.

Note: Code is only generated if the XBUG macro is invoked with BGL¬=0. If trace facilities are not required, BGL may be in the range 1 to 9. The DUMP option must be specified in the *PROCESS statement. The compiler dumping phase (Phase AI) cannot be loaded into the phase area as it would overwrite the phase being executed. Therefore at least 16K bytes of storage are required for residence of Phase AI throughout compilation.

A statement to invoke the XDYDP macro can be inserted at any suitable place in the phase code. If more than one invocation is required in a phase, it is recommended that the XDYDP macro statement be included in a subroutine that can be called as required. This method reduces the amount of code that is generated.

The format of the XDYDP macro statement is:

```
┌----------------┬--------------┬--------------┐
|        [label] |     XDYDP    | value-list   |
└----------------┴--------------┴--------------┘
```

```
value-list = [F][S][C][R][U][H][E]
                [,D=(F|U,{W                                  })]
                         {T1[,T2][,T3][,T4,R,r[,l]] )}
                [,T=(F|U,{W                  )}]
                         {C                  )}
                         {L                  )}
                          X1[,X2]
                [,RGE=(S,E)][,STMT=(a[,b])]
```

Each of the arguments in the value list is optional but there is no default value; if no arguments are specified, no dump will be printed.

The key characters that can be specified in the first argument indicate:

F = label trace table or execution trace bit string
S = phase working storage (XSTG)
C = communications area (XCOMM)
R = contents of registers at time of macro invocation
U = current register usage table (Phase QA only)
H = page header information
E = current diagnostic message page

If specified, these characters can be written in any order.

In the next argument, the key character D specifies that a dump of dictionary data is required. The values that can be assigned to the key character indicate:

F = formatted dump

U = unformatted dump

W = the whole of the dictionary existing at the time

Tx = type of dictionary section, i.e., N,G,V, or S. If more than one dictionary section is specified, the qualifications can be written in any order.

R specifies that a single dictionary entry is to be dumped. The type of the dictionary section is specified in the Tx entry immediately preceding the R.

r = the reference of a particular dictionary entry. It must be defined as a DC or DS of the type X, C, or H, or a self-defining term consisting of four hexadecimal digits.

I = the length of the entry, if it is in the names or general dictionary. It must be defined as a DC or DS of the type X, C, or H, or a self-defining term.

In the next argument, the key character T specifies that a dump of text data is required. The values that can be assigned to the key character indicate:

F = formatted dump

U = unformatted dump

W = whole of current output text. This includes the main output text stream, the second output text stream (if applicable), and the scratch page (if applicable).

C = latest output text page.

I = current input text page.

x1[,x2] specifies that statement x1, or the range of statements x1 to x2 is to be dumped. x1 and x2 must be specified by DC instructions of type X, C, or H, register-names, or self-defining terms.

The key characters RGE specify that the contents of a particular area of main storage are to be dumped. The qualifications indicate:

S = the start address of the area. If it is a register name, or a DC or DS instruction of type 'F' or 'A', the address is assumed to be contained in the field indicated. Otherwise the symbol is assumed to be a label indicating the start of the area.

E = the end address of the area. It may be specified in any of the forms allowed for the S qualification.

The STMT argument is used to suppress dynamic dumps where the number of the statement being processed at the time of macro invocation is outside the range of statements specified in the qualifications. The range can be specified as a single statement number (a) or as start and end statement numbers (a,b). The number of the statement currently being processed is assumed to be in the XSTAT field of the communications area. A facility similar to that provided by the use of the STMT argument is provided by use of the DYSTMT option, described below.

Use of the DYSTMT Option

The DYSTMT option can be used in a *PROCESS statement to complement use of the XDYDP macro. The function of the DYSTMT option is to suppress printing of a dynamic dump where the number of the statement being processed at the time of invocation of the XDYDP macro is outside the range of statements specified in the DYSTMT option.

The format of the option is:

```
DYSTMT (x[,y])
```

The option can be used to specify a single statement number, x, or a range of statements where x = the first statement number and y = the last statement number in the range.

FACILITY FOR TESTING MODIFIED PHASES

The compiler provides a facility for testing one or more modified processing phases without deleting the existing version of the phase(s) from the core image library.

The modified phase (or phases) must be appropriately named (as described below) and added to the core image library by suitable linkage editing. The phase testing facility is invoking by specification of a special option in the *PROCESS statement. The format of this option is:

```
Tn(pp[,pp]),
```

where n = any single numeric or alphabetic character.

pp = the last two characters of the symbolic name(s) of the phase(s) to be replaced.

Specification of this option causes Phase PLIOpp to be replaced by a phase named PLIOppn during the compilation affected by the containing *PROCESS statement. An example of use of the option is shown below.

The statement:

```
* PROCESS T1 (GA,GI,GE), T2 (KA,KV),TX(IE,KT)
```

will result in phases { PLIOGA, PLIOGI, PLIOGE, PLIOKA, PLIOKV, PLIOIE, PLIOKT } being replaced by { PLIOGA1, PLIOGI1, PLIOGE1, PLIOKA2, PLIOKV2, PLIOIEX, PLIOKTX }

The option is processed by Phase AE (Compiler Initialization Phase) when it reads the *PROCESS statement at the start of a compilation. In response to the option, Phase AE modifies the phase list in the XPHSL field of the communications area, by replacing the existing phase name in the list with the name of the modified phase. When the XPST macro in a processing phase specifies the original phase name as that of the next phase to be loaded, the phase loading routine examines XPHSL, matches the specified phase name with the modified phase name, and has the modified phase loaded.

If, after testing a modified phase, it is desired to permanently replace an existing phase with the modified phase in the core image library, the can be done by using the DOS Librarian Service Program, MAINT. The instructions required to do this are:

```
//JOB jobname
//EXEC MAINT
b DELETC oldmaster [,oldmaster]
b RENAMC testname, newmaster [,testname, newmaster]
/*
/&
```

COMPILER DIAGNOSTIC MESSAGES

The diagnostic messages output by Phase UA all have the message number IEL0xxxI. Each message can, in general, be identified with the compiler phase during the execution of which the appropriate error condition occurred by means of this number, as illustrated in figure 6.1. However, some messages are produced by phases prior to the one shown in this figure. The wording of the messages, and explanations of their meanings, are contained in the publication : DOS PL/I Optimizing Compiler: Messages.

Message Number IEL0xxxI	Phase(s)	Message Number IEL0xxxI	Phase(s)
001	Preprocessor error message		
002-040	AE	787	KI
041-049	AA	798-810	KT
050-229	CA(Preprocessor)		
230	Compiler-error messages. See Appendix B.	811-812	KM
		813	KL
		814-817	KM
231	BA		
232-400	EA, EE, & EI	818	KM &II
401	GA, GE, GI, & GM	819-820	KM
402-430	GA	821-826	KL
431-440	GA &GM	827-828	KL & KM
441-495	GE	829-833	KL
		834-835	KL & KM
496-510	GI		
511-523	GM	836	KL
524	GA & GM	837	KL & KM
525-526	GM	838-847	KL
527	All phases using the XBREAK macro.	848-865	KQ
		866-872	KK
		873-874	KQ
		879	All phases using KNSRT macro
		880	KV
528-531	GM	886-891	OC
540-559	GA	892	OC & PA
560-579	GI		
580-600	GE	903-905	OX
601-617	IA	906	KX
618-640	ID	907	KA
641-658	IE	908	KX
		909	OE
659	Various phases.	910-913	OA
671	II & ID	914	KQ, OE & KK
672-690	II	915	OA
691	II & KA	916	OI
692-695	II	917-920	OE
701-740	IM		
741-742	IQ		
761-769	KA		
776	KE	921-924	PC
777	KE & PA	925	OI
778	KE	926-927	PA
779	KE & PA	931-932	PE
		933-934	PC
		940	OE
		960	SI
		961	SK
		965-967	SI
		969	KQ
		970	Compiler-error messages. See Appendix B.
		971	KL & KM
		972-981	KL

Figure 6.1. Compiler diagnostic messages - phase identification

APPENDIX A: FUNCTIONS OF THE COMPILER MACROS

The lists in this appendix show the names of macro instructions and books used in the compiler program, together with a brief description of the funciton of each macro. Macros are grouped under the following headings:

Phase and Module Construction Macros.

Input/Output Macros.

Text Accessing Macros.

Dictionary Accessing Macros.

General Purpose Macros.

Special Purpose Macros.

Dubugging Macros.

Books Invoked by the COPY Statement.

Within each group, the macros are listed in alphabetic sequence.

MODULE CONSTRUCTION MACROS

XENDSTG — Delimits module storage.

XINIT — Module initialization.

XROUT — Conditionally invokes subroutine macros to generate subroutines required by macros invoked in the module. The subroutines that can be invoked by XROUT are:

XBAKTXR	XREADR
XBREAKR	XRFABI
XBRICM	XRFSEQI
XDISCR	XSEQRT
XDIRECI	XSKEXPR
XDSTATI	XSRCHR
XLINKR	XTCHR
XMESGR	XTRCER
XNSRTR	XTUNLR
XNXROUTA	XTXENR
XNXROUTB	XTXPGR
XPRNTR	XROUTEQ(unconditional)

XSTG — Allocates and initializes phase working storage. In a phase root module, XSTG defines a CSECT. In a non-root module, XSTG defines a DSECT at a specified offset.

INPUT/OUTPUT MACROS

XBUF — Builds a record in a print buffer.

XLOADI — Initializes the IJSYSLN output data set.

XLOADF — Copies a record to the IJSYSLN output data set.

XPRNTI	Initializes the IJSYSLS print data set.
XPRNT	Copies a 121 byte record to the IJSYSLS print data set.
XPRNTN	Sets the print data set to new page.
XPUNCHI	Initializes the IJSYSPH output data set.
XPUNCH	Copies a record to the IJSYSPH output data set.
XREADI	Initializes the IJSYSIN input data set.
XREAD	Reads an 80-byte record from the IJSYSIN input data set.

TEXT ACCESSING MACROS

The macros used for text accessing operations are grouped according to whether they are used for general text-handling operation, or for accessing text data with a particular format.

General Text Accessing Macros

XBAKTX	Backspaces the output text stream pointer, either maintaining the current output pointer or discarding text to backspace reference.
XCHECK	Checks for completion of page input/output operation.
XDISC	Discards a list of text pages.
XFREE	Places UNMOVABLE text and/or dictionary and directory pages in the MOVABLE page chains.
XTARF	Converts an absolute address to a five-byte text reference.
XTXPG	Gets new or existing text pages.
XTXRF	Converts a five-byte text reference to an absolute address.
XTXST	Changes the status of a text page.

Sequential-Text Accessing Macros (Type-1 Text or Extended Code)

XBREAK	Moves text into a main text stream output page, identifying a page-start break point.
XBREAK2	Moves text into a second text stream page, identifying a page-start break point.
XBRIC	Increments the input-pointer register, and chains input pages.
XBRICI	Initializes the scan of input text pages.
XOUT	Moves text into a main text stream output page, acquiring a new page if necessary.
XOUTI	Initializes the main output text stream.
XOUT2	Moves text into a second text stream output page, acquiring a new page if necessary.
XOUTI2	Initializes a second stream of output text pages.

