SOFTWARE SUPPORT MANUAL

ASSEMBLY LANGUAGE REFERENCE MANUAL

PROGRAMMING SUPPORT SYSTEM (PSS-B)

(TACFIRE)

(LITTON DATA SYSTEMS)
DAAB07-68-C-0154

PREPARED FOR U.S. ARMY COMPUTER SYSTEMS COMMAND FORT BELVOIR, VIRGINIA

15 January 1972

LIST OF EFFECTIVE PAGES

Insert latest changed pages; dispose of superseded pages in accordance with applicable regulations.

NOTE: On a changed page, the portion of the text affected by the latest change is indicated by a vertical line, or other change symbol, in the outer margin of the page. Changes to illustrations are indicated by miniature pointing hands. Changes to wiring diagrams are indicated by shaded areas.

Total number of pages in this manual is 55 consisting of the following:

Page No.	*Change No.	Page No.	*Change No.	Page No.	*Change No.
Title	0				
A	0				
B Blank	0				
i – iii	0				
iv Blank	0				
1-1	0				
1-2 Blank	0				
2-1 – 2-4	0				
3-1 – 3-8	0				
4-1 – 4-6	0				
5-1 – 5-5	0				
5-6 Blank	0				
6-1 – 6-2	0				
7-1 – 7-8	0				
8-1 – 8-5	0				
8-6 Blank	0				
9-1 – 9-4	0				
10-1 – 10-2	0				

*Zero in this column indicates an original page.

TABLE OF CONTENTS

TABLE OF CONTENTS (Continued)

LIST OF ILLUSTRATIONS

LIST OF TABLES

LIST OF TABLES (Continued)

CHAPTER 1
INTRODUCTION

Section I. GENERAL

1-1. Purpose

The purpose of this manual is to provide information necessary for programmers to prepare AN/GYK-12 computer assembly programs. Information about the AN/GYK-12 computer assembler language and the assembler itself is presented.

1-2. Applicable Documents

The AN/GYK-12 Principles of Operation Manual USACSCS-TF-4-3 provides information prerequisite to this manual.

1-3. Description of the Manual

Chapter 1 introduces the manual. The purpose and applicable documents are given, and a description of the contents of the manual is provided. Chapter 2 presents general information about the assembler and the assembler language. Information required to construct an input deck to the assembler is presented. In addition, discussions pertaining to communicating with the AN/GYK-12 computer, programming in the AN/GYK-12 assembler language, and the AN/GYK-12 assemblier itself are included. Chapter 3 discusses the different types of assembler language source statements, the assembler instruction fields, and the assembler input format. The basic elements of the operand (symbols, terms, and expressions) are also discussed. Chapter 4 presents detailed discussions of the components of the operand field of machine instructions. The content and format of the different kinds of machine instructions is explained. Addressing modes and indexing are defined and explained. Examples are provided as useful reference material. Chapter 5 presents complete information about the extended mnemonics. Each extended mnemonic instruction is explained in detail and the instruction format used by the assembler is described. Chapter 6 contains detailed descriptions of the use of Compool symbols in an assembler language program. Examples are provided. Chapter 7 discusses the AN/GYK-12 assembler instructions (pseudo operations). The fourteen instructions are organized into groupings, defined, and explained in terms of operand content, format, and usage. Examples are included. Chapter 8 discusses the AN/GYK-12 assembler macro language. Examples are provided to support the detailed definitions and explanations. Chapter 9 explains the assembler object code output. Tables and figures are provided to show the various object code formats. Chapter 10 describes the content and format of the assembly listing. Included are discussions of the program listings, the external symbol dictionary, the cross reference and set/used listing, the assembly error indications, and the error and error symbol counts. Appendix A presents an example of AN/GYK-12 assembler output. Appendix B is a table of the EBCDIC character codes. Appendix C is a table of the ASCII character codes, and Appendix D consists of a summary table of the AN/GYK-12 machine function codes.

CHAPTER 2
FUNCTIONS OF THE AN/GYK-12 ASSEMBLER

Section I. GENERAL

2-1. Introduction

This chapter presents general information about the AN/GYK-12 assembler. The structure and function of the assembler are discussed and the various programmer features of the assembler are outlined.

2-2. Communicating with the AN/GYK-12 Computer

The AN/GYK-12 assembler program instructs the AN/GYK-12 computer to assemble or translate the symbolic assembler language of the source program into AN/GYK-12 machine code instructions to form the object program. The assembly process is illustrated in figure 2-1. The source program consists of a series of instructions. There are several types of instructions in an assembler language, and each type is discussed in a different area in the manual. Machine instructions are discussed in chapters 3 and 4. The assembler instructions (pseudo operations) are discussed in Chapter 7, and the macro instructions are discussed in chapter 8. The actual machine language program will be constructed as indicated by the machine instructions. The assembler instructions control the assembly process by giving directions to the assembler. The assembler will not generate any machine code from these instructions. Macro instructions are used to conveniently generate a desired sequence of instructions many times in one or more programs. A sequence of instructions may be generated as a macro at will by the programmer. This process is called defining a macro. When a macro is referenced by the programmer and used in a program, the process is referred to as macro expansion. The source program is composed of all types of instructions. The source program is written on a special coding form to be keypunched. This form is illustrated in figure 2-2. Any format requirements implied by the form may not hold, as the assembler provides a generally variable field format. The keypunched source program (usually on cards) becomes the input to the assembly process shown in figure 2-1. There are two outputs from the assembler program. The first is an object program output consisting of actual machine instructions which correspond to the source program instructions presented to the assembler program as input. The object program may be punched on cards, output to magnetic tape, or output to disk. The second output is a program listing or assembly listing. This listing shows the original source program instructions side by side with the object program instructions produced from them. Other programmer aids such as error indications, a symbol dictionary, and a cross reference listing are provided.

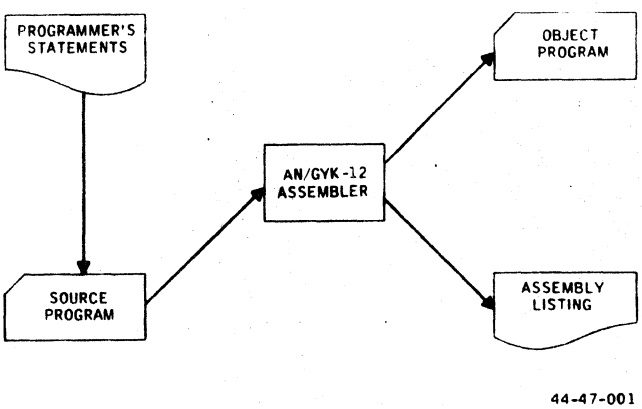


Figure 2-1. Assembly Process

2-3. Programming in the AN/GYK-12 Assembler Language

Programming in the AN/GYK-12 assembler language offers a number of important advantages over programming in the actual machine language of the AN/GYK-12 computer.

a. Mnemonic operation codes are provided. For instance, the actual operation code for the add logical full instruction is hexadecimal 0A. In the AN/GYK-12 assembler language the programmer can write the mnemonic operation code ALF. Most programmers never need to learn the actual numeric operation codes.

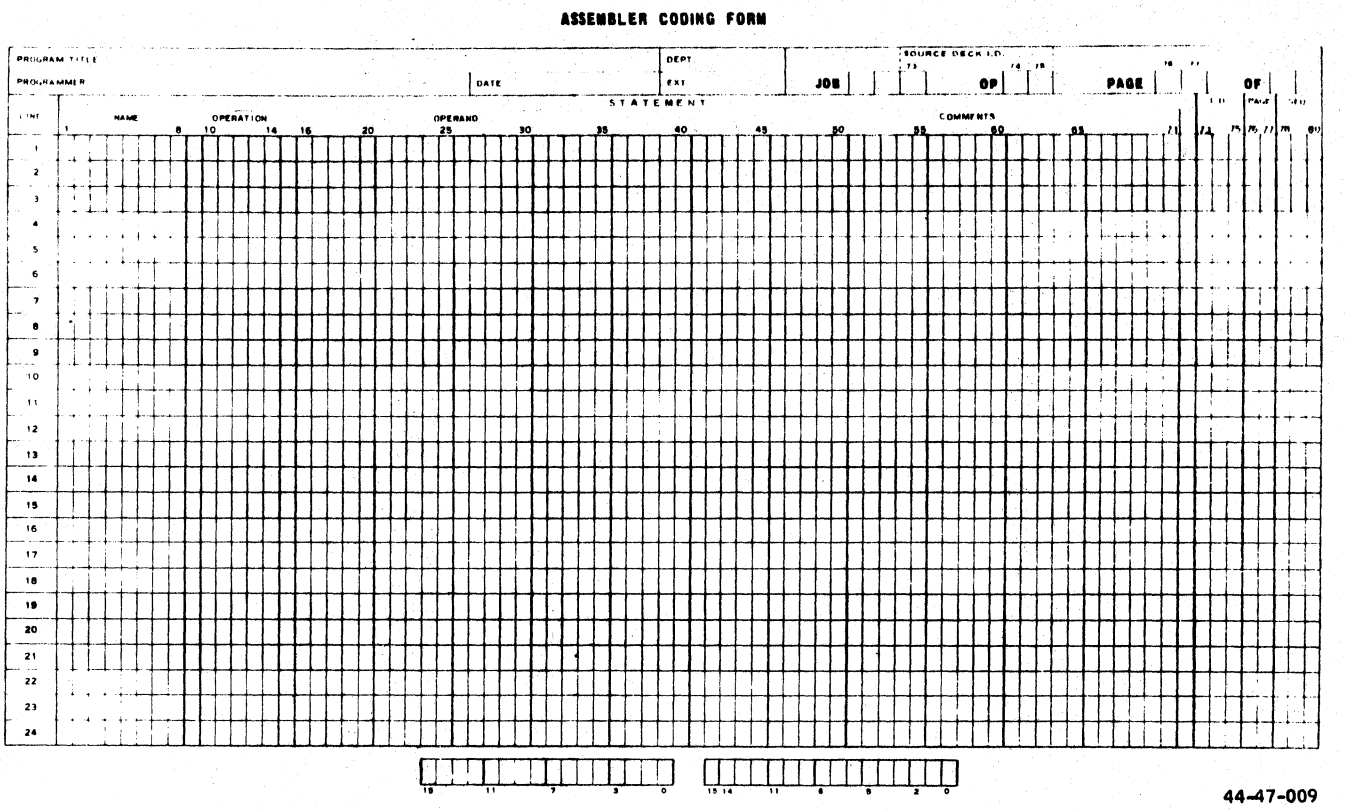


Figure 2-2. AN/GYK-12 Assembler Coding Form

b. Addresses of data and instructions can be written in symbolic form, and in practice almost all addresses are so written. The programmer is thereby relieved of the task of allocating storage. The use of symbolic addresses reduces the clerical aspects of programming, and the usual resultant errors, and makes the program easier to modify. If the symbols are chosen to be meaningful, the program is also much easier to read and understand than if written with numeric addresses.

c. A macro instruction feature is provided in the assembler language. These instructions are used to simplify the coding of a program, to reduce the change of programming errors, and to provide a standard sequence of instructions for accomplishing a specific task. Whenever a specific sequence of code is needed, the programmer inserts the macro instruction which corresponds to the sequence of code he desires. This macro instruction is recognized by the assembler and a sequence of instructions is generated to represent the macro instruction. The generated instructions are then placed into the code instead of the macro instruction, and are processed like other instructions. Both machine instructions and assembler instructions may be used in a macro definition.

d. Data may be introduced into the program structure, and space reserved for results, by the use of suitable assembler instructions. The communication pool (Compool) feature is an additional data handling feature of the AN/GYK-12 assembler.

e. Many other assembler instructions direct the assembler in the performance of its functions.

Section II. AN/GYK-12 ASSEMBLER

2-4. AN/GYK-12 Assembler Basic Functions

The AN/GYK-12 assembler is a two pass system encompassing various routines, including a control routine, a macro assembler, and an operand interpreter. The AN/GYK-12 assembler program translates symbolic instructions into machine-language instructions, assigns storage locations, performs tasks initiated by the programmer in the form of machine instructions, makes use of an externally supplied data base description (Compool), generates and resolves macro instructions, provides an assembly listing, and performs various auxiliary functions necessary to produce an efficient, executable machine language program for the AN/GYK-12 computer. The assembly process is accomplished in two passes. The first pass control routines along with the operand interpreter, and the macro assembler, produce a communication file. The operand interpreter is called by pass one to convert and analyze the variable field of certain assembler instructions. The macro assembler is also called by pass one to define and expand macro instructions. The communication file contains the results of the first pass analysis and the original input source language statements in card image format. The second pass uses this communication file as input. The second pass consists of three major processing routines, and a number of input/output routines. The second pass outputs a relocatable binary object deck and a program listing.

a. Pass One Functions. The functions performed by pass one of the AN/GYK-12 assembler are outlined as follows:

(1) Accepts card images of programs in AN/GYK-12 assembler language format.

(2) Verifies and converts operation codes.

(3) Constructs the symbol table.

(4) Evaluates the operands of certain machine instructions. Evaluates portions of the assembly instructions.

(5) Recognizes macro instructions and has macro assembler process macro-instruction definitions or expansions.

(6) Classifies each instruction into one of a number of classes for action by pass two.

(7) Writes the developed information onto a communication device (tape or disk) along with the data from the source code input.

b. Pass Two Functions. The functions performed by pass two of the AN/GYK-12 assembler are outlined as follows:

(1) Accepts the communication file data as input.

(2) Evaluates the several classes of instructions.

(3) Evaluates operands of the assembler instructions and machine instructions not evaluated by the first pass.