Type-2 Text-Accessing Macros

XITCH	Initializes the text-chaining macro, XTCH.
XLINK	Chains two Type-2 text tables in sequence.
XNEXT	Follows a chain of Type-2 text tables.
XNEXTL	A combination of the XNEXT, XTUNL, and XTLOK macros.
XNSRT	Inserts a Type-2 text table into a chain of Type-2 text tables.
XTCH	Follows a statement-type chain of Type-2 text tables, having pages read into main stroage if necessary.
XTLOK	Controls the locking into main storage of a text page.
XTUNL	Unlocks a text page that was locked by the XTLOK macro.

DICTIONARY ACCESSING MACROS

XDSTAT	Changes the status of a dictionary page from 'read-only' to 'read/write'.
XLOK	Controls the locking into main storage of a dictionary page.
XRFAB	Converts a two-byte dictionary reference to an absolute address, and has the relevant page read into main storage if necessary.
XRFSEQ	Adds a new sequential dictionary entry.
XSEQ	Scans the names dictionary for an existing entry.
XSEQI	Initializes the dictionary scan by XSEQ.
XUNLOK	Unlocks a dictionary page that was locked by the XLOK macro.

GENERAL PURPOSE MACROS

XADD	Adds two operands and assigns the result.
XALM	Tests whether an item is an alphameric character, and branches if so.
XALPH	Test whether a character is a blank, an alphabetic letter, a numeric digit, an underscore, or some other character.
XAND	Performs a logical AND operation.
XASN	Assigns the second operand to the first operand.
XASNA	Performs a Load Address operation.
XASNV	Assigns the contents of the second operand to the first operand, irrespective of implied lengths.
XASNX	Performs assignment to and from registers, to and from character items with lengths of one to four bytes, or between both types of operands. (Invoked only by other macros.)
XBKSTK	Scans backwards through entries in a stack.
XBPIK	Tests, sets on, or sets off, a particular bit in a string, or produces a mask and offset to enable some other operation to be performed.

XBSET	Sets a bit in a bit vector.
XBTST	Tests a bit in a bit vector.
XBUMP	Increments a counter by a specified amount.
XBUST	Sets off a bit in a bit vector.
XCALL	Branches and Links to a specified subroutine.
XCOMP	Generates a comparison operation for all types of operand.
XCOMPARE	Generates a CLC and a conditional-branch instruction, allowing the programmer to specify a length.
XCPIK	Scans a bit vector for the next bit set on.
XCSEQ	Generates a sequence of CLI instructions, with branches on specified conditions.
XDEC	Decrements a counter by a specified amount. Can also specify a conditional branch.
XENTRY	Saves the contents of specified registers at secondary entry points to subroutines (used in conjunction with XSAVE, XRTN, and XRST).
XEX	Performs a logical EXCLUSIVE-OR operation.
XFSTK	Adds a fixed length entry to a stack, and increments the stack pointer.
XGOTO	Branches to a specified address if a specified condition is satisfied.
XIF	Compares two operands, and transfers control according to the result of the comparison.
XIFB	Compares two bits, and transfers control according to the result of the comparison.
XIND	Sets a pointer to a particular index in a table of fixed length entries.
XLBIT	Given a library subroutine name, sets the appropriate bit in the XLIBSTR bit vector.
XLD4	Loads four bits from a byte into a register.
XLET	Tests whether a character is an alphabetic letter, and branches if not.
XLMVE	(Long Move). Moves a string of bytes, the length of which may exceed 256 bytes, from a source field to a target field.
XLMVR	(Long Move, Registers.) Moves a string of bytes, the length of which may exceed 256 bytes, when the address of the source field, the length of the source field, and the address of the target are all contained in registers.
XLZERO	Pads out a character field to any specified length with zeros (or any other character). The field may be of any length.
XMESG	Makes entries in the diagnostic message page stream.
XMOVE	Moves a string of bytes, the length of which is less than 256 bytes, from a source field to a target field.
XMULT	Multiplies two operands and assigns the result to the first operand.
XMV4	Moves four bits from a source field to a target field.

XNXVL	Increments a value and stores the result in a specified field.
XOR	Performs a logical OR operation.
XRRTN,XRRST	Subroutine exits for recursive procedures. Restore registers saved by XRSAVE.
XRSAVE	Subroutine entry for recursive procedures. Saves register contents in storage allocated by XSTG.
XRTN,XRST	Subroutine exits. Restore registers saved by XSAVE.
XSAVE	Subroutine entry. Saves register contents in a field allocated by XSTG.
XSHIFT	Performs single register shift operations. Operations may be left or right, arithmetic or logical.
XSPIK	Tests, sets on, or sets off, a particular bit in a string, given a mask and offset to use. Designed to be used after XBPIK.
XSTACK	Makes an entry in a stack, and updates the stack pointer. Fields that are stacked need not be fixed length. (The stack and pointer must be initialized by XSTKSET.)
XSTKSET	Used to initialize storage and pointers for stacks, if variable-length-entry stacking macros are used (XSTACK, XUNSTK, XBKSTK.)
XSUB	Subtracts the third operand from the second operand, and assigns the result to the first operand.
XTM	Generates a single Test Under Mask instruction, followed by one, two, or three conditional branches.
XTRT	Generates the code and tables for scanning character strings and transferring control for particular characters found in the string. The translate tables are produced automatically by the XSTG macro if the XTRT macro is used.
XTSEQ	Used to generate a sequence of Test Under Mask instructions. For each instruction in the sequence, the field to be tested, the mask, the condition to be satisfied, and the label to be branched to if the condition is not satisfied, must be specified.
XTYPE	Analyzes the data types of up to three operands, and places the results in a set of global variables.
XUNIV	Generates object code for instructions that can have RR, RX or SS instruction formats, e.g., AND, OR, comparisons etc.. (Invoked only by other macros.)
XUNSTK	Removes the top entry from a stack by moving the stack pointer back one entry. (The stack and pointer must be initialized by XSTKSET.)
XZERO	Pads out a character field to a specified length with zeros (or any other character). The field length must not exceed 256 bytes.

SPECIAL PURPOSE MACROS

XATFLD	This macro is used in Phase GA. It generates a name for a field to be associated with an attribute, in the attribute list, ATLST.
XATROUT	This macro is used in Phase GA. It generates a list of address constants. The address constants are used for addressing routines associated with an attribute.

XBFSK — This macro is used in Phase KK. It creates a nine-byte 'driver' table, defining the contents required in various fields of a phase-output text table.

XBT,XBTEND — These macros are used in the Code Generation Phases, SA, SQ, SD, and SC. The XBT macro is invoked for each invocation of XSK. XBT is used to build an array of bit strings, the length of each strip varying between 1 and 16 bits. When the array is complete, XBTEND is invoked to transpose each bit strip so that the array in main storage is the transposed version of the array as printed in the program listing.

XBT0 — This macro is used in the Code Generation Phases, SA, SQ, SD, and SC, to produce a label preceding the bit strip arrays or special case code.

XCADD — This macro is used in Phases IQ and KM. The macro accesses an aggregate table entry in the general dictionary to construct an aggregate descriptor for a major or minor structure. It is used in conjunction with the book invoked by a COPY ADDSECT statement.

XCD0 — This macro is used by the Code Generation Phases, SA, SQ, SD, and SC, to produce a label preceding the bit strip arrays or special case code.

XCHIP — Bumps the input pointer and tests for endpage. (Used by Phase II.)

XCODE — This macro is used in the Code Generation Phases, SQ, SQ, SD, and SC. It initializes a number of global C macro variables that are used by invocations of the XSK macro.

XCOPT — This macro is used in many phases. It tests for the specification of any particular compiler option, with the exception of those specifying listings - type output.

XCVD — This macro is used in the Object Code Listings Phase, SM. It converts the contents of a register from binary format to binary-coded decimal format. The result is returned in XALIGN (in XSTG).

XCV1 — This macro is used in a number of phases for conversion operations. It evalus ons. It evaluates the expression CEIL (source precision*3.32).

XDIAG — Tests the various conditions affecting the loading of the diagnostic and error message editing phases (CE and UA), and generates an XPST macro instruction to indicate the next required phase

XFUNC — This macro is used in Phases ID and IE. It tests an operand code byte, defined in the book ZTEXT and named ZCDE, to determine whether it indicates a built-in function or a programmer-defined function.

XGOD — This macro is used in Phase II. It is used for branching between CSECTS that have been based on the same registers.

XINF — This macro is used in the Code Generation Phases, SA, SQ, SD, and SC. For a given text table operator, it generates a directory to the relevant code skeleton, bit-string array, and special-case code.

XITOC — This macro is used in the Object Code Listings Phase, SM. It translates an identifier name from internal to external character form, and normally moves the result to a print output buffer.

XKILL — This macro is used by a number of phases, particularly those in the Statement Processing Stage. It tests an operand code byte to see if the operand is an active temporary operand and, if so, deactivates it. It can also be used to activate inactive temporary operands.

XLBTB — This macro is used in Phase SI. It is used to index the Label Table built in Phase SK.

XMDE — This macro is used in the error-message editing phases, CE and UA. It is used to contstruct five tables: a keyword table (KEYTAB), a

keyword-reference table (KEYREF), a message table (MESTAB), a message-reference table (MESREF), and a table of attributes, built-in functions, and environment options (ATTAB).

XPOP — Decrements the stack pointer and tests for underflow. (Used by Phase II.)

XPST — This macro is used at the end of processing in each phase. It is used to specify the next phase to be loaded.

XPUSH — Increments the stack pointer and tests for stack overflow. (Used by Phase II.)

XSCPE — This macro is used in the Dictionary Build Stage, Phases GA, GI, GE, and GM. It updates a list (KNOLST) of currently known blocks.

XSETCD — This macro is used in most of the phases that make entires in the dictionary, or insert dictionary references in the text. It sets bits in the dictionary code byte.

XSK — This macro is used in the Code Generation Phases, SA, SQ, SD, and SC. It builds skeleton machine instructions.

XSK0 — This macro is used in the Code Generation Phases, SA, SQ, SD, and SC, to define the DSECTs containing the code skeleton arrays.

XSKEXP — This macro is used by phases in the Dictionary Build Stage. It is used to skip expressions during scans of sequential text.

XSRCH — This macro is used by phases in the Dictionary Build Stage. It searches the dictionary (via hash chains) for an entry identified by a name. The name can be qualified or subscripted. The code generated by the macro calls the XSRCH subroutine generated by the XSRCHR macro in XROUT.

XTCDE — This macro is used by a number of phases. It tests the code byte of a variables dictionary reference.

XTDAT — This macro is used by many phases. It tests the data type byte in compile-time DEDs.

XTEMP — Tests code byte of an operand in text to determine whether it is a temporary operand or a particular type of temporary operand.

XTND — This macro is used by Phase GA. It extends the names dictionary hash chain by adding an entry for a name.

XTOPT — This macro is used in many phases. It tests for the specification of those compiler options requiring listings - type output.

XTREE — This macro is used in Phases GA and GI. It generates a branch of the attribute tree, used to apply attributes to an identifier when building dictionary entries.

XTXEN — This macro is used by phases that process sequential Type-2 text. It finds the text table preceding a specified text table.

XXTOC — This macro is used by the Object Code Listing Phase, SM. It translates the contents of a specified field from binary format to external hexadecimal format.

DEBUGGING MACROS

XBUG — This macro is used in each module of the compiler. It sets the debugging level of the module and enables the debugging facilities of other macros to be made available.

XDYDP — This macro can be used in most modules of the compiler. It causes a dump of the contents of various data areas at the time of the macro invocation to be printed.

XETRC — This macro generates execution-trace code if BGL=E is specified in the XBUG macro. The macro is included in the code generated by many other macros.