(4) Builds the binary object deck, and the ENTRY and EXTERN tables.

(5) Outputs the object program.

(6) Produces a program assembly listing including source and object code listings and error diagnostics, as well as a symbol dictionary, and cross reference listing.

2-5. Features of the AN/GYK-12 Assembler

The AN/GYK-12 assembler has a number of features to aid the programmer in the performance of his tasks.

2-6. Use of Mnemonic Operation Codes

The assembler utilizes mnemonic operation codes to provide auxiliary functions that assist the programmer in checking and documenting programs, controlling address assignment, defining data and symbols, generating macro instructions, and controlling the assembly process itself.

2-7. Variety in Data Representation

Decimal, fixed point decimal, hexadecimal, octal, binary, EBCDIC character, or ASCII character representation of machine language binary values may be employed by the programmer in writing source statements. The programmer may select representations best suited to his purpose.

2-8. Macro Capability

The assembler provides an extensive macro capability for the convenience of the programmer. This capability enables the programmer to create, define, and use a macro instruction, wherein a mnemonic symbol, supplied by the programmer, becomes the operation code of the instruction. There are two types of macro instructions: system macro instructions, which provide interface between the object program and the operating system; and programmer-created macro instructions for use in the program at hand, or for incorporation into the macro library for later use. Macro instructions are used to simplify the coding of a program and to provide a standard sequence of instructions for accomplishing a specific task. Whenever the sequence of code is needed, the programmer inserts the macro instruction in the desired place. The assembler program then inserts the sequence of code represented by the macro instruction following the macro instruction mnemonic in the source program.

2-9. Compool Data Definition Capability

An external data definition capability known as communication pool (Compool) is provided. This feature allows the programmer to define symbolic items for use by different program modules. Names of tables, subroutines, and programs may also be included in the Compool. Two types of Compool are provided; the master Compool, and the subset Compool. Both must be generated by a separate program called the Compool Generator and both provide extensive data definition capability.

2-10. Relocatability

The object programs produced by the assembler can be in a format which enables relocation from the originally assigned storage area to any other suitable area.

2-11. Partitioning and Linking

The AN/GYK-12 assembler allows for partitioning programs into parts called modules. Modules may be added or deleted when loading the object program. Because modules do not have to be loaded contiguously in core, a modularized program may be loaded and executed even though there is not a continuous block of core available that is large enough to accommodate the entire program. The assembler allows symbols to be defined in one program and referred to in another, thus effecting a link between separately assembled programs. This permits references to data and transfer of control between programs.

2-12. Program Listings

A listing of the statements of the source program and the resulting object program may be produced by the assembler for each program that it assembles. The programmer can partially control the content and format of the assembly listing.

2-13. Error Indicators

As a source program is assembled, it is analyzed for actual and potential errors in the programmer's use of the assembly language. Detected errors and irregularities which are potential errors are indicated in the program listing.

2-14. Cross Reference and Set/Used Listings

The assembler produces a cross reference and set/used listing incorporating all programmer and Compool defined symbols.

CHAPTER 3
BASIC COMPONENTS OF THE AN/GYK-12 ASSEMBLER LANGUAGE

Section I. SOURCE

3-1. Introduction

This chapter discusses the basic components of the AN/GYK-12 assembler language. The structure of the assembler input deck is discussed. The different types of assembler language source instructions are introduced. The format of the machine instructions is presented and each field of the general format is discussed. In addition, the various types of symbols, terms, and expressions are discussed.

3-2. Assembler Language Source Input Format

The assembler language source input format is illustrated in figure 3-1. If it is specified, the first card of any assembly deck is the option (OPT) card. Following the OPT card the TITLE card may appear. Then the Compool (CMP) card and then the origin (ORG) or location (LOC) card should be the cards appearing just before the source instructions. Additional origin (ORG) and location (LOC) cards may appear within the source statements themselves as required. The appearance of such a card will cause all of the code following to be made relocatable or non-relocatable and to start at the location specified. A summary of AN/GYK-12 assembler instructions is illustrated in Chapter 7. These four cards form the introduction to the assembler language program. The OPT card instructs the assembler as to what type of input/output the assembler itself should provide, and what types of outputs are desired by the programmer. The TITLE card identifies the assembly module. The ORG or LOC cards specify where the assembler is to begin assembly of the source program (i.e., at what location should the object code begin), and whether or not the object module will be relocatable from its assembled area at load time. The ORG card identifies the module as reloctable and the LOC card identifies the module as not relocatable. The Compool (CMP) card identifies the Compool to be used with the assembly. Following these four cards; the OPT, TITLE, CMP, and ORG or LOC, are the source language instructions of the program itself. At the conclusion of the program is the END card which identifies the end of the assembly module and the entry point of the module at load time.

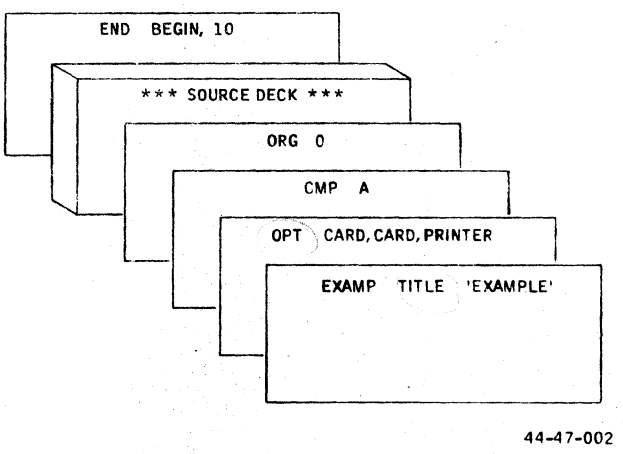


Figure 3-1. Assembler Language Source Input Format

3-3. Assembler Language Source Statements

There are four basic types of assembler language source instructions; machine instructions, assembler instructions (pseudo operations), macro reference, and macro definition instructions. The type of instruction is determined by the three to five letter mnemonic operation code which is part of the instruction. Machine instructions directly affect the object program or the actions of the assembler itself.

a. Machine Instructions. Machine instructions are those instructions for which there is a direct equivalent in AN/GYK-12 machine language. All machine instruction mnemonic codes are made up of from three to five letters. Machine instructions have five fields: name, operation, operand, comments, and identification/sequence. The construction of machine instructions corresponds to the

general assembler source input format described in Section II.

b. Macro Definition Instructions. Each macro definition is a compound instruction consisting of a macro definition header followed by a macro prototype that can be followed by any number of macro model instructions. These macro model instructions may be any AN/GYK-12 assembler language instructions except another macro definition. The macro definition instruction is concluded with a macro definition trailer instruction. The macro model instructions have a format which is identical to the general assembler source input format described in Section II. The macro prototype instruction has a format identical to the macro reference instruction which will correspond to the macro definition instruction in which it appears. Both the macro definition header and the macro definition trailer instructions have special formats.

Section II. GENERAL ASSEMBLER SOURCE INPUT FORMAT

3-4. General

The AN/GYK-12 assembler language source instructions are input to the assembler from punched cards or punched card images on magnetic tape, disk, paper tape or other suitable input device. The format of each machine instruction consists of an 80 character record divided into five fields as shown in table 3-1.

Table 3-1. General Assembler Source Input Format

Field	Maximum length	Starting column
Name	8	1
Operation	5	–
Operand	75	–
Comments		–
ID-Seq.	8	73

20079-1

a. The format is generally variable field and free format. The only format requirements are:

(1) The name field must begin in column 1. If column 1 contains a blank the assembler assumes that a name has not been entered.

(2) The identification/sequence field must begin in column 73 and end in column 80.

(3) At least one blank must separate each of the fields.

(4) Each record must not exceed 80 characters.

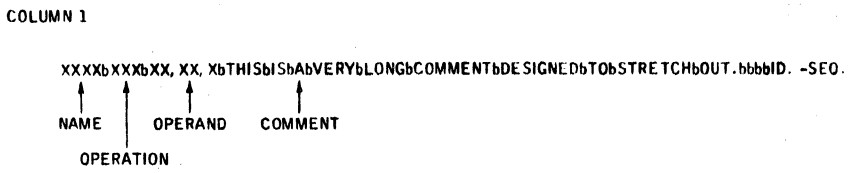
An example of an assembler input is shown in figure 3-2.

b. Name Entry. The name entry contains a symbol created by the programmer to identify or flag an instruction. Any legal symbol may be used as a name. This entry is optional, but if present must consist of not more than eight alphanumeric characters and must contain at least one letter. The first character must be in column 1, and may be a letter or a number. No blanks or special characters may appear within the symbol. If column 1 contains a blank, the assembler assumes that a name has not been entered.

c. Operation Entry. The operation entry contains the mnemonic operation code specifying the machine operation or assembler function desired. The operation entry may begin in any column after column 1, if there is a blank in column 1, or following the first blank after the name entry, if column 1 is not a blank. Valid operation codes consist of groups of alphabetic characters of from three to five letters which are recognizable by the assembler. Invalid operation codes will produce an error diagnostic.

d. Operand Entry. The contents of the operand entry describe data to be acted upon by the instruction. The operand specifies such things as values, storage locations, addressing mode, index register, and accumulator. Depending upon the instruction, one or more operands may be required. Operands are separated by commas with no intervening blanks. The first blank terminates the operand.

e. Comments Entry. The comments entry contains descriptive information about program statements. All valid characters acceptable to the computer, including blanks, may be used in writing the comments entry. This entry cannot extend



44-47-003

Figure 3-2. Example of an Assembler Input Record

beyond column 72 and must be separated from the operand entry at least one blank. If there is no operand entry the comments entry must be separated from the operation entry by a blank followed by a comma, followed by another blank. The comments entry will not cause the assembler to generate or modify any executable code resulting from the operation or operand entries of the instruction, in which they appear. If desired entire cards may be used as comments (refer to the REM, Remark Assembly Instruction, Chapter 7).

f. Identification/Sequence Entry. The identification/sequence entry is used to enter program identification and/or instruction sequence characters. Any characters acceptable to the hardware may be used in the identification/sequence entry. The characters in this entry will be checked by the assembler for ascending order according to the standard collating sequence for the computer. An identification/sequence entry with a value less than or equal to that of the preceding entry will be noted on the assembly listing with an asterisk. The identification/sequence entry must begin in column 73 and end in column 80. This entry need not be separated from the operand or comments field by a blank (i.e., column 72 need not be blank).

3-5. Symbols

A symbol is a character or combination of characters used to represent an address or an arbirary value. Symbols, through their use in the name and operand entries, provide the programmer with an efficient way to catagorize and reference a program element.

a. Defining Symbols. The rules which must be followed in defining symbols are as follows:

(1) The symbol must not consist of more than eight characters, at least one of which must be a letter.

(2) Only alphanumeric (letter and number) characters may be used. Special characters are not allowed.

(3) Blanks are not allowed in a symbol.

(4) If the symbol is used in the name entry the first character of the symbol must be in the first position of the name entry area (column 1 of the instruction or statement).

(5) The first character of the symbol may be a number or a letter.

The following are valid symbols:

READER
23A46
X4F2
LOOP2
R6
N
01234
L
A100

The following are invalid symbols, for the reasons noted:

256	(no letter)
RECORDAREA	(more than eight characters)
BCD*34	(contains a special character, *)
IN AREA	(contains a blank)

b. Assignment of Symbols. The assembler assigns a value to each symbol appearing as a name entry in a source instruction. The value assigned represents the address of a storage area, instruction, or constant. Since the addresses of these items may change upon program relocation, the symbols are considered reloctable terms. An exception to this rule is symbols which are within

nonrelocatable portions or programs. Symbols used in the name entry of equate (EQU) statements are assigned the value designated in the operand entry of the instruction. Since this value may be relocatable or absolute, depending upon the nature of the operand, the symbol is considered the same as the value to which it is equated.

c. Previously Defined Symbols. Some instructions require that a symbol appearing in the operand entry be previously defined. This means that the symbol, before it is used in the operand entry, must have appeared as a name entry in a previous instruction, or as an identifier in the Compool.

```
EXAMPLE: DELTA   EQU   R'8'
                 .     .
                 .     .
                 .     .
                 LDF   =X'FFF',DELTA
```

Normally, the second subfield of an instruction contains a number from 0 to 15 which connotes a Process Register number. The above example illustrates a method in which a symbol can be used in place of a Process Register number. In such a case, for the assembler to know that DELTA means Process Register 8, DELTA must be defined prior to its use.

d. General Restrictions on Symbols. A symbol may be defined only once in an assembly. If any symbol is defined more than once in an assembly the assembler will generate an M error (multiple defined symbol) for the instruction in which the subsequent duplicate definition appears. If more than one instruction contains a duplicate name then the error will appear for each such instruction except the first.

3-6. Terms

A term is the basic unit of the operand entry. All terms represent a value. This value may be assigned by the assembler (symbols, or location counter reference) or may be inherent in the term itself. An arithmetic combination of terms is reduced to a single value by the assembler. All terms are unsigned and no blanks are allowed within the term. A term with a sign is considered an expression. Terms are either absolute, relocatable, or special.

a. Absolute Terms. An absolute term represents a self defining value, analogous to a constant in mathematics. The value of an absolute term is inherent in the term itself and is not assigned by the assembler. There are eight types of absolute terms: decimal, hexadecimal, octal, binary, EBCDIC character, ASCII character, process register, and certain symbolic terms. Since the values represented by absolute terms do not change upon program relocation, absolute terms are considered as self defining terms, i.e., their values are defined by the context of the term itself and only once (at assembly time).

(1) *Decimal absolute terms.* A decimal absolute term is an unsigned decimal number written as a sequence of decimal digits with a decimal point. The decimal point must be placed within or before the number. An integer does not require a decimal point. A scale factor must follow a mixed number or fraction otherwise the fractional portion of the number is lost. In addition a scaling factor and/or a decimal exponent may be used. The decimal number can be an integer, a fraction, or a mixed number (i.e., one with integer and fractional portions). The decimal term is the only term which may use a scaling factor or a decimal exponent. High order ZEROS may be used or omitted for numbers greater than 1 (e.g., 00193 or 193 and 001.67 or 1.67).

(a) A negative decimal term is treated as an expression consisting of a minus sign followed by a decimal constant. At assembly time the decimal term is converted into a binary number of a length sufficient to contain the converted binary number. For example, 37 would be represented as 100101, which is six bits in length. Depending upon the use of the term, the converted binary number must not exceed 31 bits (full word) or 63 bits (double word).

(b) To code a decimal integer all that is needed is the value without any special notation. For fractions or mixed numbers a scale modifier (the letter B) must be used to tell the assembler how many frcational bits to reserve for the value. The reason for this is that integer values are always right justified within a computer word.

EXAMPLE: 15 converts to 1111_2
15.25 converts to 1111_2

In the above example both numbers converted to the same binary values. The assembler was not told to reserve fractional bits so the fractional part of the value was lost (truncated).

EXAMPLE: 15 converts to 1111_2
15.25B2 converts to 111101_2

In the above example a scale modifier is used to correctly position both the integer and the fractional part of the mixed number and the result in binary is different from the result obtained when the scale modifier was not used. To correctly use the scale modifier the letter B immediately follows the fractional part with no intervening blank spaces. Following the letter B is a decimal unsigned number which indicates the number of fractional bits to be reserved. In the second example, two fractional bits were reserved (.25 translates to 01 binary). The entire translation in binary is 1111 for the integer 15 followed by 01 for the fraction. Leading ZEROS would be appended by the assembler (positive value used in the examples). It is also possible to raise or lower a decimal value by powers of 10. This necessitates the use of the term E (for exponent) followed by an optionally signed decimal number.

EXAMPLE: 15 converts to 1111_2
1.5E1 converts to 1111_2
150E-1 converts to 1111_2

In the illustrated example the value 1.5 is followed by E1 which is 10^1 or 10. It could also have been coded E+1. Multiplying 1.5 by 10 results in the value 15. The number following the term E can produce any power of 10 (E2 is 10^2, E3 is 10^3, etc.). For negative powers of 10 a minus sign must precede the decimal value (E-1 is 10^{-1}, E-2 is 10^{-2}, etc.). In the illustrated example the number 150 was scaled down by using a negative power of 10 to obtain the value 15. Both the scale factor and the decimal exponent can appear together in the same statement. When this occurs, the decimal exponent must appear first.

EXAMPLE: 15.25B2 converts to 111101_2
1.525E1B2 converts to 111101_2

In the above example the 1.525 is multiplied by 10^1 to produce 15.25. This is then translated to binary with the sacle modifier speciying two fractional bit positions. Any fractional value which requires more than the number of significant bits designated by the scale modifier to fit into the designated binary bit positions of the computer word will be truncated on the right (least significant bits) to fit. Any integer value which is too large to fit into the designated binary bit positions of the computer word will be truncated on the left (most significant bits) to fit. For mixed numbers, if both the integer and the fractional values are too large to fit into the designated binary bit positions of the computer word truncation will occur at both ends.

(2) *Hexadecimal absolute terms.* A hexadecimal absolute term is written as an unsigned sequence of hexadecimal digits. The digits must be enclosed in single quotation marks preceded by the letter X, for example, X'3A7' or X'AB7'. Each hexadecimal digit is assembled as its four bit binary equivalent. Thus, a hexadecimal term used to represent an eight bit binary number would consist of two hexadecimal digits. Limitations on the value of a hexadecimal term depend upon its use. The hexadecimal digits and their corresponding binary values are as shown in the following chart.

HEXADECIMAL	BINARY VALUE
0	0000
1	0001
2	0010
3	0011
4	0100
5	0101
6	0110
7	0111
8	1000
9	1001
A	1010
B	1011
C	1100
D	1101
E	1110
F	1111

A negative hexadecimal number is considered an expression consisting of a minus sign followed by an unsigned hexadecimal number. Scale factors and exponents may not be used in hexadecimal terms.

(3) *Octal absolute terms.* An octal absolute term is written as an unsigned sequence of octal digits. The digits must be enclosed in single quotation marks preceded by the letter 0, for example '743' or '746' (The letter is slashed when used with numbers to distinguish it from the number ZERO.) Each octal digit is assembled as its three bit binary equivalent. Thus, an octal term used to represent a six bit binary number would consist of two octal digits. Limitations on the value of an octal term depend upon its use. The octal digits and their corresponding binary values are as follows:

OCTAL	BINARY VALUE
0	000
1	001
2	010
3	011
4	100
5	101
6	110
7	111

A negative octal number is considered an expression consisting of a minus sign followed by an unsigned octal number. Scale factors and exponents may not be used in octal terms.

(4) *Binary absolute terms.* A binary absolute term is written as an unsigned binary number consisting of a sequence of binary digits. The digits must be enclosed in single quotation marks and preceded by the letter B, for example, B'010111'. Each binary digit is assembled directly. Thus, a binary term used to represent a six bit mask would consist of six binary digits. Limitations on the value of the binary term depend upon its use.

(5) *EBCDIC absolute terms.* An EBCDIC absolute term is a sequence of Extended Binary Coded Decimal Interchange Code characters. The characters must be enclosed in single quotation marks and preceded by the letter C, for example, C'TEST(7)'. All letters, decimal digits, and special characters may be used in a character term. Because of the use of single quote marks in the assembler language, two single quote marks must be used to represent a single quote mark or an apostrophe within the character field itself. Examples of EBCDIC absolute terms are: C'/', C'ABC', C' ', C'13', C'''A'''. Each character is assembled as its eight bit EBCDIC code equivalent. A chart of the EBCDIC character codes is provided in Appendix B.

(6) *ASCII absolute terms.* An ASCII absolute term is a sequence of American Standard Code for Information Interchange characters. The characters must be enclosed in single quotation marks and preceded by the letter A, for example, A'ATEST(7)'. All letters, decimal digits, and special characters may be used in a character term. Because of the use of quote marks in the assembler language, two quote marks must be used to represent a quote mark within the character field itself. Examples of ASCII absolute terms are: A'/', A'/ABC', A' ', A'13', A'''A'''. Each character is assembled as its eight bit ASCII code equivalent. A chart of the ASCII character codes is provided in Appendix C.

(7) *Process register absolute terms.* A process register absolute term is used to reference or indicate a process register, and is written as an unsigned decimal number. The number designation must be enclosed in single quotation marks and preceded by the letter R, for example R'15'. A process register term is assembled as its three or four bit binary equivalent and is placed in the appropriate register containing field indicated by its placement in the operand field. If a process register term appears as the first entry in the operand field it is assembled as the binary equivalent of its special register address in the A field. The hexadecimal addresses generated are as shown in the following chart.

REGISTER	HEXADECIMAL ADDRESS GENERATED
0	0
1	2
2	4
3	6
4	8
5	A
6	C
7	E
8	10
9	12
10	14
11	16
12	18
13	1A
14	1C
15	1E

Process register terms may be used to specify accumulators, index registers, tally registers, and special registers (as previously shown). Process register terms always designate the set of registers for the currently active program level. Limitations on the value of a process register term depend upon its use.

(8) *Symbolic absolute term.* A symbolic absolute term is used to specify a non-relocatable address. Any symbols defined within an area governed by a LOC (fixed location counter) card will be non-relocatable. Symbols used in the name field of the equate instruction (EQU) are assigned the value designated in the operand field

of the instruction. A symbolic absolute term is one whose equated value is absolute.

EXAMPLE:

```
        LOC   X'5000'
ABA     GEN   F0
ABB     GEN   FX'FFFFFFFF'
```

In the above example, ABA and ABB are symbolic absolute terms (addresses). They are non-relocatable because they are assembled following a LOC assembler instruction.

b. Relocatable Terms. A relocatable term is a symbolic term whose value is determined by the assembler. The value of a relocatable term is the address of a 16 bit half-word in memory. This address is subject to alteration when the program in which the term appears is loaded for execution. The address will change when the program is relocated. However, portions of a program may be designated relocatable or not relocatable, and terms appearing within a non-relocatable portion of a program will not be relocated. Symbols used in the name field of equate instructions (EQU) are assigned the value designated in the operand field of the instruction. A relocatable symbolic term is one whose equated value is relocatable.

c. Special Designators. There are a number of special designators in the AN/GYK-12 assembly language. The equal sign, =, is used to designate the literal mode of addressing (mode 0). The left and right parentheses, ( ), are used to designate the indirect mode of addressing (mode 3). The dollar ($) symbol is used to designate the relative mode of addressing (mode 2), and also represents the instruction location register. The letter R is used to indicate an index register, and the letter D is used to indicate double indexing with the register specified. The asterisk (*) symbol is used as a means of referencing the location counter without affecting the addressing mode.

(1) *Literal addressing designator* (=). When the equal sign, =, appears as the first character in subfield one of an instruction the literal mode of addressing (mode 0) is indicated.

(2) *Indirect addressing designator* (( ). When the left parenthesis, (, appears as the first character in subfield one of an instruction the indirect mode of addressing (mode 3) is indicated. The address is enclosed in parentheses.

EXAMPLE: (PC4A)

(3) *Relative addressing designator* ($). The programmer may refer to the current value of the instruction location register (ILR) and thereby designate the relative mode of addressing by using a $ symbol in the operand field of an instruction. The relative mode is used when the programmer wishes to refer to a location relative to the instruction from which the reference is being made. The use of the $ symbol will designate the relative mode of addressing which is effectively the same as inserting the current value of the instruction location register (the address of the current instruction plus two) into the operand field in place of the $ symbol.

(4) *Location counter designator* (*). A Location Counter is a software counter used to assign addresses to instructions during program assembly as distinguished from the hardware instruction location counter (ILR) which normally contains the instruction location count plus two. The programmer may refer to the current value of the location counter without affecting the mode at any place in the program by using an asterisk (*) symbol as a term in the operand field. The asterisk (*) symbol represents the address of the instruction in which it appears. The asterisk (*) symbol has no effect on the addressing mode and does not indicate relative mode addressing.

(5) *Process register indexing designator (R).* If the process register absolute term is used following an arithmetic operator in the operand field, the indexing mode of the specified addressing mode will be designated.

(6) *Double indexing designator (D).* The double indexing designator D is used only with the direct mode of addressing and is used to designate the direct with double indexing mode of addressing. The use of the double indexing designator D implies the use of process register 1 to contain one of the index values. The other index value is designated by the number following the D in the instruction. The double indexing designator is used in place of the R for a normal indexing mode designator, for example, D'2'. A double indexing designator may only specify index registers 2 through 7.

3-7. Expressions

An expression is an operand entry consisting of either a single term or an arithmetic combination of terms. Up to ten terms can be combined with the following arithmetic operators:

OPERATION	EXAMPLE
+ Addition	ALPHA+BETA
- Subtraction	ALPHA-BETA
* Multiplication	ALPHA*BETA
/ Division	ALPHA/BETA

An expression may not contain two terms or two operators in succession.

a. Evaluation of Expressions. A single term expression, e.g. BETA, S, X5G, takes on the value of the term involved. A multiterm expression, e.g. BETA+10, ENTRY-EXIT, 10+A/B, is reduced to a single value, as follows:

(1) Each term is given its value.

(2) Arithmetic operations are performed left to right. Multiplication and division are performed before addition and subtraction, e.g. A+B*C is evaluated as A+(B*C), not (A+B)*C. The computed result is the value of the expressions.

(3) Division yields an integer result; any fractional portion of the result will be dropped. For example, the expression 1/2*10 equals zero but the expression 10*1/2 equals 5.

b. Absolute Expressions. An absolute expression is a combination of from one to ten terms which may consist of the following:

(1) All absolute terms.

(2) An even number of relocatable terms with opposite signs.

(3) An even number of relocatable terms with opposite signs and any number of absolute terms. An even number of relocatable terms with opposite signs cancels the effect of relocation. For example, in the absolute expression A-B+4, A and B are relocatable terms and 4 is an absolute term. If A equals 50 and B equals 40, the values after relocation by +20 would be 70-60+4=14. An expression is called absolute if its value is unaffected by program relocation. Extreme caution should be used when generating relocatable or absolute expressions with more than one symbolic term.

c. Relocatable Expressions. A relocatable expression is one whose value would change by n if the program in which it appears is relocated n 16 bit words away from its originally assigned area of storage. A relocatable expression may contain relocatable terms, alone or in combination with absolute terms under the following conditions:

(1) There must be an odd number of relocatable terms.

(2) If a relocatable expression contains an odd number of relocatable terms, the extra odd term must not be preceded by a minus (-) sign.

(3) No relocatable term can enter into a multiply or divide operation. An expression is called relocatable if its value at execution time depends upon program relocation. A relocatable expression reduces to a single relocatable value. This value is the value of the odd relocatable term, adjusted by the values represented by the absolute terms and/or paired relocatable terms. For example, in the expression A-B+A, A and B are relocatable terms. If initially A equals 40 and B equals 30, the value of the expression is 50. However, if a relocation factor of 20 is applied, the value of the expression is 70. The value of the pair of terms A-B remains constant, and the value of the expression increases by the relocation factor as a result of its application to the positive odd relocatable term. Note that the intent is to relocate the entire expression by the proper amount, not just one or some of the terms. For example, if the expression A+B-C+D-E is to be relocated by 20, it is the value of the final evaluated expression which should be greater by 20 when the relocation factor of 20 is applied to each of the relocatable terms in the expression. Thus, relocation of expression becomes a question of mathematics and not rules.