XLAB — Used as a checkpoint for the label trace facility.

XLABTAB — Generates a consolidated list at the end of a module in which the execution trace facility has been enabled. The list relates bit positions in the label trace to labelled statements in the module.

XSTOP — This macro is invoked when a terminal error occurs in execution of the compiler program. It causes the compilation to be terminated and a compiler error message to be printed.

XTRCE — This macro is included in the code generated by a number of macros. Its function is to generate a statement trace, which is not available in the published version of the compiler.

XTRSW — This macro can be used to suppress the generation of tracing code in certain areas of a module when BGL=L or BGL=E is specified in the XBUG macro.

BOOKS INVOKED BY A COPY STATEMENT

COPY XADDSECT — Generates a DESCT, based on Register 5, defining aggregate descriptor descriptor elements.

COPY XCGSDCS — Defines character-string constants for compiler-generated subroutine names used in object listings.

COPY XCGSEQU — Generates EQU values for compiler-generated subroutine names used in compiler phases.

COPY XCOMM — Generates a DSECT that defines the compiler communication area. (Invoked in each module.)

COPY XEQU — Generates EQU statements to define symbolic register names, branch-code mnemonics, and the one-byte switch settings ON and OFF.

COPY XLIBDC — Defines character-string constants for library-module entry points, for use in object listings.

COPY XLIBEQU — Generates EQU values for library-module entry points.

COPY XMESGP — Generates a DSECT which describes entries in the current diagnostic message page stream.

COPY XTXEQU — Generates EQU statements to define all text code bytes.

COPY YDICT1 — Generates a DSECT, based on Register 4, which contains description of entries in the names, general, and variable dictionaries.

COPY YDICT2 — Generates a DSECT, based on Register 5, which contains a second version of the dictionary entries in YDICT1. The names of the fields in this DSECT

are the same as those for YDICT1, concatenated with the digit '2'. This DSECT enables two entries to be referred to simultaneously.

COPY YSDICT Generates a DSECT, based on any specified register, which describes the format of entries in the storage dictionary. This DSECT does not contain a register USING statement.

COPY ZDATAL Generates halfword data definitions, giving instruction sizes at object module assembly time. Used by Phases SI and SM.

COPY ZEQUB Generates EQU values for print-buffer offsets, used by the listing phases.

COPY ZEQUL Generates EQU values for markers and branch-load instructions. Used by Phases SI and SM.

COPY ZTEXT Generates a DSECT, based on Register 1, which contains descriptions of Type 1 text formats.

COPY ZTEXTL Genrates a DSECT, based on Register 1, which contains descriptions of extended code text formats. Used by Phases SA, SQ, SD, SC, SI, and SM.

COPY ZTEX2A Generates a DSECT, based on Register 1, which contains descriptions of Type 2 text tables. The initial letter of each field name is 'I'.

COPY ZTEX2B Generates a DSECT, based on Register 3, which is similar to ZTEX2A but in which the initial letter of each field name is 'O'. ZTEX2A and ZTEX2B enable two text tables to be referred to simultaneously.

COPY ZTEX2C Generates EQU values for Type 2 text tables.

APPENDIX B: COMPILER-ERROR MESSAGES

The compile-time messages from the DOS PL/I Optimizing Compiler can include compiler control messages, preprocessor messages, and compiler-error messages. Those messages forming this last group are listed in this appendix.

When a program-check interrupt occurs during execution of a compiler phase, an XSTOP macro statement is invoked. Execution of the statement causes a compiler-error message to be generated and written into the current message page. Control is then passed to the interrupt-handling routine in Phase AA, and this routine has Phase UA loaded to edit and print the message. (In the case of a preprocessor error, Phase CE is loaded.) The message is printed before any diagnostic messages that may have been generated prior to the interrupt, and immediately following the compiler dump, if one has been specified.

All compiler-error messages are of the "unrecoverable" type, and all have one of the following two message numbers, IEL0230I, or IEL0970I if severe errors are detected.

The format of a compiler-error message is:

$ COMPILER ERROR NUMBER n DURING PHASE pp.

where "$" represents the statement number in which the error occurred, "n" identifies the particular compiler-error message (the message list in this appendix is in numerical order), and "pp" identifies the phase in which the interrupt occurred. In some instances the phase may be one of several in which the interrupt could occur; these cases are represented by 'x'.

The preprocessor-error message has the message number IEL0001I and the format

$ PREPROCESSOR ERROR NUMBER n DURING PHASE pp.

In the following list, the information given for each compiler-error message contains, where applicable, the compiler-error number, the phase in which the error occurred, an explanation of the probable cause of the error, and possible programmer response to an occurrence of the error. The standard action for a member of IBM programming support personnel is to refer the problem to the appropriate program maintenance group within IBM for analysis and correction. This involves the submission of an APAR (Authorized Program Analysis Report), which must be accompanied by material to enable the program maintenance personnel to analyze the problem. In addition to submitting an APAR, in some cases it may be possible for the programmer to carry out a form of bypass action to alleviate the problem until the APAR has been processed and actioned. This bypass action is contained in the programmer-response information given in the following pages.

$ COMPILER ERROR NUMBER 0 DURING PHASE (any).

Explanation: A program check interrupt has occurred.

Programmer Response: Try correcting source errors, or running with larger SIZE parameter if possible.

$ COMPILER ERROR NUMBER 2 DURING PHASE AA.

Explanation: All pages in main storage are UNMOVABLE. An attempt has been made, in response to a request from the stated phase, to find a page which may be spilled in order to make room for either a new or an existing page. However, since all the pages are marked UNMOVABLE, no such spill candidate could be found.

Programmer Response: If possible, re-run the program with a larger SIZE specification. This will increase the size of the page area, and thus the number of pages in main storage.

$ COMPILER ERROR NUMBER 3 DURING PHASE (any).

Explanation: A call from the stated phase has been made to the control phase which necessitates either (a) writing a page to the spill file, or (b) reading a page into main storage from the spill file. Prior to the I/O operation (a) or (b), the track address of the page concerned has been found to be invalid. In case (a), the track address held in the header of the page in main storage has been overwritten, and in case (b) the track address of the requested page is invalid.

Programmer Response: Attempt simplification of the statement referred to in the error message.

$ COMPILER ERROR NUMBER 4 DURING PHASE AA.

Explanation: An attempt has been made by the stated phase to read into main storage an existing page (specified by its track address) from the spill file. This page, however, has not been spilled, the record at the given track address on the spill file being a dummy record at this stage. When this record is read into main storage, its track address field in the page header, not having been initialized, does not match that of the record.

Programmer Response: Attempt simplification of the statement referred to in the error message.

$ COMPILER ERROR NUMBER 5 DURING PHASE AI/UA/UE DUE TO PREVIOUS ERROR NUMBER n IN PHASE p.

Explanation: A compiler error has occured which makes it impossible for the error editor or the dump phase to continue.

Programmer Response: As for previous compiler error (NUMBER n).

$ COMPILER ERROR NUMBER 81 DURING PHASE EA.

Explanation: The compiler has attempted to correct a series of source errors, and this has had a cumulative effect leading to an "unrecoverable" error.

Programmer Response: Correct the source errors diagnosed before the above error and rerun the program.

$ COMPILER ERROR NUMBER 100 DURING PHASE (any).

Explanation: Invalid dictionary reference passed to decoding routine XRFAB.

$ COMPILER ERROR NUMBER 101 DURING PHASE (any).

Explanation: Dictionary full.

Programmer Response: If the compile-time preprocessor was used, check the logic of %GOTO and %DO statements for a permanent loop.

If a permanent loop is not found, or if the error occurred at a later stage in compilation, then increasing the storage available to the compiler may remove the error. If the error continues to occur, the program may have to be divided into smaller sections.

$ COMPILER ERROR NUMBER 103 DURING PHASE (any).

Explanation: An attempt has been made to create a dictionary entry larger than a page.

$ COMPILER ERROR NUMBER 105 DURING PHASE (any).

Explanation: Phase has requested a page that is said to be in the page area, but is not.

$ COMPILER ERROR NUMBER 151 DURING PHASE GA.

Explanation: Invalid or incorrect specifications have been included in the VALUE option of a DEFAULT statement.

Programmer Response: Avoid the use of, or correct, the relevant VALUE option specification(s) in the statement referred to in the error message.

$ COMPILER ERROR NUMBER 152 DURING PHASE GA.

Explanation: Too deep a parenthesis level has been used in an ENVIRONMENT attribute option-list.

Programmer Response: Avoid nested parentheses in ENVIRONMENT attribute option-list arguments.

$ COMPILER ERROR NUMBER 154 DURING PHASE GA.

Explanation: Error during the processing of the attributes in a DECLARE statement.

$ COMPILER ERROR NUMBER 201 DURING PHASE GM.

Explanation: An error has been made in statement-label handling.

Programmer Response: Check the syntax of the label prefix of the statement referred to in the error message.

$ COMPILER ERROR NUMBER 220 DURING PHASE (GA|GE|GI|GM).

Explanation: During the scan of an expression, the semicolon has been found in an apparently incorrect position in the statement.

Programmer Response: Check the syntax of the statement. If this is correct, the statement should be simplified.

$ COMPILER ERROR NUMBER 221 DURING PHASE IA.

Explanation: An illegal statement type has been found in the secondary input text stream.

$ COMPILER ERROR NUMBER 222 DURING PHASE IA.

Explanation: Underflow of implicit locator chain stack.

$ COMPILER ERROR NUMBER 223 DURING PHASE IE.

Explanation: Unqualified REFER item found.

Programmer Response: Avoid using the REFER option in this statement.

$ COMPILER ERROR NUMBER 224 DURING PHASE IA.

Explanation: An illegal statement type has been found in the secondary input text stream.

$ COMPILER ERROR NUMBER 261 DURING PHASE IE.

Explanation: Structure element descriptor cannot be found.

Programmer Response: Avoid using structures in this statement.

$ COMPILER ERROR NUMBER 262 DURING PHASE IE.

Explanation: Dimension entry cannot be found in dimension stack.

Programmer Response: Avoid using arrays in this statement.

$ COMPILER ERROR NUMBER 263 DURING PHASE IE.

Explanation: End of structure stack found where not expected.

Programmer Response: Avoid use of structures in this statement.

$ COMPILER ERROR NUMBER 264 DURING PHASE IE.

Explanation: End of dimension stack found when processing array of structures.

Programmer Response: Avoid using arrays of structures in this statement.

$ COMPILER ERROR NUMBER 265 DURING PHASE IE.

Explanation: End of text page found where not expected.

Programmer Response: Avoid array assignments in this statement.

$ COMPILER ERROR NUMBER 266 DURING PHASE IE.

Explanation: Aggregate assignment marker not followed by dictionaryreference.

Programmer Response: Avoid using functions with aggregate arguments in this statement.

$ COMPILER ERROR NUMBER 281 DURING PHASE II.

Explanation: Main stack underflow.

$ COMPILER ERROR NUMBER 282 DURING PHASE II.

Explanation: Main stack overflow.

Programmer Response: Simplify the statement involved.

| $ COMPILER ERROR NUMBER 301 DURING PHASE (any).

Explanation: More than 32 qualified temporaries are currently active.

Programmer Response: Simplify any expressions in the statement involved, particularly any that refer to based or subscripted variables.

| $ COMPILER ERROR NUMBER 302 DURING PHASE (any).

Explanation: The phase has encountered a reference to a qualified temporary without having encountered code for its creation. (Qualified temporaries are used for based and subscripted variables.)