CHAPTER 4
AN/GYK-12 MACHINE INSTRUCTIONS

4-1. Introduction

This chapter discusses the format and construction of the entries which make up AN/GYK-12 assembler machine language instructions. Brief comments on the name entry, operation entry, comments entry, and identification/sequence entry are presented. A detailed discussion of the operand entry format and the use of Compool symbols in the assembler language machine instructions is presented. The construction of the operand entries for the standard assembler language machine instructions and the extended mnemonic codes is presented. A discussion of the addressing modes is also inccluded. A detailed discussion of the extended mnemonic code instructions is presented in Chapter 5. Appendix D contains a list of the machine instructions acceptable to the assembler. Detail explanations of the operation of the machine instructions may be found in the USACSCS-TF-4-3 AN/GYK-12 Principles of Operation Manual.

4-2. Machine Instruction Format

The assembler language machine instruction format follows the general assembler source input format described in Chapter 3. Figure 4-1 illustrates the construction of a machine instruction. Figure 4-2 shows how a sample assembler language machine instruction is assembled.

Figure 4-1. Construction of Assembler Language Machine Instruction

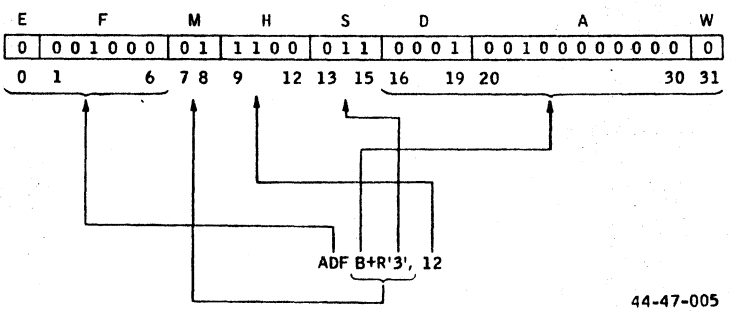


Figure 4-2. Assembler Language Machine Instruction Conversion

a. Name Entry Format. The name entry contains a symbol to be used as a name. The name need not be referenced at any place in the program and is optional.

b. Function Code Entry Format (E and F fields). The function code entry must have a three letter mnemonic operation code, or a three to five letter extended mnemonic operation code.

c. Operand Entry Format. The operand entry specifies the core memory address of data to be accessed, the process register to be used as an accumulator, the process register to be used as an index register, the process register to be used as a tally register, the bit of the field to be considered (in bit manipulation instructions), or the location to which control is to be transferred (for transfer instructions). The extended mnemonic code instructions have a different operand entry format from the assembler language machine instructions and this format is discussed in Chapter 5. The machine instruction operand entry has two subfields. Each subfield is separated by a comma and the first blank terminates the operand entry. If the first subfield were not used then it would be replaced by a comma and the second subfield

would appear directly following the comma with no intervening blank(s). Such an instruction would appear as:

OPERATION	OPERAND
ADF	,2

All unused subfields are considered to be ZERO.

(1) *Operand subfield one.* The mode, index, page, word address, and half word address fields of the computer instruction word are provided by this subfield.

(a) Addressing special symbols. Certain special symbols may be used in subfield one for specific purposes. These are discussed in detail in Chapter 3. An explanation is also provided for reference in table 4-1. If none of the special symbols is used in subfield one then the direct mode of addressing (mode 1 or 2) is indicated.

Table 4-1. Addressing Special Symbols

Special character	Meaning
=	An equal sign as the first character in the subfield indicates the literal mode of addressing (mode 0).
$	The dollar sign symbol indicates the relative mode of addressing (mode 2).
( )	If the first subfield is enclosed in parentheses, the indirect mode of addressing (mode 3) is indicated.
NOTE:	The use of these addressing modes with the transfer and extended mnemonic instructions requires additional information. Only the literal mode (mode 0) may be used with the extended mnemonic instruction. Refer to the USACSCS-TF-4-3 AN/GYK-12 Principles of Operation Manual for information regarding the transfer instructions.

20079-2

The special symbols are as follows:

(b) The literal mode (mode 0) is indicated by an equal sign (=) appearing as the first character of the operand field.

(c) The relative mode (mode 2) is indicated by a dollar sign ($) appearing anywhere within subfield one. The dollar sign also denotes the value of the instruction location register (ILR).

(d) The direct mode with double indexing (mode 2) is indicated by a double indexing term appearing anywhere within subfield one of the operand.

(e) The indirect mode (mode 3) is indicated by a left parenthesis (( ) appearing as the first character in the operand field. The mode is contained in the M field of the machine instruction.

Table 4-2 lists the nine possible addressing mode combinations with their corresponding symbolic representations in subfield one.

Examples showing the use of each of the nine possible addressing mode combinations are shown in table 4-3.

NOTE

All the instructions in table 4-3 take place as one smooth operation. There are no partial results or distinct steps requiring auxiliary storage, concerning the programmer. In these operations, any partial results or distinct steps indicated in the explanations in the examples are used for the purpose of illustration only. Examples of the use of the special symbols are shown in table 4-4.

(f) Indexing. Indexing is indicated by the appearance of an index register term in subfield one. An index register term is indicated by the special character R followed by the process register number (1-7) enclosed in single quotation marks. If the indexing term is immediately preceded by any other operation indicator but the plus sign (+), the expression will not be correctly evaluated and no error symbol will be placed by the assembler. (The value of the index register will always be added when the addressing mode with indexing is specified.) The S field of the instruction word will be loaded with the indexing register specified and normal indexing execution will take place. If the index register specified is not within the limits 1-7, the assembler will generate an L (limit) error symbol. The indexing register term may appear anywhere within subfield one except as the first term. If mode 2 (relative) is indicated with the presence of the $ symbol in subfield one and if the S field is ZERO, no indexing will be used. The only acceptable indexing value for the relative mode is register one. Any index register value other than one is

illegal. If the direct mode with double indexing (no special symbols in the operand and the D, double indexing term used), then mode 2 will also be selected but the only acceptable registers for double indexing are registers 2-7. Double indexing may not be used with any mode other than direct addressing (mode 2).

(2) *Operand subfield two.* The accumulator (H field) of the machine instruction is specified by this subfield for all instructions, except move into upper byte (MIU) and move into lower byte (MIL), which use two operand subfields. Any term or expression may appear in this subfield with the exception of the double indexing term (D'N'). The limitation on the value of this subfield is 15 for all instructions except MIU and MIL for which the limitation is 255.

*Table 4-2. Addressing Mode Combinations (Transfer Instructions Excluded)**

Addressing mode			Symbolic representation (in ranges from 1 through 7)
M	S		
0	0	literal	= Y
0	1-7	literal with indexing	= Y+R'n'
1	0	direct	Y
1	1-7	direct with indexing	Y+R'n'
2	0	relative	Y -$ or Y+$
2	1	relative with indexing	Y-$+R'1' or Y+$+R'1'
2	2-7	direct with double indexing	Y+D'm'(m=2-7)
3	0	indirect	(Y)
3	1-7	indirect with indexing	(Y+R'n') or (Y)+R'n' (See note 1)

Note: See Principles of Operation Manual USACSCS-TF-4-3

20079-3

Table 4-3. Addressing Mode Combination Examples

Mode with example	Assumptions	Operation
LITERAL (MODE=0, S=0) ADF = Y, 10 Note: If a symbol is used, the address or value (EQU) of the symbol will be placed into the A field of the instruction by the assembler. If the symbol is undefined, an error will be generated.	1) The operand in the DAW field of the instruction contains 200. 2) Register 10 contains 300.	Add the contents of the Y field of the instruction, which is 200, to the contents of register 10, which is 300. Store the result, 500, into register 10. The previous contents of register 10 have now been replaced.
LITERAL WITH INDEXING (MODE=0, S=1-7) ADF = 150+R' 3', 11 Note: If a symbol is used, the address or value (EQU) of the symbol will be placed into the A field of the instruction by the assembler. If the symbol is undefined, an error will be generated.	1) The operand in the DAW field of the instruction contains 150. 2) Register 3 contains 50. 3) Register 11 contains 300.	Add the contents of the Y field of the instruction, 150, to the contents of register 3, 50 and obtain the partial result 200. Now add 200 to the contents of register 11, 300, and store the result, 500, into register 11. The previous contents of register 11 have now been replaced.
DIRECT (MODE=1, S=0) ADF Y, 8	1) The address of the operand, Y, is location 1000_{16}. 2) The contents of location 1000_{16} is 200. 3) Register 8 contains 300.	Add the contents of location 1000_{16}, which is 200, to the contents of register 8, which is 300, and store the result, 500, into register 8. The previous contents of register 8 have now been replaced.
DIRECT WITH INDEXING (MODE=1, S=1-7) ADF Y+R'3', 9	1) The contents of the DAW field of the instruction is 1000_{16}. 2) Register 3 contains 400_{16}. 3) The contents of location 1400_{16} is 200. 4) Register 9 contains 300.	To obtain the address of the operand add the contents of the DAW field, which is 1000_{16} to the contents of register 3, which is 400_{16}, to obtain the effective address of the operand, 1400_{16}. Now do the arithmetic calculation, add the contents of the operand (200) to the contents of register 9 which is 300, and store the result, 500, in register 9. The result has now replaced the original contents of register 9.
RELATIVE (MODE=2, S=0) ADF Y+$, 5	1) The contents of the DAW field of the instruction is 100_{16}. 2) The address of the instruction is $3FE_{16}$, therefore the instruction location register is set to 400_{16}. 3) The contents of location 500_{16} is 200. 4) The contents of register 5 is 300.	Obtain the address of the operand by adding the contents of DAW field of the instruction, 100_{16}, to the value of the instruction location register, 400_{16}. Now add the operand, 200, at address 500_{16} to the contents of register 5, 300, and replace the contents of register 5 with 500. Note: For the XEX command the address of the location of the instruction to be executed is used.

20079-4

Table 4-3. Addressing Mode Combination Examples (Cont)

Mode with example	Assumptions	Operation
RELATIVE WITH INDEXING (MODE=2, S=1) ADF Y+$+R'1', 5	1) The contents of the DAW field of the instruction is 100_{16}. 2) The address of the instruction is $1FE_{16}$, therefore the instruction location register is set to 200_{16}. 3) The contents of register 1 is 200_{16}. 4) The contents of location 500_{16} is 200. 5) The contents of register 5 is 300.	To obtain the address of the operand, add the contents of the DAW field, 100_{16}, to the instruction location register, 200_{16}, and then add that result 300_{16} to the contents of register 1, 200_{16}. Then take the operand, the contents of 500_{16}, which is 200, and add it to the contents of register 5. The result 500 then replaces the previous contents of register 5. Note: For the XEX command the address of the location of the instruction to be executed is used.
DIRECT WITH DOUBLE INDEXING (MODE= 2, S=2-7) ADF Y+D'3'5	1) The contents of the DAW field of the instruction is 200_{16}. 2) The contents of location 302_{16} is 100_{16}. 3) The contents of register 3 is 100_{16}. 4) The contents of register 1 is 2. 5) The contents of register 5 is 300.	Obtain the address of the operand by adding the value of the DAW field which is 200_{16} to the contents of register 3, which is 100_{16}. Then add to the result the contents of register 1, which is 2, to obtain the address 302_{16}. Add contents of 302_{16} to the contents of register 5, 300. The result, 500, replaces the previous contents of register 5.
INDIRECT (MODE=3, S=0) ADF (Y),5	1) The contents of the DAW field is the address 200_{16}. 2) The contents of location 200_{16} is 1000_{16}, the address of the operand. 3) The contents of 1000_{16} is 200. 4) The contents of register 5 is 300.	Obtain the address of the operand by taking the contents of the address specified in the DAW field of the instruction as a second address (contents of 200_{16} are 1000_{16} which is the effective address). Now obtain the operand, 200, from the contents of 1000_{16} and add it to the contents of register 5. Store the result, 500, into register 5, replacing its previous contents.
INDIRECT WITH INDEXING (MODE=3, S=1-7) ADF (Y+R'3'),5	1) The contents of the DAW field of the instruction is 900_{16}. 2) The contents of register 3 is 100_{16}. 3) The contents of 1000_{16} is the address of the operand 300_{16}. 4) The contents of 300_{16} is 200. 5) The contents of register 5 is 300.	Obtain a working address by adding the contents of the DAW field, 900_{16}, to the contents of register 3, 100_{16}. Now take the contents of the sum 1000_{16}, which is another address, 300_{16}. Take the contents of that address, 200, the operand, and add it to the contents of register 5, replacing its previous contents.

20079-5

Table 4-4. Examples of the Use of Addressing Special Symbols

Subfield one construction	Meaning
=C'ER'	Specifies literal mode (M=0) with D-A-W field equal to the binary code equivalent of the EBCDIC characters E and R
=3+R'2'	Specifies literal mode (M=0) with indexing (S=2) and D-A-W field equal to the value of decimal 3
X'7F00'	Specifies direct mode (M=1) with D-A-W field equal to hexadecimal 7F00
O'30500'+R'3'	Specifies direct mode (M=1) with indexing (S=3) and D-A-W field equal to octal 30500
$+4 or 4+$	Specifies relative mode (M=2) with D-A-W field equal to decimal 4
A-$+R'1'	Specifies relative mode (M=2) with indexing (S=1) and D-A-W field equal to the value of the symbol A minus the value of the location counter plus two
R'10'+D'3'	Specifies direct mode (M=2) with double-indexing (S=3) and D-A-W field equal to hexadecimal 14
(B)	Specifies indirect mode (M=3) with the D-A-W field equal to the value of the symbol B
(C+R'6')	Specifies indirect mode (M=3) with indexing (S=6) and the D-A-W field equal to the value of the symbol C.