Programmer Response: Simplify any expressions in the statement involved.

$ COMPILER ERROR NUMBER 303 DURING PHASE KA.

Explanation: The phase has found a reference to a string temporary but has not found code for the creation of such a string temporary.

Programmer Response: Simplify any string expressions in the statement involved.

$ COMPILER ERROR NUMBER 304 DURING PHASE KA.

Explanation: The phase has found a request for the creation of a string temporary in an operation that should not require one.

Programmer Response: Simplify the use of string expressions in the statement involved.

$ COMPILER ERROR NUMBER 305 DURING PHASE KA.

Explanation: Too many string temporaries (more than 25) are active.

Programmer Response: Simplify any string expressions in the statement involved.

$ COMPILER ERROR NUMBER 306 DURING PHASE KA.

Explanation: An error in the compiler labels generated for the program has been discovered.

Programmer Response: Rearrange the branching in an IF..THEN GOTO...statement.

$ COMPILER ERROR NUMBER 321 DURING PHASE IK.

Explanation: An incorrect entry has been found in the sort pages.

Programmer Response: Do not specify either or both of the ATTRIBUTE and XREF compiler options for this program.

$ COMPILER ERROR NUMBER 322 DURING PHASE IK.

Explanation: An incorrect entry has been found in the ENVIRONMENT attribute option-list for a file.

Programmer Response: Do not specify the ATTRIBUTE compiler option for this program.

$ COMPILER ERROR NUMBER 341 DURING PHASE IM.

Explanation: The "end of program" marker has been found in error. The marker has been encountered during a text scan before the "end of program" text table has been found.

$ COMPILER ERROR NUMBER 361 DURING PHASE IQ.

Explanation: For computing the size of a target of a concatenate operation the phase uses a stack whose maximum depth is 30. The maximum has been exceeded.

Programmer Response: Avoid using more than 30 operands in a concatenate operation.

$ COMPILER ERROR NUMBER 362 DURING PHASE IQ.

Explanation: Erroneous coding in the phase.

Programmer Response: Avoid built-in functions as operands in concatenate expressions.

$ COMPILER ERROR NUMBER 402 DURING PHASE KI.

Explanation: Owing to bad input from a previous phase (probably in the syntax checking stage) Phase KI is unable to find the text table corresponding to the end of a user-written DO-loop.

$ COMPILER ERROR NUMBER 421 DURING PHASE KT.

Explanation: Invalid condition seen as argument to a SIGNAL statement.

Programmer Response: Remove SIGNAL statement.

$ COMPILER ERROR NUMBER 461 DURING PHASE KM.

Explanation: Text table stack is full - logic error in Phase KM.

$ COMPILER ERROR NUMBER 481 DURING PHASE KQ.

Explanation: Text input to Phase KQ does not start with an SL text table.

Programmer Response: Simplify the first statement in the compilation.

$ COMPILER ERROR NUMBER 483 DURING PHASE KQ.

Explanation: A FORME text table of unknown type has been encountered by the phase. This is probably due to bad output from Phase II or a logic error in the processing of FORME text tables by Phase KQ.

Programmer Response: Simplify the appropriate stream I/O statement.

$ COMPILER ERROR NUMBER 485 DURING PHASE KQ.

Explanation: A Q-temp. encountered in a stream I/O text table has not been seen previously in the text.

Programmer Response: Simplify the appropriate stream I/O statement.

$ COMPILER ERROR NUMBER 488 DURING PHASE KQ.

Explanation: Error in input text - a null operand has been found in a DATAE text table.

Programmer Response: Simplify the stream I/O statement referred to in the error message.

$ COMPILER ERROR NUMBER 489 DURING PHASE KQ.

Explanation: Text input to Phase KQ contains no text tables for a format list.

Programmer Response: If possible, rewrite the GET|PUT EDIT statement with fewer pairs of data and format lists.

$ COMPILER ERROR NUMBER 492 DURING PHASE KQ.

Explanation: Input text error. The format list input text to Phase KQ in an edit I/O statement starts with a FITE text table.

Programmer Response: Simplify the format list in the edit I/O statement indicated by the error message.

$ COMPILER ERROR NUMBER 501 DURING PHASE KV.

Explanation: The phase has encountered an UNSPEC of a picture that should have been replaced by a reference to a character string.

Programmer Response: Avoid UNSPEC, particularly of pictures.

$ COMPILER ERROR NUMBER 522 DURING PHASE OA.

Explanation: The table containing information about temporary operands has been searched for a temporary which could not be found.

$ COMPILER ERROR NUMBER 524 DURING PHASE OA.

Explanation: The table containing information about Q-temps. has been searched for a Q-temp. which could not be found.

$ COMPILER ERROR NUMBER 529 DURING PHASE OA.

Explanation: The stack of active temporary operands maintained by Phase OA was not empty when a fresh statement was due to be processed.

$ COMPILER ERROR NUMBER 543 DURING PHASE OE.

Explanation: The table containing information about temporary operands has been searched for a temporary which could not be found.

$ COMPILER ERROR NUMBER 544 DURING PHASE OE.

Explanation: The table containing information about temporary operands is full; further entries can not be made. This fact should have been detected and acted upon by Phase OA. The occurrence, therefore, of the above error message also indicates that Phase OA did not fully handle the situation.

$ COMPILER ERROR NUMBER 545 DURING PHASE OE.

Explanation: The table containing information about Q-temps. has been searched for a Q-temp. which could not be found.

$ COMPILER ERROR NUMBER 548 DURING PHASE OE.

Explanation: The stack of active temporary operands maintained by Phase OE was not empty when a fresh statement was due to be processed.

$ COMPILER ERROR NUMBER 602 DURING PHASE KK.

Explanation: Text table stack is full - logic error in Phase KK.

$ COMPILER ERROR NUMBER 641 DURING PHASE OX.

Explanation: A Q-temp. has been referenced which has not been set.

Programmer Response: If possible, rewrite the statement indicated by the error message.

$ COMPILER ERROR NUMBER 642 DURING PHASE OX.

Explanation: The qualified temporary stack is full. This happens when previous phases of the compiler have not flagged qualified temporaries correctly on their last use.

Programmer Response: Look for 30 previous statements in the program which are similar to the one involved. Remove statements until there are less than 30.

$ COMPILER ERROR NUMBER 643 DURING PHASE OX.

Explanation: Input text error. A SELECT, WHEN, or OTHERWISE statement has been encountered with an incorrect value in slot ITSELCT.

Programmer Response: If the program contains nested SELECT groups, simplify the nesting.

$ COMPILER ERROR NUMBER 644 DURING PHASE OX.

Explanation: SELECT stack is full. Logic error in phase OX.

$ COMPILER ERROR NUMBER 645 DURING PHASE OX.

Explanation: SELECT stack contains a bad entry. Logic error in phase OX.

$ COMPILER ERROR NUMBER 661 DURING PHASE KX.

Explanation: An invalid conversion, generated by one of the phases II through OX, has been encountered.

Programmer Response: Simplify the statement referred to by the error message.

$ COMPILER ERROR NUMBER 681 DURING PHASE PC.

Explanation: Phase PC has been asked to construct a symbol table for an invalid identifier. Variables only can occur in data-directed I/O; variables, label constants, or entry-point constants are allowed in CHECK-condition lists. Any invalid or "unsual" identifiers may not have been detected in earlier compiler phases.

Programmer Response: Check the use of data-directed I/O statements or the CHECK condition. Replace any that may cause trouble.

$ COMPILER ERROR NUMBER 683 DURING PHASE PC.

Explanation: A pictured operand or PICTURE format item requiring a DED or FED cannot be associated with its correct PICTURE specification as its dictionary reference has been lost.

Programmer Response: Check the use of PICTURE format items and the passing of pictured variables to library subroutines.

$ COMPILER ERROR NUMBER 721 DURING PHASE PE.

Explanation: An invalid entry has been found during a scan of the variables dictionary.

$ COMPILER ERROR NUMBER 722 DURING PHASE PE.

Explanation: An invalid entry has been found during a scan of the storage dictionary.

$ COMPILER ERROR NUMBER 723 DURING PHASE PE.

Explanation: The compiler has failed to assign correct alignment to a STATIC variable which has been initialized.

Programmer Response: Avoid the use of the INITIAL attribute for STATIC variables.

$ COMPILER ERROR NUMBER 741 DURING PHASE PI.

Explanation: On input to PI, a qualified temporary has been referred to without being previously defined.

Programmer Response: 1. Try to simplify the statement involved.

2. Avoid indirect reference to variables; that is BASED, subscripted POSITION(expression) and SUBSTR.

$ COMPILER ERROR NUMBER 742 DURING PHASE PI.

Explanation: Input to PI indicates need for data element descriptor for a data type which does not require one.

Programmer Response: If a ccnversion is invloved, attempt to avoid conversion.

$ COMPILER ERROR NUMBER 744 DURING PHASE PI.

Explanation: The input to PI tries to take the address of an operand that does not have an address.

Programmer Response: Simplify the statement involved.

$ COMPILER ERROR NUMBER 745 DURING PHASE PI.

Explanation: No storage base has been provided for a variable in the input to PI.

$ COMPILER ERROR NUMBER 746 DURING PHASE PI.

Explanation: Too many temporaries alive at the same time.

Programmer Response: Try to simplify the statememt involved.

$ COMPILER ERROR NUMBER 762 DURING PHASE QI.

Explanation: A text table that should have been deleted by an earlier phase has been found in the input text stream.

$ COMPILER ERROR NUMBER 763 DURING PHASE QI.

Explanation: Invalid input - addressing vector contains incorrect information.

$ COMPILER ERROR NUMBER 781 DURING PHASE QA.

Explanation: Invalid input to the phase.

Programmer Response: Modify the statement referred to, if possible.

$ COMPILER ERROR NUMBER 782 DURING PHASE QA.

Explanation: More registers are required than are available. Caused either by bad input or by logic error in the phase.

Programmer Response: Simplify the statement referred to; for example, perform subscript calculation before the statement.

$ COMPILER ERROR NUMBER 783 DURING PHASE QA.

Explanation: Q-temp. table full, or missing Q-temp.

Programmer Response: Simplify the statement referred to.

$ COMPILER ERROR NUMBER 784 DURING PHASE QA.

Explanation: All storage for register temporaries has been used, or missing register temporary.

Programmer Response: Simplify the statement. If a multiple assignment, ensure that there are not more than 32 targets.

$ COMPILER ERROR NUMBER 785 DURING PHASE QA.

Explanation: Register allocation has found a reference to a base number that should already have been set up, but either it has not been set up or it has been lost.

$ COMPILER ERROR NUMBER 801 DURING PHASE QE.

Explanation: An unrecognizable text table has been found in the input text stream.

$ COMPILER ERROR NUMBER 901 DURING PHASE SK.

Explanation: Raised by missing, invalid, or duplicate label.

$ COMPILER ERROR NUMBER 902 DURING PHASE SK.

Explanation: General register 0 has been used as a base register.

$ COMPILER ERROR NUMBER 903 DURING PHASE SK.

Explanation: An error has been made in the allocation of region numbers.

Programmer Response: Attempt to break up large EDIT or FORMAT statements.

$ COMPILER ERROR NUMBER 904 DURING PHASE SK.

Explanation: Untranslated text table - a text table has not been converted to object code by any of the code generation phases.

$ COMPILER ERROR NUMBER 905 DURING PHASE SK.

Explanation: Too many labels (both user-supplied and compiler-generated) in the program, resulting in overflow of the label table.

Programmer Response: Attempt to simplify the program by reducing the number of labels used.

$ COMPILER ERROR NUMBER 906 DURING PHASE SK.

Explanation: An invalid operation code has been produced by one of the code generation phases.

$ COMPILER ERROR NUMBER 907 DURING PHASE SK.

Explanation: Too many blocks (BEGIN, PROC, and ON) in the program.

Programmer Response: Rerun with larger SIZE parameter.

$ COMPILER ERROR NUMBER 921 DURING PHASE SI.

Explanation: Instructions selected from a code skeleton include a local branch without a corresponding local label.

Programmer Response: Rewrite the statement referred to in the error message.

$ COMPILER ERROR NUMBER 922 DURING PHASE SI.

Explanation: The number of ADCONS requested by phase SK exceeds the number allocated by storage allocation. (The value in XSAADCS exceeds the value in xadcs.)

$ COMPILER ERROR NUMBER 941 DURING PHASE SM.

Explanation: An invalid entry has been found in the pseudo constants pool.

$ COMPILER ERROR NUMBER 942 DURING PHASE SM.

Explanation: An inline constant has been found with an invalid type flag.

Programmer Response: Rewrite the statement referred to in the error message.

$ COMPILER ERROR NUMBER 943 DURING PHASE SM.

Explanation: A marker in the text has an invalid type byte.

Programmer Response: Rewrite the statement referred to in the error message.

$ COMPILER ERROR NUMBER 944 DURING PHASE SM.

Explanation: An invalid dictionary reference has been found in the input text stream.

Programmer Response: Rewrite the statement referred to in the error message.

$ COMPILER ERROR NUMBER 945 DURING PHASE SM.

Explanation: An invalid dictionary reference has been found in one of the input text streams.

$ COMPILER ERROR NUMBER 946 DURING PHASE SM.

Explanaton: An invalid dictionary reference has been found, derived indirectly from text or dictionary.

This Chart is designed to enable it to be readily removed and for convenience is bound at the rear of the manual.

APPENDIX D: CREATION AND USAGE OF DATA AREAS

The chart contained in this appendix indicates the creation and usage of the main data areas used during compilation.

The processing phases are shown in the basic phase loading order: optionally-loaded phases are indicated. Beside each phase are shown the main data areas that can exist in main storage during execution of the phase. Because only one phase plus the resident control phase, (Phase AA), can reside in main storage at any one time, each horizontal section of the chart can be considered as a symbolic representation of the compiler partition of main storage. Where access to a data area for either reading or writing purposes is not shown, some or all of the pages containing that data may remain either in main storage or on the spill data set throughout execution of the relevant phase.

The key indicates other information that is included in the chart.

This Chart is designed to enable it to be readily removed and for convenience is bound at the rear of the manual.

Glossary

This section provides definitions for many of the terms used in this publication.

aggregate: see data aggregate.

array expression: an expression whose evaluation yields an array of values.

back dominator (of a flow unit, F): the flow unit nearest to F through which control must flow before F receives control for the first time.

back target (of a loop): the flow unit nearest to the loop entry unit through which control must pass before the loop is entered. A loop back target is the back dominator of the loop entry unit.

backward connector (of a flow unit, F): a flow unit which can pass control directly to F.

batch processing: a systems approach to processing where a number of similar input items are grouped for processing during the same machine run.

block-header statement: the PROCEDURE or BEGIN statement that heads a block of statements.

book: an invariant sequence of code and/or data definitions that can be introduced into the assembly of a compiler module by use of a COPY statement.

built-in function: a function that is supplied by the language.

chameleon temporary operand: a form of temporary operand generated when an expression is used as an argument and no corresponding parameter descriptor is available, or when an expression is used in a PUT LIST statement. This form of temporary operand is changed during later processing.

communication area (XCOMM): a control section, contained within the resident control phase (Phase AA), that defines an area of storage used for communication between phases.

compile-time statements: special statements appearing in the source program that specify how the source program text is to be altered; they are identified by a leading percent sign (%) and are executed as they are encountered by the compile-time preprocessor (they appear without the percent sign in preprocessor procedures).

connected storage: an area of storage associated with a reference, such as an expression that specifies adjacent elements of an array, which contains data items not associated with the reference.

contextual declaration: the association of attributes with an identifier according to the context in which the identifier appears.

controlled parameter: a parameter for which the CONTROLLED attribute is specified in a DECLARE statement; it can be associated only with arguments that have the CONTROLLED attribute.

controlled storage: storage whose allocation and release is controlled by the programmer, with immediate access to the latest allocation only.

controlled variable: a variable whose allocation and release are controlled by special purpose statements (ALLOCATE and FREE), with access to the current generation only.

core page (core copy): the main storage copy of the text page currently being processed.

data: representation of information or of value.

data aggregate: a logical collection of two or more data items that can be referred to by a single name (possibly subscripted and/or qualified), as well as by individual names; an array or structure.

data element: see data item.

data element descriptor (DED): a control block generated for an edit-directed I/O statement to provide a description of the characteristics of data passed to the PL/I library for conversion or stream I/O. An object-time DED is therefore the equivalent of the data list of an edit-directed I/O statement. Compile-time DEDs have a different format to those that are used during execution and never appear in the executable program. See also descriptor and format element descriptor.

data item: a single unit of data.

data set: a collection of data external to the program that can be accessed by the program by reference to a single file name.

DED: see data element descriptor.

descriptor: a control block which provides additional information to that given in a data element descriptor. It is generated when adjustable extents are involved, or when an item is to be passed as an argument and the associated parameter is the type that can be declared with an asterisk among its attributes. See also data element descriptor.

dictionary: the term dictionary is used to refer to a particular collection of data, used extensively during compilation.

dimensionality: the number of bounds specifications associated with an array declaration.

element: a single data item as opposed to a collection of data items, such as a structure or an array, (sometimes called a scalar item).

element expression: an expression whose evaluation yields an element value.

element variable: a variable that can represent only a single value at any one point in time.

entry expression: the representation of an entry value, that is, an entry constant or an entry variable.

entry unit: a flow unit that begins with a block-entry statement (PROCEDURE, BEGIN, ON-BEGIN, or ENTRY) or which has a label that can be branched-to from an on-unit.

extended code: the mixture of Type-1 and Type-2 text formats and special markers which exists from the start of code generation until the end of compilation. This mixture is due to the fact that more than one phase is involved in the generation of machine code, and because each of them only processes certain types of text tables. Thus, a time exists when the text stream contains machine code that has replaced text tables, and text tables that have yet to be replaced. In addition, special markers are inserted in the text to indicate processing required by later phases.

external procedure: a procedure that is not contained in any other procedure.

factoring: the applicaiton of one or more attributes or of a level number to a list of parenthesized names.

FED: see format element descriptor.

flow unit: a sequence of text defining a sequence of code that has no intermediate branching points, and therefore can only be logically entered at the beginning and left at the end.

format element descriptor (FED): a control block generated for an edit-directed I/O statement or FORMAT statement to provide a description of the data characteristics of the I/O buffer used. A FED is therefore equivalent to the format list of an edit-directed I/O statement. See also data element descriptor.

forward connector (of a flow unit, F): a flow unit to which F can pass control directly.

forward target (of a loop): the flow unit to which control passes when the execution of a loop is complete.

general dictionary: built initially in the Dictionary Build Stage (Phases GA, GI, GE, and GM), the general dictionary is used throughout compilation to store a wide variety of information, such as:

- details of the block structure of the program.
- the format and dimensions of data aggregates.
- descriptions of constant values too great to be conveniently held in text.
- standard default attributes to be applied to implicit declarations.
- collections of information required for optimization.
- descriptions of control blocks, etc., to be generated in the object module.

Entries in the general dictionary are of variable length but have a fixed alignment.

global optimization: optimization performed by a particular section of the compiler, which is only executed in response to programmer specification of the OPTIMIZE compiler option. This option enables the programmer to specify that the program is to be modified in such a way that less time is required for execution of the object program. This optimization may also have a secondary effect of reducing the amount of storage required for the object module.

global temporary operand: a temporary operand that may be used in several statements, either within a flow unit, or within a block.

infix operator: an operator that defines an operation between two operands.

iteration factor: an expression that specifies:

1. the number of consecutive elements of an array that are to be initialized with a given constant.
2. the number of times a given format item or list of format items is to be used in succession in a format list.

iterative group: a do-group whose DO statement specifies a control variable and/or a WHILE option.

level number (of a flow unit, F): a number that is one greater than the smallest number of flow units through which control must pass to get from an entry unit to F. Entry units have a level number of one.

library: a collection of related files or modules.

library routine: a proven routine that is maintained in a program library.

local optimization: optimization performed when particular language features or situations are recognized by various sections of the compiler during any compilation.

local temporary operand: a temporary operand whose use is confined to within one statement.

loop entry unit: a flow unit that starts with the entry point of a loop. A flow unit is a loop entry unit either if it has itself as a backward connector or if it has one or more backward connectors for which it is the back dominator.

macro instruction: an instruction in a source language that is equivalent to a specified sequence of machine instructions.

names dictionary: built entirely in the Dictionary Build Stage (Phases GA, GI, GE, and GM), the names dictionary is used to hold the names of all the variable identifiers and some of the constants that appear in the text. Each entry contains pointers to associated entries in other dictionary sections.

no-storage temporary operand: a temporary operand generated in circumstances where the argument or operand can be considered to be defined or overlaid on an existing data aggregate, and therefore no storage allocation is required for it.

optimization: the generation, by the compiler, of machine code that can be executed more efficiently than code produced by direct translation of the PL/I source program. Two main forms of optimization are performed, local optimization and global optimization.

overflow pages: additional text pages acquired by Phase KA at the rate of one overflow page for every four existing text pages. These pages are used to hold any additional text tables that are created in the Statement Processing Stage (and subsequent stages).

overlay: the technique of repeatedly using the same blocks of internal storage during stages of a program. When one routine is no longer needed in storage, another routine can replace all or part of it.

page: a record containing data-management information and processable data, and used internally in the compiler.

page address: the start address of a page in main storage.

page area: the area of main storage available for the storage of data.

page header: the first 16 bytes of a page, containing data management information. Only part of the page header is copied onto the spill data set during page spilling operations.

page number: each section of the dictionary (names, variable, general, and storage) is built on a separate page or sequence of pages. Within each section, the pages are numbered sequentially, starting at zero. This page number is contained in a field in the page header.

page size: the minimum page size is 1300 bytes, of which 20 bytes are used to hold data-handling information, and 1280 bytes are available for processable data. If storage is available for fourteen or more pages of this size, the page size is increased to 2580 bytes, of which 2560 bytes can be used for processable data. The page size is calculated by a routine contained in the initialization phase, Phase AE.

page status: a page in main storage is always given a specific status. The status, which can be one of six grades, indicates the relationship between the core copy and the spill copy, and the accessibility of the core page. The six status grades are:

- UNMOVABLE READ/WRITE
- UNMOVABLE READ-ONLY
- SPILLABLE (MOVABLE READ/WRITE)
- USABLE (MOVABLE READ-ONLY)
- DISCARDED
- UNUSED

phase: a series of executable instructions and definitions contained in a load module which can be individually loaded and executed to perform one or more functions required in the process of compilation.

phase area: that area of main storage into which each phase of the compiler (with the exception of the resident control phase, Phase AA) is loaded in turn.

postfix Polish notation: a method of forming mathematical expressions in which each operator is preceded by its operands.

preprocessed text: the output from the compile-time statement preprocessor. This output is a sequence of characters that is altered source program text and which serves as input to those stages of the compiler in which the actual compilation is performed.

preprocessor statements: see compile-time statements.