NOTE: Only the = and (must appear as the first character of the subfield if they are used. Any term may follow the = or the (but it is recommended that the absolute term follow the equal sign. The $, *, and D designators may appear as terms (or within terms) anywhere in subfield one.

20079-9

CHAPTER 5
AN/GYK-12 COMPUTER EXTENDED MNEMONIC INSTRUCTIONS

5-1. Introduction

This Chapter discusses the AN/GYK-12 computer extended mnemonic codes. The format and construction of these instructions is discussed. In addition, the instructions themselves are explained in detail with illustrations. A detailed discussion of the operand fields of the different types of instructions is presented.

5-2. Extended Mnemonic Code Instructions

The assembler provides extended mnemonic codes for the transfer on indicators, shift, and format instructions. These codes allow the transfer, shift, and format instructions to be specified mnemonically through the assembler as well as directly through the use of the XIN, SHF, SHH, FIF, and FEF machine instructions.

a. Extended Mnemonics for Conditional Transfer Instruction XIN. The extended mnemonic codes for conditional transfer instructions imply the transfer on indicators (XIN) instruction function code and a value for the H field of the instruction. The allowable codes and the implied H field values are shown in table 5-1.

Table 5-1. Extended Mnemonics for Conditional Transfer Instructions

Extended code	Meaning	H Field
XLS	Transfer on Less	1
XGR	Transfer on Greater	4
XEQ	Transfer on Equal	2
XNG	Transfer on Not Greater	3
XNL	Transfer on Not Less	6
XNE	Transfer on Not Equal	5
XCY	Transfer on Carry	8
XOF	Transfer on Overflow	4

20079-10

b. Extended Mnemonic Conditional Transfer Instruction Operands. The extended mnemonic conditional transfer instruction operand entry is identical to the operand entry for the standard assembler language instructions. Some examples of extended mnemonic conditional transfer instructions are shown below:

Transfer to address A if equal indicator is set

XEQ =A

Transfer to address B if greater indicator is set

XGR =B

Transfer to address C, modified by index register 4, if the carry indicator is set.

XCY =C+R'4'

c. Extended Mnemonic Code Instruction Conversion for XIN. The assembler conversion of the extended mnemonic code for the Transfer on Indicators instruction, XIN, is illustrated in figure 5-1. The assembler converts the extended mnemonic into the recognized machine language instruction, XIN, (E and F fields) and then sets the appropriate values into the M, H, S and D, A, W fields of the instruction word.

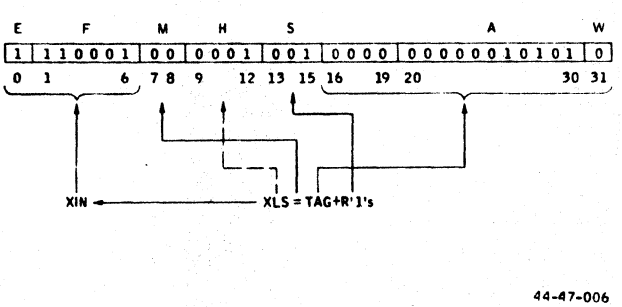


Figure 5-1. Assembler Language Extended Mnemonic Conditional Transfer Instruction Conversion

d. Extended Mnemonics for Shift Instructions. The extended mnemonic codes for shift instructions imply the shift half (SHH) or shift full (SHF) function code and a value for the T field (shift options) of the instruction. Since the T field replaces part of the normal operand (D, A, W) field, the literal mode (mode 0) must be used with the extended mnemonic codes. The extended mnemonic code assembler instructions have special operand field formats to allow the programmer to easily use the shift instructions. The notation used for the shift instructions is shown in table 5-2.

Table 5-2. Notation Used in Extended Mnemonic Shift Instruction Codes

Position*	Letter	Meaning
1	S	Shift
2	A	Algebraic
	L	Logical
	C	Circular
	N (See note 1)	Normalize
3	R	Right
	L	Left
4	F	Full word
	D	Double word
	H	Half word

Note 1: All shift mnemonics are four letters except normalize SNF, SND, SNH; shift and count, SCF, SCH; and reflect, RFT. Algebraic and logical as first specifiers are always linear. Circular as a first specifier is always logical.

20079-11

e. Extended Mnemonics for Full Word Shift Instructions. Detailed explanations of the extended mnemonic codes for the full word shift instructions are presented in the following paragraphs. Table 5-3 shows the allowable codes and the implied T field for the full word shift options of the shift full (SHF) instruction.

f. Extended Mnemonics for Double Word Shift Instructions. Detailed explanations of the extended mnemonic codes for the double word shift instructions are presented in the following paragraphs. Table 5-4 shows the allowable codes and the implied T field for the double word shift options of the shift full (SHF) instruction.

g. Extended Mnemonics for Half Word Shift Instructions. Detailed explanations of the extended mnemonic codes for the half word shift instructions are presented in the following paragraphs. Table 5-5 shows the allowable codes and the implied T field for the shift half (SHH) instruction.

h. Extended Mnemonic Shift Instruction Operands. The operand entry for extended mnemonic shift instructions is specified in the following paragraphs. Some examples of the extended mnemonic shift instructions are shown below:

Shift Right Algebraic (Linear) Half

SARH =7,4

Register = 4 Shifts = 7

Shift Circular Left Full

SCLF =17,11

Register = 11 Shifts = 17

Normalize Double Word

SND =32,7,6

Register = 7

Shifts =32 max. Tally Register = 6

i. Operand Subfields. Since the extended mnemonic codes for the shift instructions modify the rightmost half of the instruction word, the literal mode of addressing must be specified whenever these codes are used. The format of these instructions, as illustrated above, can be different than for regular machine instructions in that the standard operand format is extended, at times, to include an additional subfield. The format for the operand subfields is as follows:

(1) *Subfield one.* Specifies the mode, which must be literal (denoted by an equal sign). Also specifies the value of the K (shift) field. The shift value must be between 0 and 63 with the value 0 specifying no shift is to be made.

(2) *Subfield two.* Subfield two specifies the value of the H field (the register to be shifted). The value must be between 0 and 15.

(3) *Subfield three.* The third subfield specifies the value of the tally register (R) field for shift and count and normalize instructions. The value must be between 0 and 15.

Table 5-3. Extended Mnemonics for the Full Word Shift Options of the Shift Full Word Instruction

Extended code	Meaning	T Field (hexadecimal)
SARF	Shift algebraic, right, (linear) full word	00
SLARF	Shift logical, right, (linear) full word	04
SCRF	Shift circular, right, full word	05
SALF	Shift algebraic, (linear) full word	02
SLLF	Shift logical, left, (linear) full word	06
SCLF	Shift circular, left, full word	07
SNF	Normalize, full word	12
SCF	Shift and count (linear), full word	16
SCCF	Shift and count, circular, full word	17

20079-12

Table 5-4. Extended Mnemonics for the Double Word Shift Options of the Shift Full Word Instruction

Extended code	Meaning	T Field (hexadecimal)
SARD	Shift algebraic, right, (linear) double word	08
SLRD	Shift logical, right, (linear) double word	0C
SCRD	Shift circular (logical), right, double word	0D
SALD	Shift algebraic, left, (linear) double word	0A
SLLD	Shift logical, left, (linear) double word	0E
SCLD	Shift circular, left, double word	0F
SND	Normalize, double word	1A
RFT	Reflect, double word	1F

20079-13

j. Extended Mnemonic Code Instruction Conversions for Shifting. The assembler conversion of the extended mnemonic code shift instructions is illustrated in figure 5-2. The assembler converts the extended mnemonic into the recognized machine language instruction and sets the appropriate values into the fields of the instruction word.

k. Extended Mnemonics for Format Instructions. The extended mnemonic codes for format instructions imply the format insert full (FIF), format insert half (FIH), format extract full (FEF), and format extract half (FEH) space function codes and a value for the T field (option field) of the instruction. Since the T field replaces part of the normal operand field, the literal mode (mode 0) must be used with the extended mnemonic codes. The notation used for the format instructions is snown in table 5-6. Table 5-7 shows the allowable extended mnemonics and the implied T fields for the format insert full and half (FIF and FIH), and format extract full and half (FEF and FEH) instructions. Detailed explanations of the extended mnemonic codes for the format instructions are presented in the following paragraphs.

Table 5-5. Extended Mnemonics for the Shift Half Word Instructions

Extended code	Meaning	T Field (hexadecimal)
SARH	Shift algebraic, right, (linear) half word	00
SLRH	Shift logical, right, (linear) half word	04
SCRH	Shift circular (logical), right, half word	05
SALH	Shift algebraic, left, (linear) half word	02
SLLH	Shift logical, left, (linear) half word	06
SCLH	Shift circular, left, half word	07
SNH	Normalize, half word	12
SCH	Shift and count (linear), half word	16
SCCH	Shift and count, circular, half word	17

20079-14

l. Extended Mnemonic Format Instruction Operands. The operand entry for extended mnemonic shift instructions is specified in the following paragraphs. Note that the mask register is always register 14. Some examples of extended mnemonic format instructions are shown following the explanations of the three operand subfields required in coding the extended mnemonics.

(1) *Subfield one.* This subfield specifies the mode, which must be literal (denoted by an equal sign). It also denotes the value of the shift (K) field. The shift value must be between 0 and 63 with the value zero (0) specifying no shift is to be made.

(2) *Subfield two.* Subfield two specifies the value of the H field (the 'from' register). The value must be between 0 and 15.

(3) *Subfield three.* Subfield three specifies the value of the R field (the 'to' register). The value must be between 0 and 15.

Examples:

FELRF =8,8,11
Format Extract Full, Right Linear Shift
Shift Count = 8
Origin (From) Register = 8
Destination (To) Register = 11

FILLF =8,7,12
Format Insert Full, Left Linear Shift
Shift Count = 8

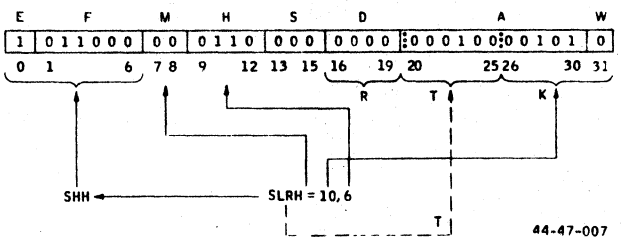


Figure 5-2. Assembler Language Extended Mnemonic Shift Instruction Conversion

Table 5-6. Notation Used in Extended Mnemonic Format Instruction Codes

Position*	Letter	Meaning
1	F	Format
2	E	Extract
	I	Insert
3	L	Linear
	C	Circular
4	R	Right
	L	Left
5	F	Full
	H	Half

*all format extended mnemonic codes are 5 letters in length.

20079-15

Table 5-7. Extended Mnemonics for Format Instructions

Extended code	Meaning	Operation code	T Field
FELRF	Format extract; (shift)linear, right, full word	FEF	0
FECRF	Format extract; (shift)circular, right, full word	FEF	1
FELLF	Format extract; (shift)linear, left, full word	FEF	2
FECLF	Format extract; (shift) circular, left, full word	FEF	3
FELRH	Format extract; (shift)linear, right, half word	FEH	0
FECRH	Format extract; (shift)circular, right, half word	FEH	1
FELLH	Format extract; (shift)linear, left, half word	FEH	2
FECLH	Format extract; (shift)circular, left, half word	FEH	3
FILRF	Format insert; (shift)linear, right, full word	FIF	0
FICRF	Format insert; (shift)circular, right, full word	FIF	1
FILLF	Format insert; (shift)linear, left, full word	FIF	2
FICLF	Format insert; (shift)circular, left, full word	FIF	3
FILRH	Format insert; (shift)linear, right, half word	FIH	0
FICRH	Format insert; (shift)circular, right, half word	FIH	1
FILLH	Format insert; (shift)linear, left, half word	FIH	2
FICLH	Format insert; (shift)circular, left, half word	FIH	3

20079-16

Origin (From) Register = 7
Destination (To) Register = 12

m. Extended Mnemonic Code Instruction Conversions for Format Instructions. The assembler conversion of the extended mnemonic code for format instructions is illustrated in figure 5-3. The assembler converts the extended mnemonic into the recognized machine language instruction and sets the appropriate values into the fields of the instruction word.

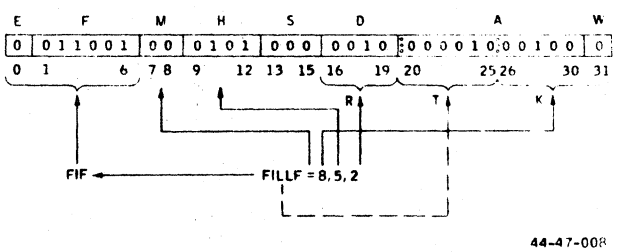


Figure 5-3. Assembler Language Extended Mnemonic Format Instruction Conversion

CHAPTER 6
COMPOOL SYMBOLS

Section I. GENERAL

6-1. Introduction

This chapter defines the term Compool and discusses the use of symbols in a Compool and the methods of converting Compool symbols to AN/GYK-12 machine instructions.

6-2. Compool

A Compool is defined as a collection of information relating to one or more program modules that make up a system. This collection of information takes the form of items, tables, parameters, and programs. Compool sets up a common data base that may be accessed by all programs in a system. As a result of Compool, the need for a repeated data base in each program in a system is eliminated.

6-3. Compool Symbol Usage

Symbols defined in a Compool may be used in the operand field. The assembler searches the Compool for the specified symbol, and if found, utilizes the attributes specified in the Compool for the symbol to derive the value of the operand. The programmer may specify that an address is to be provided for a Compool-defined table or item; that a mask is to be made up for an item; or that a shift value is to be supplied to move an item from where it normally resides within a word to a specified bit position. The method used by the assembler to interpret what is to be provided, depends upon the type of instruction being specified, the subfield of the operand containing the Compool symbol, and the type of symbol encountered.

6-4. Converting Compool Symbols

Methods of converting Compool symbols to AN/GYK-12 machine instruction operands are standard operands, and operands for extended mnemonic instructions as described in the following paragraphs.

a. Standard operands. Standard operands are comprised of subfield one and subfield two.

(1) *Subfield one.* In subfield one, a Compool symbol may specify an address or a mask. The method of specifying an address is to use the Compool symbol itself as a term. To specify a mask, the Compool item symbol is enclosed in single quotes. The assembler will generate a string of ONE bits for each bit of the item. For example, if an item begins in bit eight and is four bits in length, the mask will consist of ONE bits in bit positions eight through eleven, with all of the other bits equal to ZERO. If a mask of an item set to a particular value is desired, the item symbol in quotes is followed by a decimal number which specifies the value of the mask. The assembler will insert the binary equivalent of the specified decimal number in the bit positions occupied by the item. Refer to table 6-1 for examples of Compool symbols.

Table 6-1. Examples of Compool Symbols

Mnemonic	Compool defined item and register	Remarks
LDF	TPOS,10	Assembler provides address of TPOS
LDF	='TPOS',10	Assembler provides mask of TPOS
LDF	='TPOS'6,10	Assembler provides mask of TPOS set to a value of six

20079-17

(2) *Subfield two.* In subfield two, the Compool symbols have special significance only for the set and test bit instruction. The number of the bit to be set, reset, or tested may be specified relative to the least significant bit of an item. The symbol for the item may be specified in subfield one (also specifying the address of the item), with the relative bit number appearing in subfield two; or the symbol may appear in subfield two, en-

closed in single quotes, followed by the relative bit number. In either case, the relative bit number is applied to the least significant bit of the item to determine the absolute bit number. Refer to table 6-2 for examples of set bit instructions.

b. Operands for Extended Mnemonic Instruction. Operands for extended mnemonic instructions are provided for transfer, shift and format instructions.

(1) *Transfer instructions.* The Compool symbol usage for transfer commands is the same as specified for the standard operand (refer to paragraph 6.4a above).

(2) *Shift and format instructions.* For shift and format instruction, the number of shifts may be specified in subfield one by indicating the bit position to be shifted to and the item symbol to be shifted. The item symbol is specified, enclosed in single quotes, with the bit number to which the item is to be shifted specified as a decimal number outside of the quotes. If the item is to be left-justified to a bit number, that number appears to the left of the symbol; if right-justified, it appears to the right. The assembler will compute the number of shifts necessary to align the most-significant bit (for left alignment), or least significant bit (for right alignment), on the specified bit number. The type of shift will be determined from the operation mnemonic, but the direction of the shift will be that direction which requires the smallest number of shifts. Refer to table 6-3 for examples of shift and format instructions.

Table 6-2. Bit Instruction Examples

Set Bit instr (mnemonic)	Compool defined item and register	Remarks
SBT	TPOS, 2	Set bit two of the item TPOS
SBT	TPOS, 'TPOS'2	Set bit two of the item TPOS
SBT	R'10', 'TPOS'2	Set the bit in register 10 which is the same as bit 2 of item TPOS

20079-18

Table 6-3. Shift and Format Instruction Examples

Shift and format instructions	Compool defined item and register	Remarks
FECRF	='TPOS'31,10	Align TPOS on bit 31, shift from LSB of TPOS to bit 31
FILLF	=0'TPOS',10	Insert bits starting with bit 0 into bits occupied by TPOS, after shifting bit 0 to the MSB of TPOS

20079-19

CHAPTER 7
AN/GYK-12 ASSEMBLER INSTRUCTIONS

Section I. GENERAL

7-1. Introduction

This chapter discusses the AN/GYK-12 assembler instructions. Each instruction is categorized, defined, and explained in terms of operand content, format, and usage. Unlike the machine instructions, assembler instructions are for use during assembly only. They are not for execution during the object program operation and do not produce any object code. Instructions such as block start symbol (BSS) and generate data (GEN) generate no machine instructions but do cause storage areas to be set aside for constants and other data. Others, such as equate (EQU) and PAGE are effective only at assembly time and they generate nothing in the assembled program and have no effect on the location counter. Table 7-1 lists all of the AN/GYK-12 assembly instructions within their various categories and opposite their corresponding purposes.

7-2. Symbol Definition Instruction (EQU)

The instruction in the chart below, assigns the expression in the operand field to the symbol that appears in the name field.

Name	Operation	Operand
A symbol	EQU	An expression

The expression may be symbolic, numeric, or mixed. Both the value and relocatability attributes of the expression in the operand are assigned to the name field. Evaluation of an expression that results in a value greater than 16 bits in magnitude will result in an assembly limit error (L). If the expression in the operand field is not defined previous to the EQU instruction, it is saved in a file by the assembler. At the completion of the first pass of the assembler, these undefined EQU instructions are read and further attempts to resolve the operand field will be made. As each instruction is resolved, it is removed from the file, and the pass through the file is continued. If a pass through the file results in no instruction being resolved, the file is closed and all unresolved instructions are flagged as undefined. The EQU instruction is a means of equating symbols to immediate data, register numbers, and other arbitrary values.

The instructions:

```
A  EQU   '1500'
```

will assign the value of octal 1500 to the symbolic tag A.

The instructions:

```
A  EQU   R'3'
```

will equate the address of register three to the name A. In a program, the assembler will recognize within the expression B+A that the mode is direct, indexing, is desired, and that the symbol, A, represents index register three.

The instruction:

```
A  EQU   STAG
```

will cause the address of the symbol, STAG, to be assigned to the symbol A thus equating the two symbols. STAG must be defined itself somewhere within the assembly. Two symbols which are equated actually represent the same location with two different names.

Table 7-1. AN/GYK-12 Assembler Instructions

INSTRUCTION	CODE	NAME	MEANING
System Definition	EQU	Equate	Equate Symbol
Data Definition	GEN	Generate	Generate data
	BSS	Block start symbol	Block started by symbol
	DATA	Data	Define length of local data
Program Linking	ENT	Entry	Identify entry point symbol
	EXT	External	Identify external symbol
Listing Control	REM	Remark	Remark
	PAGE	Page	Start new page
	OPT	Option	Specify assembler input and outputs
Program Control	TITLE	Title	Identify assembly module
	ORG	Origin	Set location counter to relocatable value
	LOC	Location	Set location counter to non-relocatable value
	END	End	End assembly
	CMP	Compool	Fetch compool
Miscellaneous	CNOI	Conditional no operation	Align location

20079-20

Section II. DATA DEFINITION INSTRUCTIONS

7-3. GEN, BSS, Data Instructions

These statements are used to enter data constants into storage and to define and reserve areas of storage. The statements may be named so that other program statements can refer to the fields generated by them.

7-4. Generate Data Instruction (GEN)

The generate data instruction (GEN) is used to provide constant data in storage. (See chart below.) Each instruction may provide one constant or a series of constants. All types of terms and expressions may be generated, as well as address constants. The generated constants receive the same relocatability attributes as the expressions from which they are defined.

Name	Operation	Operand
A symbol or blank	GEN	One or more operands in the format specified below, separated by commas.

Each operand consists of three subfields. The first two subfields describe the constant, and the third provides the constant or constants. The first and third subfields may be omitted but the second must be specified. Note that more than one constant may be specified in the third subfield for most types of constants. Each operand so specified must be of the same type; the descriptive

subfields that precede them apply to all of them. The subfields of each GEN operand are written in the following sequence:

1 2 3

Duplication Factor Type Constant(s)

Although the constants specified in one operand must have the same characteristics, each operand may specify different types of constants. For example, a double word generate might specify a hexadecimal operand, an octal operand, and a character operand. Some examples of GEN instructions are shown in table 7-2.

Table 7-2. Example of the Use of the Generate (GEN) Instruction

Examples of GEN statements:		
GEN	HC'AB'	(half word; characters A and B)
GEN	3DX'FA00A800'	(double word; hex FA00A800, three times)
GEN	FABCD	(full word; symbol ABCD)
GEN	FC'THIS IS A TEST'	(four full words; characters THIS IS A TEST)
GEN	HX'FF00',0, O'75',ABCD	(four half words; hex FF00, decimal 0, octal 75, and symbol ABCD)

20079-21

a. Operand Subfield One, Duplication Factor. The duplication factor may be omitted. If specified it causes the constant(s) appearing in subfield three to be generated the number of times specified along with the attributes specified in subfield two. The duplication factor is applied after the constant is assembled and is the same as if the entire GEN statement were repeated the stated number of times. A duplication factor equal to zero forces the location counter to a double, full, or half word boundary depending upon the type specified in subfield two.

b. Operand Subfield Two, Type Designation. The type subfield defines the type of constant being specified. From the type specified, the assembler determines how it is to interpret the constant and translate it into the appropriate AN/GYK-12 machine format. The type of GEN is specified by a single letter code as follows:

D - Double Word Constant

F - Full Word Constant

H - Half Word Constant

c. Operand Subfield Three, Constant Entry. This subfield supplies the constant described by the subfields that precede it. The constants may consist of any allowable term or expression. For character (EBCDIC ro ASCII) constants, as many characters as desired may be specified, and space will be generated for all characters specified. All other constants must fit within the specified storage area, i.e., half word, full word, or double word. The total storage requirement of an operand is the product of the length times the number of constants in the operand. The total storage requirement of a GEN instruction is this factor times the duplication factor, if any is present. All constant types are aligned on the proper boundary by the assembler.

7-5. Block Started by Symbol Instruction (BSS)

The BSS instruction is used to reserve areas of storage and to assign names to those areas. (See chart below.)

Name	Operation	Operand
A symbol or blank	BSS	An expression

The size of the storage area that can be reserved by a BSS instruction is limited only by the maximum value of the location counter (65,535). The BSS instruction causes the location counter to be incremented by the value of the operand field. Therefore the storage area defined by the BSS instruction is effectively placed where the BSS instruction was itself located in the object program. No machine words are generated in response to a BSS instruction, however a storage area of the size specified is set aside. Note also that the area defined by a BSS is not reset when the program is loaded and therefore the program in which the BSS appears must reset the area to zero or some other value if that is required. Any symbol which appears in the operand field of a BSS instruction must have been previously defined.

The instruction:

TAG BSS 16

will reserve an area in line for storage which is sixteen half words in length. The area will be assigned the symbolic tag TAG so that it may be

referred to by other instructions in the program. The address of the first half word in the contiguous area will be assigned to the symbol TAG.

The instruction:

TAG BSS NUMBER

will reserve an area in line for storage which is a given number of half words in length. The length of the area in half words, will be equal to the value assigned to the symbol NUMBER by the assembler. Note that the symbol NUMBER must be previously defined before it can be used in a BSS instruction.

The instruction:

TAG BSS 10*NUMBER+12

will reserve an area of length in half words equal to the value of the expression appearing in the operand field of the BSS instruction. Note that the symbol NUMBER must be previously defined by the assembler.

7-6. Define Length of Local Data Instruction (DATA)

The DATA instruction is used by the TACPOL compiler to pass to the assembler the length of the local data associated with the assembly module. (See chart below).

Name	Operation	Operand
Blank	DATA	An expression

The size of local data is output by the assembler on the program identifier (type one) card. Any symbol which appears in the operand field of a DATA instruction must have previously defined.

Section III. PROGRAM LINKING INSTRUCTIONS

7-7. ENT, EXT Instructions

These instructions are used for linking programs by providing entry points for program modules. These points can be accessed by any type of assembler language machine instructions but can be particularly useful in linking modules of separate assemblies together. An assembly module is a group of instructions which are assembled as one entity by the assembly program. A module begins with a TITLE instruction and ends with an END instruction (An OPT instruction may precede the TITLE instruction if it is included).

a. Identify Entry Point Symbol Instruction (ENT). The ENT instruction identifies linkage symbols that are defined in this assembly module but may be used by some other module. (See chart below.)

Name	Operation	Operand
Blank	ENT	One or more symbols, separated by commas, that also appear as statement names.

A module may contain a maximum of 100 entry point symbols. Entry symbols which are not defined (do not appear as statement names) count towards this maximum. The symbols in the ENT operand field may be used as operands by other assembly modules. More than one entry point symbol may be included in the operand of an ENT instruction.

The instruction:

ENT SINE,ARCSINE

defines the two symbols SINE and ARCSINE as available to other assembly modules. That is, the addresses of these symbols are available to assembly modules other than one in which they are defined. Therefore, they are defined as entry points.

b. Identify External Symbol Instruction (EXT). The EXT instruction identifies linkage symbols that are to be used by this assembly module but are defined in some other models. Each external symbol must be identified. (See chart below.)

Name	Operation	Operand
Blank	EXT	One or more symbols, separated by commas.

When external symbols are used, they must be the only terms in the expression containing them. The reason for this restriction is that the D, A, W field of the assembler language machine instruction in which the external symbol appears is used

for chaining the various references to the same symbol. Any modifications to this field would result in invalid chaining information.

The instruction:

```
EXT                START,COMPUTE
```

identifies the two symbols, START and COMPUTE as being defined in some module other than this one. Note that more than one external symbol may be defined with a single EXT instruction.