Q-temp: see qualified temporary operand.

qualified temporary operand (Q-temp.): a temporary operand generated to replace qualified items such as array-elements or based variables. A Q-temp. contains a description of the data type of the item referred to, and provides a means of identifying the qualifying information.

register usage table (RUT): created and maintained in the phase working area, this table indicates the current content of registers and the items within a flow unit that are already held in registers. It is updated as registers are allocated.

resident and transient libraries: these libraries consist of sets of standard subroutines that are used for the majority of interfaces with the system and for those jobs that can be most efficiently done by the use of interpretive subroutines. Resident library modules are called by and link-edited with the executable program phase. Transient library modules are called from resident library modules, and are loaded into dynamically-allocated storage when they are required; when they are no longer needed, the storage is freed and may be overwritten.

repetition factor: a parenthesized unsigned decimal integer constant preceding a string configuration as a shorthand representation of a string constant. The repetition factor specifies the number of occurrences that make up the actual constant. In picture specifications, the repetition factor specifies repetition of a single picture character.

root module: the segment of an overlay program that remains in main storage at all times during the execution of the overlay program.

RUT: see register usage table.

scalar item: a single item of data; an element.

scalar variable: a variable that can represent only a single data item; an element variable.

scratch page: a page used as temporary workspace by one or more phases.

source module: a series of statements in the symbolic language of an assembler or compiler, which constitutes the entire input to a single execution of the assembly or compiler.

source program: the program that serves as input to the compiler. The source program may contain compile-time statements.

spill candidate: the first suitable page found in the search of the page chains for either a new page or an existing page.

spill data set: a data set used to hold internal data (text, dictionary, etc.,) that is not currently being accessed or processed, and which cannot be retained in main storage.

spill page (spill copy): the spill data set copy of the text page currently being processed.

stage: the compilation process performed by this compiler consists of a number of major operations, which are performed in sequence by the execution of some or all of the 52 phases that make up the compiler. In relation to these major operations, the phases can be collected logically into ten groups which, for descriptive purposes, are referred to as stages.

storage dictionary: used to hold information about the amount and location of storage required for every identifier in the object module. It is built by the storage allocation phase (Phase PE) when most of the information about the object code to be

generated has been determined, and can be considered as an extension of the variables dictionary.

strength reduction: reduction of the complexity of operations performed within a loop. It may also involve some common expression elimination and backward movement of text.

symbol table: a control block generated during compilation to hold the name of the variable during execution and associate it with the address of the variable. Used only when data-directed I/O or the CHECK condition is specified.

temporary operands: because PL/I statements can contain an unlimited number of operands, it is frequently necessary to set up fields containing intermediate results. These fields are known as temporary operands.

text: the main stream of internal data, consisting of the internal representation of statements and other items of information, originally corresponding to the PL/I source program and progressively transformed by phases of the compiler into the format required at output.

transient library: see resident and transient libraries.

Type-1 text format: the sequential text stream in which all statements in a block appear before the first statement in a block at the next level of nesting. Within each block statements are retained in source-program order, and within each statement, statement elements are also retained in source order. The text retains the basic characteristics of Type-1 text format from its generation in the Syntax Analysis Stage until it is processed by Phase II in the Text Formatting Stage.

Type-2 text format: Phase II in the Text Formatting Stage translates the text from a stream of statements (see Type-1 text format) into a series if fixed-length text tables, each 32 bytes long. Text in this format is referred to as Type-2 text.

variables dictionary: built initially in the Dictionary Build Stage, the variables dictionary contains an entry for each variable in the text. Each entry contains code indicating the attributes of the variable, and a reference to the corresponding entry in the names dictionary.

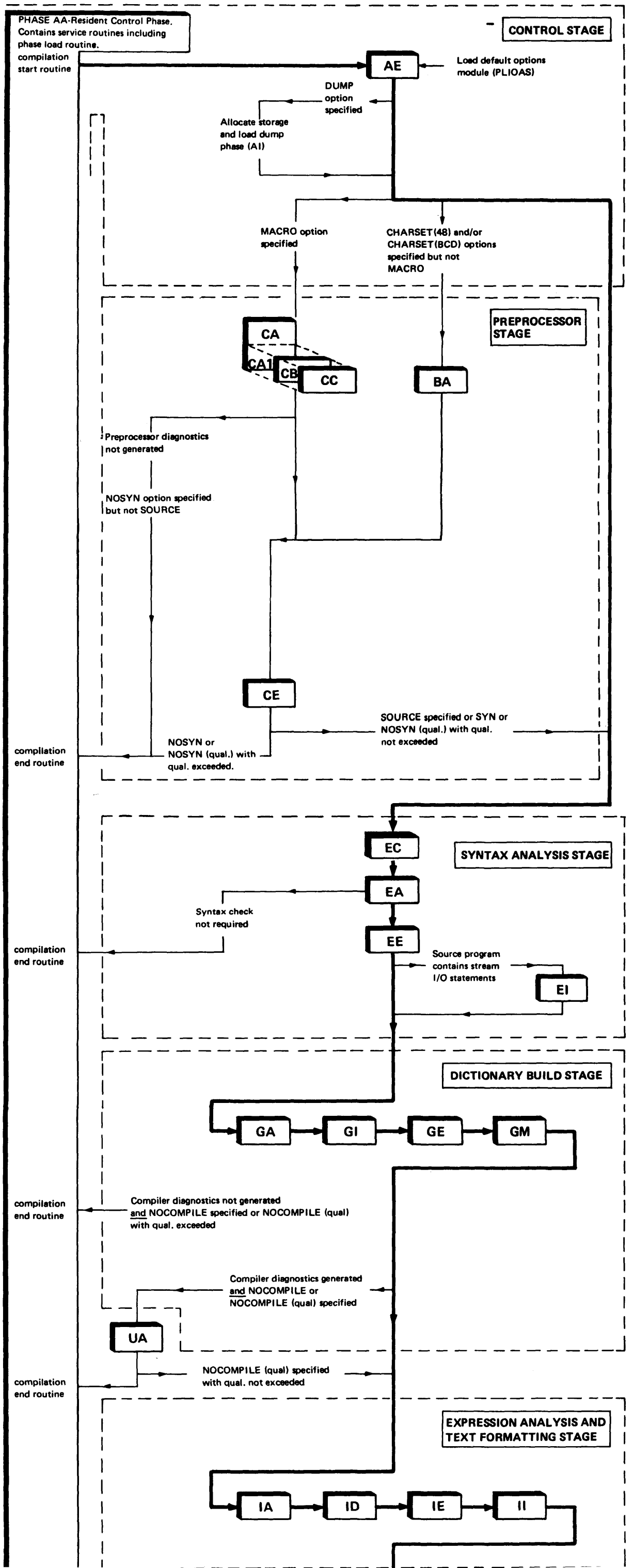
XCOMM: see communication area.