Section IV. LISTING CONTROL INSTRUCTIONS

7-5. REM, PAGE, OPT Instructions

The listing control instructions provide information regarding the inputs to and outputs from the assembler.

a. Remark Instruction (REM). The REM instruction is used to provide additional annotation on the assembly listing. (See chart below.)

Name	Operation	Operand
Blank	REM	A sequence of characters

The entire operand and comments fields may be used for comments. The operation code, REM, is not printed on the assembly listing. Only the contents of the operand and comments fields are printed. An asterisk in column 1 has the same effect as the REM instruction.

b. Start New Page Instruction (PAGE). The page instruction causes the next line of listing to appear at the top of a new page. (See chart below.) This instruction provides a convenient way to separate routines in the assembly module listings.

Name	Operation	Operand
Blank	PAGE	Blank

c. Specify Input/Output Options Instruction (OPT). The OPT instruction is used to specify the location and nature of the input to the assembler (the source deck) and the output from the assembler (the object desk and the assembly listing). (See chart below.)

Name	Operation	Operand
Blank	OPT	One or more operands, in the format specified below, separated by commas.

The option instruction applies to all assembly modules in any job package. It is an optional instruction, and its absence implies a statement with the operand format CARD, CARD, and PRINTER: such as, OPT CARD, CARD, PRINTER. The first three subfields must appear in the order shown. Operand subfields four through eleven may appear in any order. Any of the first three subfields may be deleted by replacing it with a comma with no preceding or following blanks.

The instruction:

```
OPT                CARD,CARD,PRINTER,
                   XREF,GEN,DATA,
                   TAGS,RELOC
```

specifies that the source deck input to the assembler is on cards, the object deck output is on cards, and the assembly listing will be produced on the printer. In addition all of the optional assembler outputs listed in paragraph 7-5c(4) will be provided.

(1) *Operand subfield one.* Operand subfield one identifies the input device which contains the source deck. CARD, for card reader, TAPE, for magnetic tape unit, or RAM, for random access memory device such as disk, may be specified. Only one of the options may be specified and if none is specified, CARD will be assumed. The deletion of subfield one is indicated by a comma with no blanks preceding or following.

(2) *Operand subfield two.* Operand subfield two identifies the output device or devices which are to be used to output the binary object records. These may be any combination of CARD, TAPE, or RAM. If more than one option for object output is desired, then the individual options must be separated by a plus sign, e.g., CARD+TAPE+RAM. Operand subfield two may be replaced with a comma with no blanks preceding or following. If none are specified, there will be no object code output. The CARD, TAPE, and

RAM options reference the devices as indicated in paragraph 7-5c(2).

(3) *Operand subfield three.* Operand subfield three identifies the output device or devices which are to be used to output the assembly listing. These may be any combination of PRINTER, TAPE, and RAM. As with subfield two if more than one option is specified, then each individual option must be separated by a plus sign, e.g., PRINTER+TAPE+RAM. If none are specified, there will be no assembly listing output. Subfield three may be deleted by inserting a comma, preceded and followed by no blanks.

(4) *Operand subfields four through eleven.* Operand subfields four through eleven may include any or all of the options from the following list in any sequence. If not all the options are used, then there will be fewer than eleven total subfields for that particular OPT instruction.

1) XREF A cross reference listing is provided.

2) GEN All statements generated by macro instructions are printed.

3) DATA Constants are printed out in full in the listing. Otherwise only the first of a series of duplicated constants is printed and only the first full word of constants which are longer than 32 bits, even those which are not duplicated.

4) TAGS The symbol table is included in the object deck. (The type two cards are punched into the object deck.)

5) RELOC Relocation indicators are included in the object deck. This is necessary if the program which was assembled is going to be relocated when loaded.

The lack of any of these operand subfields implies the opposite meaning of the operand subfield if it were present.

Section V. PROGRAM CONTROL INSTRUCTIONS

7-6. TITLE, ORG, LOC, END Instructions

The program control (TITLE, ORG, LOC, END) instructions are used to identify the beginning of an assembly module, to set the location counter to a value at the start of the assembly, and to identify the end of the assembly module.

a. Identify Assembly Module Instruction (TITLE). The TITLE instruction enables the programmer to identify the assembly listing and to name the assembly module being produced. (See chart below.)

Name	Operation	Operand
Name	TITLE	A sequence of characters, enclosed in apostrophes.

It appears as the first instruction of the assembly module. Only the OPT card, if included, may precede the TITLE card. The name field of the TITLE card is used to identify the assembly module name which will appear on the object deck. It may consist of from one to eight alphabetic or numeric characters in any combination. The name field of the TITLE card is interpreted only on the first TITLE instruction appearing in the assembly module. Name fields of the other TITLE instructions, if any, are ignored. The name in the name field on the first TITLE instruction will become the module name. The operand field of the TITLE card may contain up to 65 characters enclosed in single apostrophes (which will not be printed) and will be printed at the top of each page of the assembly listing. If a single apostrophe character is desired in the title printout, two consecutive apostrophes must appear in the operand of the TITLE card. Although only the first TITLE instruction is interpreted for its name field, all TITLE instructions have their operand interpreted and the contents used for the heading of all the pages of assembly listing to follow. Each TITLE instruction causes the listing to be advanced to a new page before the new heading is printed.

The instruction:

```
PROGI   TITLE   'FIRST HEADING'
```

will cause the assembly module to be named PROG1 (assuming this to be the first TITLE instruction in the module, otherwise the module name already assigned will remain unchanged). The heading, FIRST HEADING, will be printed at the top of the first page of the assembly listing and each subsequent page until a new TITLE instruction is encountered.

This instruction:

TITLE 'SECOND HEADING'

will cause a new page to be restored and the new heading, SECOND HEADING, to be printed at the top of the new page and on each subsequent page until another TITLE instruction is encountered.

b. Set Location Counter to Relocatable Value Instruction (ORG). The ORG (Origin) instruction (see chart below) is used to set the software location counter to a relocatable value.

Name	Operation	Operand
Blank	ORG	An expression

The expression may be symbolic, numeric, or mixed. However, if a symbolic tag is used in the operand field, it must be defined within the current assembly or in the Compool. If the ORG instruction is not present, the assembly will begin at location zero. The ORG instruction may be placed anywhere in the program.

The instruction:

ORG X'7F00'

will set the location counter to the value 7F00 hexadecimal, and define the assembly module as relocatable.

The instruction:

ORG *+20

will skip over the next twenty halfwords from the current value of the software location counter and start the software location counter from that point. As with the previous example, the assembly module in which this ORG instruction appears will also be relocatable.

c. Set Location Counter to Non-Relocatable Value Instruction (LOC). The LOC (Location) instruction (see chart below) is used to set the software location counter to an absolute value, and thus identify the assembly module or area of code which follows the instruction as being absolute, or non-relocatable.

Name	Operation	Operand
Blank	LOC	An expression

Any symbols in the expression must have been previously defined. The LOC instruction can be used anywhere in the program.

The instruction:

LOC '7000'

will set the software location counter to an initial value of octal 7000 and identify the module or area following the instruction as non-relocatable.

The instruction:

LOC *-20

will cause the previous twenty halfwords to be overlaid, and identify the instructions following as non-relocatable.

d. End Assembly Instruction (END). This instruction terminates the assembly of a program and appears as the last card of the symbolic program (source deck). The END statement notifies the assembler that all symbolic cards have been processed. (See chart below.)

Name	Operation	Operand
Blank	END	One or two operands, in the format specified below, separated by commas.

It may also designate a point in the program to which control may be transferred after the program is loaded, and the program level which is to be used by the program. The END instruction must always be the last statement in the source program.

The content of the operand field is as follows:

Operand 1 - Assembly module entry point

Operand 2 - Assembly module program level

Both operands are optional. If an entry point is specified (octal, decimal, hexadecimal, or symbolic), the evaluated value of the operand will

appear on the Type 9 card of the object deck. If operand one is used, the use of operand two is implied and if operand two is blank, program level 63 is assumed.

The instruction:

END

merely specified the end of an assembly module.

The instruction:

END BEGIN,63

specifies the end of an assembly module whose entry point is the label BEGIN, and whose program level is to be 63. The specification of level 63 is redundant since level 63 is assumed if operand two is blank. However, its inclusion may make the program listing easier to understand.

7-7. Miscellaneous Instructions: CMP, CNOI

There are two miscellaneous assembly instructions which provide added capability for the user.

a. Compool Instruction: CMP. The CMP instruction is used to specify the identity of a Compool for use by the assembler during the current assembly operation. (See chart below.)

Name	Operation	Operand
Blank	CMP	COMPOOL identifier

The Compool identifier consists of from one to eight alphabetic or numeric characters in any combination. The Compool being specified must be available to the assembler.

The instruction:

CMP COMP1

instructs the assembler to load the COMP1 Compool from the system tape (or other device) and use it during the affected assembly.

b. Conditional No Operation Instruction: CNOI. The purpose of the Conditional No Operation Instruction is to align the software location counter on a full word (32 bit) or a double word boundary. (See chart below.)

Name	Operation	Operand
A symbol or blank (optional)	CNOI	Two subfields separated by a comma, a 0 or 2 in subfield 1, and a 2 or 4 in subfield 2.

The assembler will insert No Operation (NOI) machine instructions into the object code as necessary to accomplish this. The CNOI instruction has two subfields.

(1) The first subfield may be either 0 or 2 and it indicated the number of 16 bit addresses the software location counter is to be advanced after the full word or double word alignment is made. The assembler will insert an NOI instruction only if the value of the subfield is 2.

(2) The second subfield determines whether the alignment will be on a full or double word. If the value of the second subfield is 2, a fullword alignment will be made. If the value of the subfield is 4, a double word alignment is made. Single address positions (16 bit half words) will be inserted along with 32 bit words consisting of NOI instructions as necessary.

The instruction:

CNOI 0, 4

will cause the software location counter to be advanced to the next double word boundary inserting NOI instructions as required.

The instruction:

CNOI 2, 4

will cause the software location counter to be advanced to the next double word boundary and further advanced an additional full word. NOI instructions will be inserted as necessary.

7-8. Assembly Deck Structure

Listed below are the major control cards, in an order in which they would normally be used.

```
OPT
TITLE
CMP
ORG (or LOC)
  .
(source program instructions)
  .
  .
END
```

CHAPTER 8
MACRO LANGUAGE

Section I. GENERAL

8-1. Introduction

This section discusses the AN/GYK-12 assembler macro language. Detailed explanations along with illustrations and examples are presented. The AN/GYK-12 macro language is an extension of the assembler language. The macro language provides a convenient way for the programmer to generate a desired sequence of assembler language instructions many times in one or more programs. The macro definition is written only once, and a single instruction (a macro instruction) is written each time a programmer wants to generate the desired sequence of instructions.

8-2. The Macro Instruction Statement

A macro instruction is a source program statement. The assembler generates a sequence of assembler language instructions for each occurrence of the same macro instruction. The generated instructions are then processed like any other assembler language instructions. Three types of macro instructions may be written: positional, keyword, and mixed mode. Positional macro instructions permit the programmer to write the operands of a macro instruction in a fixed order. Keyword macro instructions permit the programmer to write the operands of a macro instruction in a variable order. Mixed mode macro instructions permit the programmer to use the features of both positional and keyword macro instructions in the same macro instruction.

a. Defining Macro Instructions. The definition of a macro instruction (macro) consists of a set of statements which provide the assembler with the mnemonic operation code and format of the macro instruction, and the sequence of instructions the assembler is to generate each time the macro instruction appears in the source program.

Every macro definition consists of the following:

(1) A macro definition header statement.

(2) A macro instruction prototype statement.

(3) One or more macro model instruction statements.

(4) A macro definition trailer statement.

All macro definitions must appear in the source program before they are referenced as macro instructions.

b. The Macro Library. The same macro definition may be made available to more than one source program by placing the macro definition in the macro library. The macro library is a collection of macro definitions that can be used by all the assembler language programs in an installation. Once a macro definition has been placed in the macro library, it may be used by writing its corresponding macro instruction in a source program.

c. System Macro Instructions. The macro instructions that correspond to macro definitions prepared are called system macro instructions, and are in a permanent library called the system macro library. They may be used by writing their corresponding macro instructions in a source program, as above. However, this library may not be modified by the programmer.

d. Variable Symbols. A variable symbol is a type of symbol that is assigned different values by either the programmer or the assembler. When the assembler uses a macro definition to determine what statements are to replace a macro instruction, variable symbols in the model statements are replaced with the values assigned to them. By changing the values assigned to variable symbols, the programmer can vary parts of the generated instructions. A variable symbol is written as an ampersand followed by from one to eight letters and/or digits. At least one of the characters in the variable symbol must be a letter.

e. Symbolic Parameters. A symbolic parameter is a type of variable symbol that is assigned

values by the programmer when he writes a macro instruction. The programmer may vary instructions that are generated for each occurrence of a macro instruction by varying the values assigned to symbolic parameters. A symbolic parameter consists of an ampersand followed by from one to eight letters and/or digits, one of which must be a letter.

(1) The following are valid symbolic parameters (or variable symbols):

(a) &READER.

(b) &A.

(c) &12A7.

(d) &X4A0.

(2) The following are invalid symbolic paramters (or variable symbols) for the reasons noted:

(a)	CARDAREA	(first character not an ampersand)
(b)	&2564	(no letters following the ampersand)
(c)	&AREA12345	(more than eight characters following the ampersand)
(d)	&BCD%34	(contains a special character other than the initial ampersand)

Any symbolic parameters which appear in a model statement must also appear in the prototype statement of the macro definition.

f. Macro Definition Header Statement. The macro definition header statement (see chart below) must be the first statement used in defining a macro. The name field of the statement must be blank, and the operation field must contain MACRO.