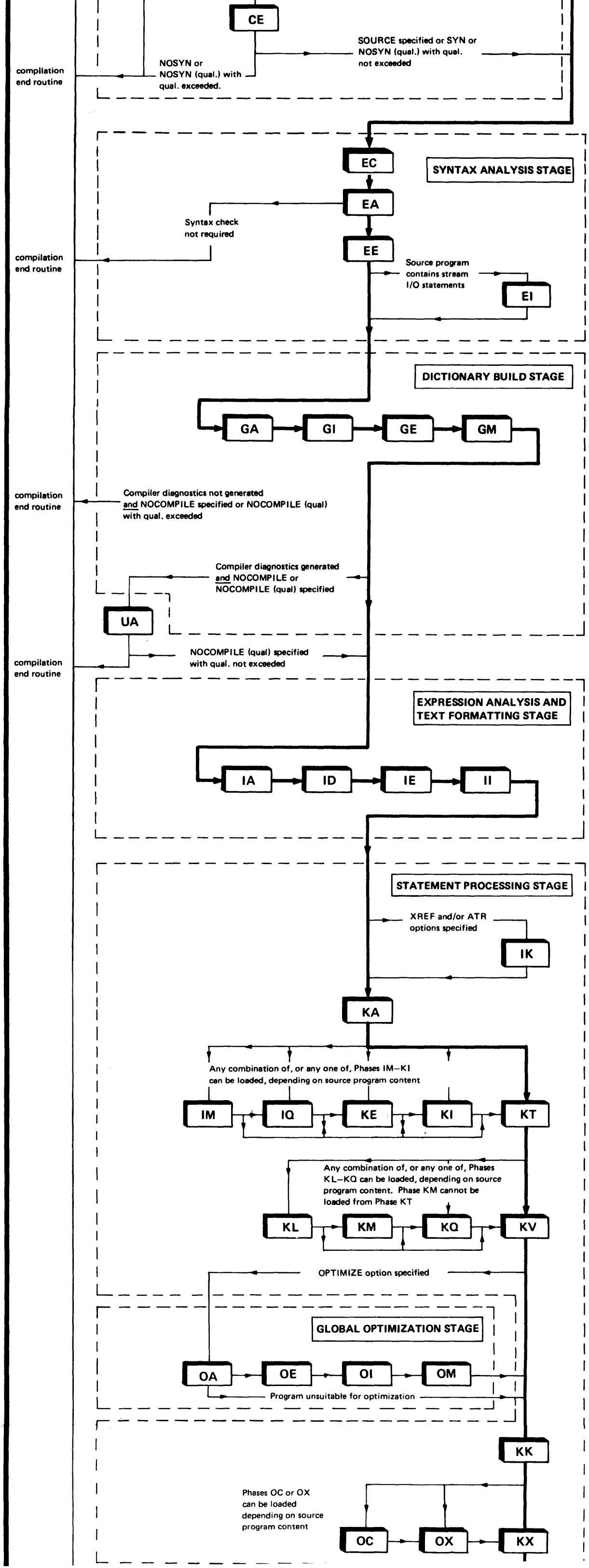
Index

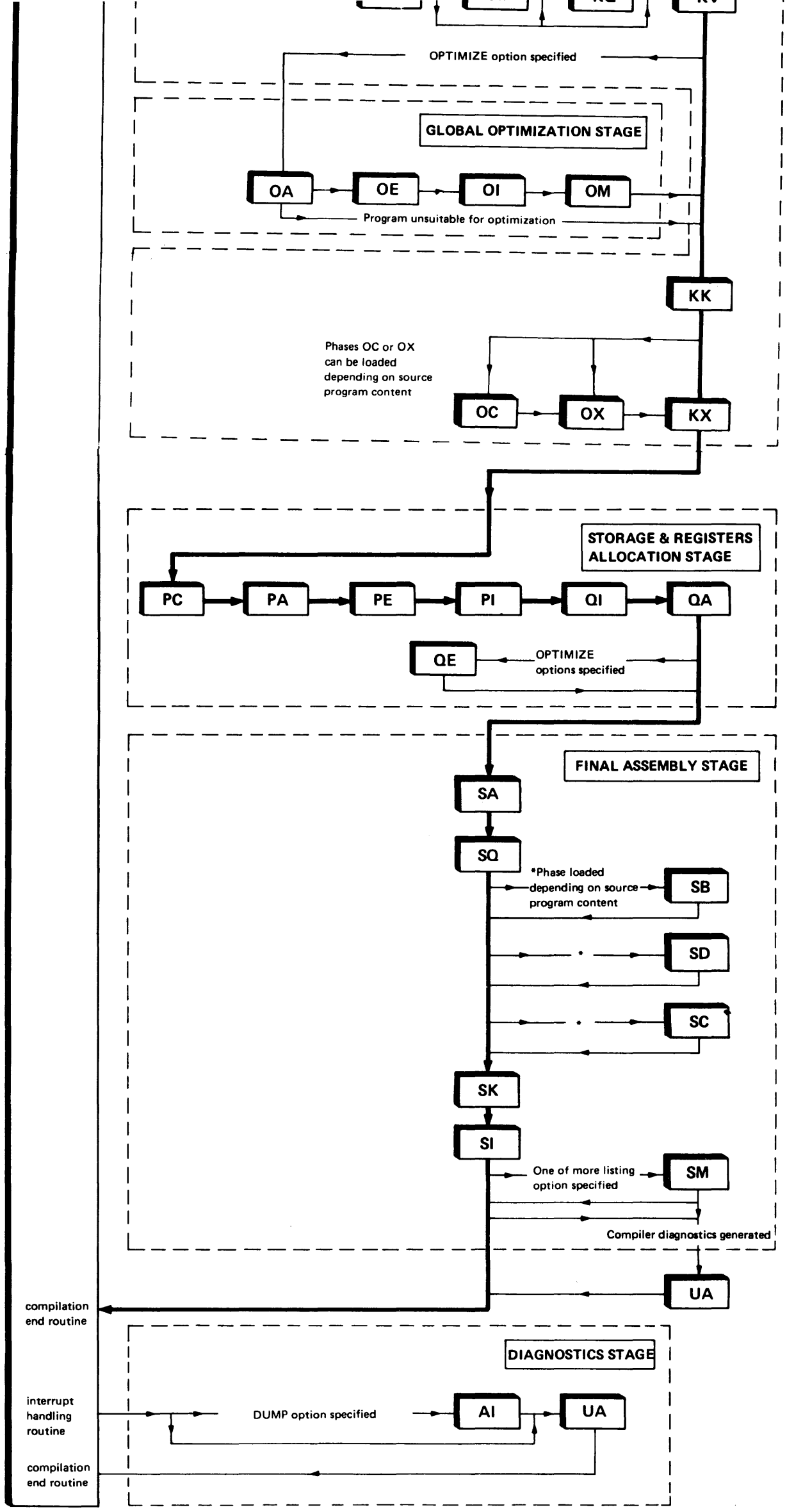
Note: This index does not contain entries for language features. It is intended to be used in conjunction with the directory in section 4 (pages 494-549), which indicates the phases that process particular language features.

48-character preprocessor (see Phase BA)

APPENDIX C SEQUENCE OF PHASE LOADING




CE
SOURCE specified or SYN or NOSYN (qual.) with qual. not exceeded
compilation end routine
NOSYN or NOSYN (qual.) with qual. exceeded
EC
SYNTAX ANALYSIS STAGE
EA
Syntax check not required
EE
compilation end routine
Source program contains stream I/O statements
EI
DICTIONARY BUILD STAGE
GA
GI
GE
GM
compilation end routine
Compiler diagnostics not generated and NOCOMPILE specified or NOCOMPILE (qual) with qual. exceeded
Compiler diagnostics generated and NOCOMPILE or NOCOMPILE (qual) specified
UA
NOCOMPILE (qual) specified with qual. not exceeded
compilation end routine
EXPRESSION ANALYSIS AND TEXT FORMATTING STAGE
IA
ID
IE
II
STATEMENT PROCESSING STAGE
XREF and/or ATR options specified
IK
KA
Any combination of, or any one of, Phases IM–KI can be loaded, depending on source program content
IM
IQ
KE
KI
KT
Any combination of, or any one of, Phases KL–KQ can be loaded, depending on source program content. Phase KM cannot be loaded from Phase KT
KL
KM
KQ
KV
OPTIMIZE option specified
GLOBAL OPTIMIZATION STAGE
OA
OE
OI
OM
Program unsuitable for optimization
KK
Phases OC or OX can be loaded depending on source program content
OC
OX
KX


OPTIMIZE option specified
GLOBAL OPTIMIZATION STAGE
OA
OE
OI
OM
Program unsuitable for optimization
KK
Phases OC or OX
can be loaded
depending on source
program content
OC
OX
KX
STORAGE & REGISTERS
ALLOCATION STAGE
PC
PA
PE
PI
QI
QA
QE
OPTIMIZE
options specified
FINAL ASSEMBLY STAGE
SA
SQ
*Phase loaded
depending on source
program content
SB
SD
SC
SK
SI
One of more listing
option specified
SM
Compiler diagnostics generated
UA
compilation
end routine
DIAGNOSTICS STAGE
interrupt
handling
routine
DUMP option specified
AI
UA
compilation
end routine

APPENDIX D CREATION AND USAGE OF DATA AREAS

RESIDENT		TRANSIENT	HANDLED ON TEXT PAGES			HANDLED ON DICTIONARY PAGES					
CON-TROL PHASE	COMM. AREA	PROCESSING PHASE	MAIN TEXT STRM.	SECONDARY TEXT STREAM OR MISCELLANEOUS TABLES	DIAG. AND ERROR MSGS.						
PREPROCESSOR STAGE											
AA	R W	PHASE BA * 48 CHAR/BCD PREPROCESSOR	R W SF		(W) 1						
AA	R W	PHASE CA * COMPILE-TIME PREPROCESSOR	R W SF	R W 2	(W) 1	R W 3			R W 4		
AA	R W	PHASE CE * PREPROC. DIAGN-MSG. EDITOR		R W 21	R W 5	R 1			R 4		
SYNTAX ANALYSIS STAGE											
AA	R W	PHASE EA SYNTAX ANALYSIS – PASS 1	R W T1		R W 6	(W)					
AA	R W	PHASE EE SYNTAX ANALYSIS – PASS 2	R W T1	R W 7 &		(W)					
AA	R W	PHASE EI ** SYNTAX ANALYSIS – PASS 3	R W T1	7 &		(W)					
DICTIONARY BUILD STAGE							NAMES DICT.	VAR. DICT.	GEN. DICT.		
AA	R W	PHASE GA EXPLICIT DECLARATIONS	R T1	R 7 &		R W 8 @	(W)	W	W	W	
AA	R W	PHASE GI CONTEXTUAL DECLARATIONS	R T1	R 7 &		R W 8 @	(W)	R W	R W	R W	
AA	R W	PHASE GE DECLARATION EXPRESSIONS		R 7	W 9 &	R W 8 @	(W)	R W	R W	R W	
AA	R W	PHASE GM IMPLICIT DECLARATIONS	R W T1		R 9 &	R 8 @	(W)	R W	R W	R W	
EXPRESSION ANALYSIS AND TEXT FORMATTING STAGE											
AA	R W	PHASE IA MERGE DECLA-RATION EXPS.	R W T1		R 9 &		(W)		R	R	
AA	R W	PHASE ID MATCHING OF DATA-AGG. ARGS.	R W T1				(W)		R W	R W	
AA	R W	PHASE IE AGGREGATE EXPANSION	R W T1				(W)	R	R	R	
AA	R W	PHASE II EXP. ANALYSIS & TEXT FORMATTING	R W T2				(W)		R W	R	
STATEMENT PROCESSING STAGE – PART 1											
AA	R W	PHASE IK * ATTRIBUTES & X-REF. LISTING	R T2			R W 10	(W)	R W	R	R	
AA	R W	PHASE KA ** IF STATEMENT PROCESSING	R W T2				(W)		R W	R	
AA	R W	PHASE IM ** LANGUAGE COMMUNICATION	R W T2				(W)		R W	R W	
AA	R W	PHASE IQ ** ARRAY & STRUC-TURE MAPPING	R W T2				(W)		R W	R W	
AA	R W	PHASE KE ** SUBSCRIPT PROCESSING	R W T2				(W)		R	R	
AA	R W	PHASE KI ** DO STATEMENT PROCESSING	R W T2				(W)				
AA	R W	PHASE KT SYST. INTERFACE STMT. PROCESSING	R W T2				(W)	R	R	R	
AA	R W	PHASE KL ** OPEN, CLOSE, & FILE DECLNS.	R W T2				(W)	R W	R	R W	
AA	R W	PHASE KM ** RECORD I/O STMT PROCESSING	R W T2				(W)	R	R	R W	
AA	R W	PHASE KQ ** STREAM I/O STMT PROCESSING	R W T2				(W)		R	R W	
AA	R W	PHASE KV SPECIAL CASE PROCESSING	R W T2				(W)		R	R	
GLOBAL OPTIMIZATION STAGE											
AA	R W	PHASE OA * EXTRN. OF ALIAS	R T2				(W)		R W	R W	

		IMPLICIT DECLARATIONS	T1		9 &		8 @					

EXPRESSION ANALYSIS AND TEXT FORMATTING STAGE

AA	R W	PHASE IA MERGE DECLA-RATION EXPS.	R W T1		R 9 &			(W)		R		R
AA	R W	PHASE ID MATCHING OF DATA-AGG. ARGS.	R W T1					(W)		R W		R W
AA	R W	PHASE IE AGGREGATE EXPANSION	R W T1					(W)	R	R		R
AA	R W	PHASE II EXP. ANALYSIS & TEXT FORMATTING	R W T2					(W)		R W		R

STATEMENT PROCESSING STAGE – PART 1

AA	R W	PHASE IK * ATTRIBUTES & X-REF. LISTING	R T2				R W 10	(W)	R W	R		R
AA	R W	PHASE KA ** IF STATEMENT PROCESSING	R W T2					(W)		R W		R
AA	R W	PHASE IM ** LANGUAGE COMMUNICATION	R W T2					(W)		R W		R W
AA	R W	PHASE IQ ** ARRAY & STRUC-TURE MAPPING	R W T2					(W)		R W		R W
AA	R W	PHASE KE ** SUBSCRIPT PROCESSING	R W T2					(W)		R		R
AA	R W	PHASE KI ** DO STATEMENT PROCESSING	R W T2					(W)				
AA	R W	PHASE KT SYST. INTERFACE STMT. PROCESSING	R W T2					(W)	R	R		R
AA	R W	PHASE KL ** OPEN, CLOSE, & FILE DECLNS.	R W T2					(W)	R W	R		R W
AA	R W	PHASE KM ** RECORD I/O STMT PROCESSING	R W T2					(W)	R	R		R W
AA	R W	PHASE KQ ** STREAM I/O STMT PROCESSING	R W T2					(W)		R		R W
AA	R W	PHASE KV SPECIAL CASE PROCESSING	R W T2					(W)		R		R

GLOBAL OPTIMIZATION STAGE

AA	R W	PHASE OA * EXTRN. OF ALIAS & CALL INFORM.	R T2					(W)		R W		R W
AA	R W	PHASE OE * EXTRN. OF VBLS-USAGE & FLO INF	R W T2					(W)		R		R W
AA	R W	PHASE OI * LOOP ANALYSIS	R W T2					(W)		R		R W
AA	R W	PHASE OM * TEXT OPTIMIZATION	R W T2				R W 11	(W)		R		R

STATEMENT PROCESSING STAGE – PART 2

AA	R W	PHASE KK BIF AND PSV PROCESSING	R W T2					(W)		R		R W
AA	R W	PHASE OC ** STRING HANDLING OPTNS. – PART 1	R W T2					(W)				R W
AA	R W	PHASE OX ** STRING HANDLING OPTNS. – PART 2	R W T2					(W)	R	R		R W
AA	R W	PHASE KX DATA CONVERSIONS	R W T2					(W)				R W

STORAGE AND REGISTERS ALLOCATION STAGE

AA	R W	PHASE PC SYMBOL TABLE RESOLUTION	R W T2	W 13 @			W 12 &	(W)	R	R W		R W
AA	R W	PHASE PA CONSTANTS ANALYSIS	R W T2	R W 13 @	R W 14		W 12 &	(W)		R W	STG. DICT.	R W
AA	R W	PHASE PE STORAGE ALLOCATION		R W 13 @		W 15 &	12	(W)		R	W	R W
AA	R W	PHASE PI ADDRESSING OF STORAGE	R W T2	R 13 @	W 16 &	R 15	12	(W)			R	R
AA	R W	PHASE QI OPTIMIZED ADDRESSING	R W T2		R 16		12	(W)			R	
AA	R W	PHASE QA REGISTER ALLOCATION	R W T2	R W 17			12	(W)				
AA	R W	PHASE QE * ELIMTION OF UN-NEC. STGE. OPS.	R W T2				12					

FINAL ASSEMBLY STAGE

AA	R W	PHASE SA OBJECT CODE GENRTN. – PASS 1	R W EC				12				R	R W
AA	R W	PHASE SQ OBJECT CODE	R W EC				12				R	R

AA	R W	PHASE OM * TEXT OPTIMIZATION	R W T2				R W 11	(W)		R		R
		STATEMENT PROCESSING STAGE – PART 2										
AA	R W	PHASE KK BIF AND PSV PROCESSING	R W T2					(W)		R		R W
AA	R W	PHASE OC ** STRING HANDLING OPTNS. – PART 1	R W T2					(W)				R W
AA	R W	PHASE OX ** STRING HANDLING OPTNS. – PART 2	R W T2					(W)	R	R		R W
AA	R W	PHASE KX DATA CONVERSIONS	R W T2					(W)				R W
		STORAGE AND REGISTERS ALLOCATION STAGE										
AA	R W	PHASE PC SYMBOL TABLE RESOLUTION	R W T2	W 13 @			W 12 &	(W)	R	R W		R W
AA	R W	PHASE PA CONSTANTS ANALYSIS	R W T2	R W 13 @	R W 14		W 12 &	(W)		R W	STG. DICT.	R W
AA	R W	PHASE PE STORAGE ALLOCATION		R W 13 @		W 15 &	12	(W)		R	W	R W
AA	R W	PHASE PI ADDRESSING OF STORAGE	R W T2	R 13 @	W 16 &	R 15	12	(W)			R	R
AA	R W	PHASE QI OPTIMIZED ADDRESSING	R W T2		R 16		12	(W)			R	
AA	R W	PHASE QA REGISTER ALLOCATION	R W T2	R W 17			12	(W)				
AA	R W	PHASE QE * ELIMTION OF UN-NEC. STGE. OPS.	R W T2				12					
		FINAL ASSEMBLY STAGE										
AA	R W	PHASE SA OBJECT CODE GENRTN. – PASS 1	R W EC				12				R	R W
AA	R W	PHASE SQ OBJECT CODE GENRTN. – PASS 2	R W EC				12				R	R
AA	R W	PHASE SD ** OBJECT CODE GENRTN. – PASS 4	R W EC				12				R	R
AA	R W	PHASE SC ** OBJECT CODE GENRTN. – PASS 5	R W EC				12				R	
AA	R W	PHASE SK LABEL RESOLUTION	R W EC			W 18 @	R 12	(W)			R W	R W
AA	R W	PHASE SI OBJECT MODULE ASSEMBLY	R W EC	W 19 &	W 20	R 18 @	R 12	(W)	R		R W	R W
AA	R W	PHASE SM * OBJECT MODULE LISTINGS	R W EC	R 19 &	R 20		R 12	(W)	R	R	R	R
		DIAGNOSTICS STAGE										
AA	R	PHASE UA DIAGNOSTIC-MESSAGE EDITOR				R W 21		R				

KEY

* = Optional phase, depending on compiler options.
** = Optional phase, depending on source program content.
R = Read
W = Write
& = Second text stream, chained from X2STRM in XCOMM.
@ = Scratch pages, reference in XSCRCH in XCOMM.
1 = Preprocessor Diagnostic and Error Message Stream.
2 = Preprocessor Internal Format Text.
3 = Preprocessor IVB Dictionary.
4 = Preprocessor General Dictionary.
5 = Preprocessor Sorted Message Stream.
6 = Read-in Area.
7 = Dictionary Text File.
8 = Hash Table and Default Directory.
9 = Declarations Expressions File.
10 = Cross Reference Tables.
11 = Subscript-list Sort Page.
12 = Pseudo Constants Pool.
13 = Constants Relocation and Base Numbers Information Page.
14 = Constant Descriptor Tables.
15 = Block Structure Information Page.
16 = Addressing and Temporary Storage Information.
17 = Register Usage Table.
18 = Label Table.
19 = Internal Representation of Object Module.
20 = Static CSECT (Adcons) Information.
21 = Diagnostic Message Sort Page.

Text Formats in Main Text Stream (Output from Phase)

SF = Source Format.
T1 = Type 1 Text.
T2 = Type 2 Text.
EC = Extended Code.

IBM / **Technical Newsletter**

This Newsletter No.	LN33-6175
Date	October, 1976
Base Publication No.	LY33-6010-1
File No.	S360/S370-29
Previous Newsletters	LN33-6079
	LN33-6115

DOS
PL/I Optimizing Compiler:
Program Logic

This Technical Newsletter, a part of Version 1, Release 5, Modification 0 of the IBM DOS PL/I Optimizing Compiler, provides replacement pages for the subject publication. These replacement pages remain in effect for subsequent versions, releases, and modifications of the compiler unless specifically altered. Pages to be inserted and/or removed are:

Front cover/Edition notice	162.1 (added)	505, 506
Contents (sheets 1 to 5)	173–178	521, 522
Figures (sheet 1)	178.1 (added)	537, 538
7–13	179–184	561, 562
17–20	187–196	573, 574
37, 38	196.1 (added)	589–594
38.1 (added)	197,198	609–614
47–52	203,204	617, 618
55–60	207, 208	627, 628
67–70	211, 212	633–636
70.1 (added)	225, 226	645–650
103, 104	273–276	657, 658
104.1 (added)	303–306	673, 674
107–110	306.1 (added)	676
110.1 (added)	307–314	679, 680
119, 120	335–338	687, 688
120.1 (added)	361, 362	699–702
123, 124	367–370	705–706.4
124.1 (added)	397, 398	719, 720
153, 154	463, 464	Appendix C (Foldout)
159–162	499, 500	Reader's comment form

A change to the text is indicated by a vertical line to the left of the change.

Summary of Amendments

Changes for Release 5 of the optimizing compiler and general updating of the manual.

Note: *Please file this cover letter at the back of the manual to provide a record of changes.*

IBM United Kingdom Laboratories Ltd, Publications Dept, Hursley Park, Winchester, Hants, England.

IBM / **Technical Newsletter**

This Newsletter No.	LN33-6115
Date	June, 1974
Base Publication No.	LY33-6010-1
File No.	S360/S370-29
Previous Newsletters	LN33-6079

DOS PL/I Optimizing Compiler:
Program Logic

This Technical Newsletter, a part of Version 1, Release 4, Modification 1 of the DOS PL/I Optimizing Compiler, provides replacement pages for the subject publication. These replacement pages remain in effect for subsequent modifications and releases unless specifically altered. Pages to be inserted/removed are:

Cover, Edition Notice	253-258
Contents (3rd sheet)	258.1, 258.2 (added)
187, 188	583, 584
231, 232	589-592
247-250	715-720

A change to the text or to an illustration is indicated by a vertical line to the left of the change.

Summary of Amendments

The changes document modifications and improvements to the DOS PL/I Optimizing Compiler for Release 4.1.

Note: *Please file this cover letter at the back of the manual to provide a record of changes.*

IBM United Kingdom Laboratories Ltd, Publications Dept, Hursley Park, Winchester, Hants, England.

Printed in U.S.A.

IBM

Technical Newsletter

This Newsletter No.	LN33-6079
Date	October, 1973
Base Publication No.	LY33-6010-1
System	S360/S370-29
Previous Newsletters	None

DOS PL/I Optimizing Compiler: Program Logic

This Technical Newsletter, a part of Version 1, Release 4, Modification 0 of the DOS PL/I Optimizing Compiler, provides replacement pages for the subject publication. These replacement pages remain in effect for subsequent modifications and releases unless specifically altered. Pages to be inserted/removed are:

Cover, Edition notice
3,4
Contents
Figures
Charts (added)
5–14
17–20
23,24
24.1,24.2 (added)
35–40
47,48
67–70
133,134
159,160
187,188
231–234
249,250
261,262
299,300
303–306
311,312
315–320
325,326
335–338
401,402
441,442
487,488
521,522
563–566
565.1 (added)
573,574
671,672
676
687–690
699–706
706.1–706.4 (added)
715–726 (Index)

A change to the text or to an illustration is indicated by a vertical line to the left of the change; some pages do not have any technical changes but are issued because of text rearrangement.

IBM United Kingdom Laboratories Ltd, Publications Dept, Hursley Park, Winchester, Hants, England.

Printed in U.S.A.

IBM / **Technical Newsletter**

This Newsletter No.	LN33-6079
Date	October, 1973
Base Publication No.	LY33-6010-1
System	S360/S370-29
Previous Newsletters	None

continued/

Summary of Amendments

The changes document modifications and improvements made to the DOS PL/I Optimizing Compiler for Release 4.0. The most significant of these are the statement frequency count option, and support for the IBM 3340 direct access storage device.

Note: *Please file this cover letter at the back of the manual to provide a record of changes.*

IBM United Kingdom Laboratories Ltd, Publications Dept, Hursley Park, Winchester, Hants, England.

Printed in U.S.A.

DOS PL/I Optimizing Compiler:
Program Logic

Order No. LY33-6010-1

Reader's Comment Form

Your comments about this publication will help us to improve it for you. Comment in the space below, giving specific page and paragraph references whenever possible. All comments become the property of IBM.

Please do not use this form to ask technical questions about IBM systems and programs or to request copies of publications. Rather, direct such questions or requests to your local IBM representative.

If you would like a reply, please provide your name and address (including ZIP code).

Fold on two lines, staple, and mail. No postage necessary if mailed in the U.S.A. (Elsewhere, any IBM representative will be happy to forward your comments.) Thank you for your cooperation.

Fold and Staple

First Class Permit
Number 6090
San Jose, California

Business Reply Mail

No postage necessary if mailed in the U.S.A.

Postage will be paid by:

IBM Corporation
P.O. Box 50020
Programming Publishing
San Jose, California 95150

Fold and Staple

DOS PL/I Optimizing Compiler: Program Logic File No. S360/S370-29 Printed in U.S.A. LY33-6010-1

IBM®

International Business Machines Corporation
Data Processing Division
1133 Westchester Avenue, White Plains, New York 10604
(U.S.A. only)

IBM World Trade Corporation
832 United Nations Plaza, New York, New York 10017
(International)

DOS PL/I Optimizing Compiler: Program Logic Printed in U.S.A. LY33-6010-1

International Business Machines Corporation
Data Processing Division
1133 Westchester Avenue, White Plains, New York 10604
[U.S.A. only]

IBM World Trade Corporation
821 United Nations Plaza, New York, New York 10017
[International]