Name	Operation	Operand
Blank	MACRO	T, M, P, or D

The operand field of the statement may contain either T, M, P, or D, or be left blank. These letters specify the macro library action to be taken, as shown in table 8-1. Only the first column of the operand is interpreted; therefore, only one option is allowed. If more than one option is specified, only the first one applies.

Table 8-1. Macro Definition Header Statement Operand Field

Operand character	Remarks
T	Instructs the assembler that the following macro is temporary and that the macro should only be used for this assembly. If no letter is present, a T is assumed.
M	Instructs the assembler that the following macro is a modification to an existing macro in the library. The macro defined will replace the appropriate macro in the library.
P	Instructs the assembler that the following macro is a permanent macro and should be added to the library. The macro defined will be added to the library.
D	Instructs the assembler that the macro defined should be deleted from the macro library. The macro defined on the prototype statement will be deleted from the library. (No model statements are required.)
blank	Option T is assumed.

20079-22

g. Macro Instruction Prototype Statement. The macro instruction prototype statement (refer to diagram) specifies to the assembler the mnemonic operation code and the format of the macro instruction that refers to the macro definition.

Name	Operation	Operand
Blank	A symbol	Zero or more symbolic parameters, separated by commas

The name field of the macro instruction prototype statement must be blank. If a symbolic tag is desired in the actual use of the macro instruction, a tag can be inserted in the name field by the programmer using the macro instruction at assembly time. The tag will then appear on the first statement of the macro expansion. The operation field will contain any collection of from one to five letters, A through Z. This is the mnemonic operation code for this macro. Any time a programmer uses this mnemonic in his coding, the

model statements for this macro will be inserted into the source program in place of the macro instruction which called for the action. The operand field is used in three different ways to provide the programmer with the ability to use the three different classes of macro instructions.

h. Positional Macro Prototype. In the positional macro, each symbolic tag which may be modified by the programmer is preceded by an ampersand (&). These tags are symbolic parameters. Examples of symbolic parameters are: &TAG, &TDAY, etc. The assembler determines where to put each of the programmer's tags in the macro expansion by their relative position in the operand.

i. Keyword Macro Prototype. In the keyword macro, each symbolic tag which may be modified by the programmer is followed by an equal sign. Examples: TAG=DAY, NO= =3, TALL=

NOTE

Literal value for NO is specified by double equal sign, one for the literal designation and one for the keyword symbolic tag designation.

In the keyword macro the assembler usage of the programmer's tags is based upon a matching of names and not position in the list. In the preceding example it should be noted that TALL is blank after the equal sign and TAG has DAY after the equal sign. This instructs the assembler that if TALL is not used by the programmer, a blank symbol name is assumed. if TAG is not used by the programmer, the assembler will assume DAY is the symbolic name.

j. Mixed Macro Prototype. The mixed macro is a combination of the positional and the keyword macro instructions. The positional portion of the macro operand is always first. The assembler will process this class of macro operand in the same manner as previously described for the positional and keyword macros. The mixed macro may contain both types of symbolic tags. Symbolic tags preceding an equal sign will be used as keyword symbolic tags and matched by name. Symbolic tags preceded by an ampersand will be positional symbolic tags and will be matched by position.

k. Macro Model Statements. Macro model statements are the macro definition statements from which the desired references of assembler language instructions are generated. Zero or more macro model statements may follow the prototype statement. A model statement consists of from one to four fields; name, operation, operand, and comments. The name field may be blank or it may contain a symbol or a symbolic parameter. The operation entry must be present and may contain any machine or assembly instruction, or a variable symbol. The operand entry may be blank or it may contain ordinary symbols or variable symbols. Model statement fields must follow the same rules for paired single quotes and blanks as machine and assembly instructions. The comments field may contain any combination of characters. No substitution is performed for variable symbols appearing in the comments field.

l. Macro Definition Trailer Statement. The macro definition trailer statement (see chart below) is used to signal the assembler that this is the end of a macro definition. The name field and the operand field are blank, and the only allowable entry in the operation field is MEND.

Name	Operation	Operand
Blank	MEND	Blank

Section II. MACRO DEFINITIONS

8-3. Macro Definition and Instruction Examples

Several examples of macro definitions and the macro instructions which reference them follow. All of the examples cause the following assembler language expansion:

SDH	TAG1, 3
LDH	TAG2, 3
ADH	TAG3, 3
LDF	TAG4, 8
LDH	TAG5, 9

a. Nonreference Macro (Temporary)

DEFINITION	HEADER	MACRO	T
	PROTOTYPE	COMP	
	MODEL	SDH	TAG1,3
	·	LDH	TAG2,3
	·	ADH	TAG3,3
	·	LDF	TAG4,8
	·	LDH	TAG5,9
	TRAILER	MEND	
INSTRUCTION		COMP	

b. Positional Macro (Modify)

DEFINITION	HEADER	MACRO	M
	PROTOTYPE	COMP	&A,&B,&C, &D,&E
	MODEL	SDH	&A,3
	·	LDH	&B,3
	·	ADH	&C,3
	·	LDF	&D,8
	·	LDH	&E,9
	TRAILER	MEND	
INSTRUCTION		COMP	TAG1, TAG2, TAG3, TAG4, TAG5

c. Nonreference and Positional Macro (Temporary)

DEFINITION	HEADER	MACRO	T
	PROTOTYPE	COMP	&A,&B
	MODEL	SDH	&A,3
	·	LDH	TAG2,3
	·	ADH	TAG3,3
	·	LDF	&B,8
	·	LDH	TAG5,9
	TRAILER	MEND	
INSTRUCTION		COMP	TAG1, TAG4

d. Keyword Macro (Permanent)

DEFINITION	HEADER	MACRO	P
	PROTOTYPE	COMP	A=,B=TAG2, C=,D=
	MODEL	SDH	&A,3
	·	LDH	&B,3
	·	ADH	&C,3
	·	LDF	&D,8
	·	LDH	TAG5,9
	TRAILER	MEND	
INSTRUCTION		COMP	A=TAG1, C=TAG3, D=TAG4

e. Mixed Macro (Temporary)

DEFINITION	HEADER	MACRO	T
	PROTOTYPE	COMP	&A,&B, C=TAG6
	MODEL	SDH	&A,3
	·	LDH	TAG2,3
	·	ADH	&B,3
	·	LDF	TAG4,8
	·	LDH	&C,9
	TRAILER	MEND	
INSTRUCTION		COMP	TAG1,TAG3, C=TAG5

f. Special Usage

DEFINITION	HEADER	MACRO	
	PROTOTYPE	COMP	&A,&B
	MODEL	SD&A	TAG1,3
	·	LD&A	TAG2,3
	·	AD&A	TAG3,3
	·	LD&B	TAG4,8
	·	LD&A	TAG5,9
	TRAILER	MEND	
INSTRUCTION		COMP	H,F

g. Deletion of Macro

```
HEADER      MACRO     D
PROTOTYPE   COMP
TRAILER     MEND
```

h. Tag Usage in a Macro. Assume the macro COMP as defined in the first example. Then the instruction below would call for the use of the macro COMP with the tag NAME as shown:

```
NAME        COMP
```

The following expansion would result. Note that the tag NAME has been placed in the name field of the first instruction of the macro expansion.

```
NAME  SDH   TAG1, 3
      LDH   TAG2, 3
      ADH   TAG3, 3
      LDF   TAG4, 8
      LDH   TAG5, 9
```

CHAPTER 9
OBJECT CODE OUTPUT

Section I. INTRODUCTION

9-1. Introduction

The object code (i.e., the machine-language result of the assembly process) is an output of the assembler in card-image format. The actual input/output (I/O) device used for outputting this code may be a card punch, paper tape punch, magnetic tape drive, or random-access memory. The specific I/O devices, that are to receive the object code, are specified on the Option (OPT) card. The object deck may be input to the computer by the use of a loader program. The object code is designed so that programs may be loaded in their assembled locations or relocated by the loader to any area of AN/GYK-12 core. The loader program also will resolve any external reference addresses at load time. The cards comprising assembler output are of nine types: 1, 0, 2, 3, 4, 5, 6, 7, and 9 (type 8 is not used). The following paragraphs describe these cards in the order of their appearance in the object deck, except for the Type 3 card, which immediately follows the Type 1.

9-2. Program Identifier Card (Type 1)

The program identifier card is the first card in the object deck. Its format is shown in the card format diagram in table 9-1.

Table 9-1. Program Identifier Card Format

Columns	Content
1	1
3-10	Assembly module name
12-15	Assembled load location (hex)
17-20	Length of module (hex)
22-25	Length of local data (hex)

20079-23

9-3. Data Cards (Type 0)

The data cards contain the AN/GYK-12 instructions and data which will be loaded into the AN/GYK-12 computer. The format of the type 0 cards is shown in the data card format diagram, table 9-2. Columns 1 and 74 are characters; all other columns are EBCDIC representations of binary data. If column 74 is blank, the entire card is relocatable; if not, the card is not relocatable. The relocation bits refer to the 32 possible half words on the card. If the bit in the corresponding relative position of the half word is equal to zero, the half word is not to be modified in the event of relocation. If the bit is equal to one, an offset value (equal to the load address minus the assembled module address) is to be added to the corresponding half word. The checksum is the value which, when added to the number of half words, the address of the first half word, all of the data half words, and the relocatability bits taken 16 at a time, equals zero (disregarding 16-bit overflow).

Table 9-2. Data Card Format

Columns	Content
1	0
2	Number of half words (16 bit) on card
3-4	Address of first word
5-68	Data
74	Not relocatable indicator
75-78	Relocation bits
79-80	Checksum

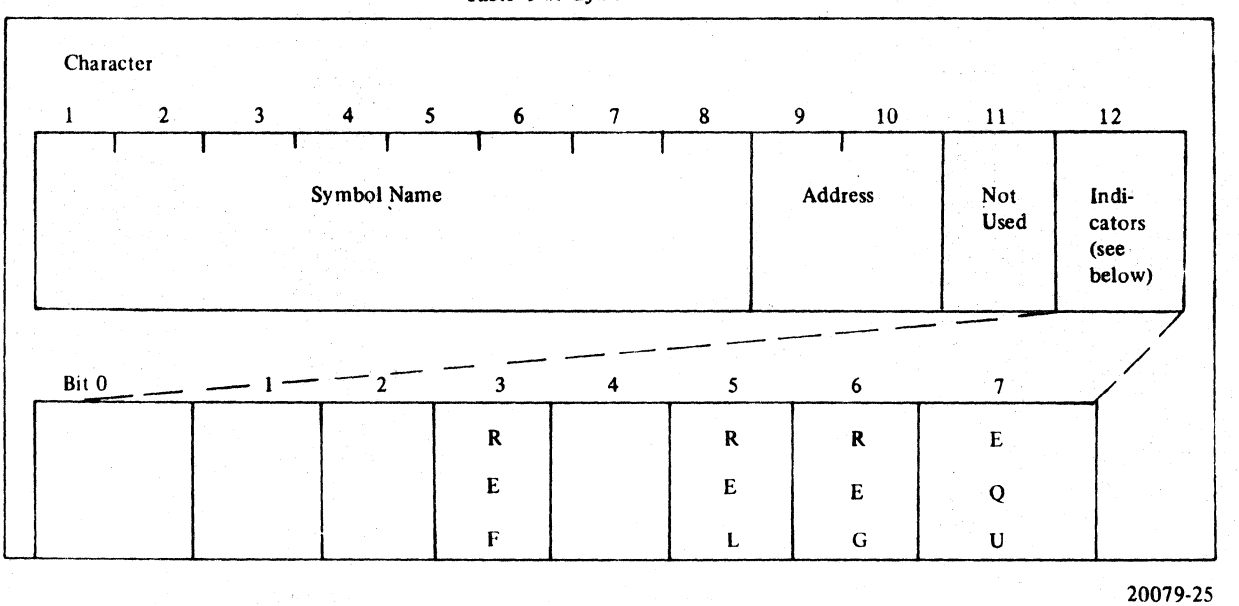
20079-24

9-4. Prologue Cards (Types 2, 3, 4, 5, 6, and 7)

With the exception of column 1 (which contains a Character 2, 3, 4, 5, 6, or 7) and columns 74-78 (which do not always contain relocatability information), the prologue cards have the same format as type 0 cards. Columns 5-76 are all used for data, which consists of: the internal symbol (tag) table for type 2 cards; the entry point symbol table for type 3 cards; the external reference table for type 4 cards; the program-accessed table for type 5 cards, the Compool-page-usage table for type 6 cards; and the Compool-reference linkage table for type 7 cards. This data is used by the loader program during the load process. The addresses on these cards begin at ZERO for each table. The table entries on the type 2, 3, and 4 cards are six half-words in length; those on the type 5 cards are four half-words long; those on the type 6 cards are four half-words long; and those on the type 7 cards are one half-word long.

a. Symbol Table Card (Type 2). The symbol table may be used by a loader program to enable the use of symbolic references to the name field identifiers within a program. The symbol table is also used by the assembler to satisfy symbolic references within the program. The format of the symbol table is shown in table 9-3. The Equate (EQU) indicator is equal to 1, if this symbol results from an EQU statement. The Register equate (REG) indicator is equal to one if the symbol resulted from an EQU statement whose operand was R'n'. The Relocatability (REL) indicator is equal to 1, if the symbol is relocatable: i.e., if it resides in an area of the program not covered by an LOC statement. The Referenced symbol (REF) indicator is equal to 1 if the symbol is referenced in the operand of another instruction in this assembly. The other bits are unused.

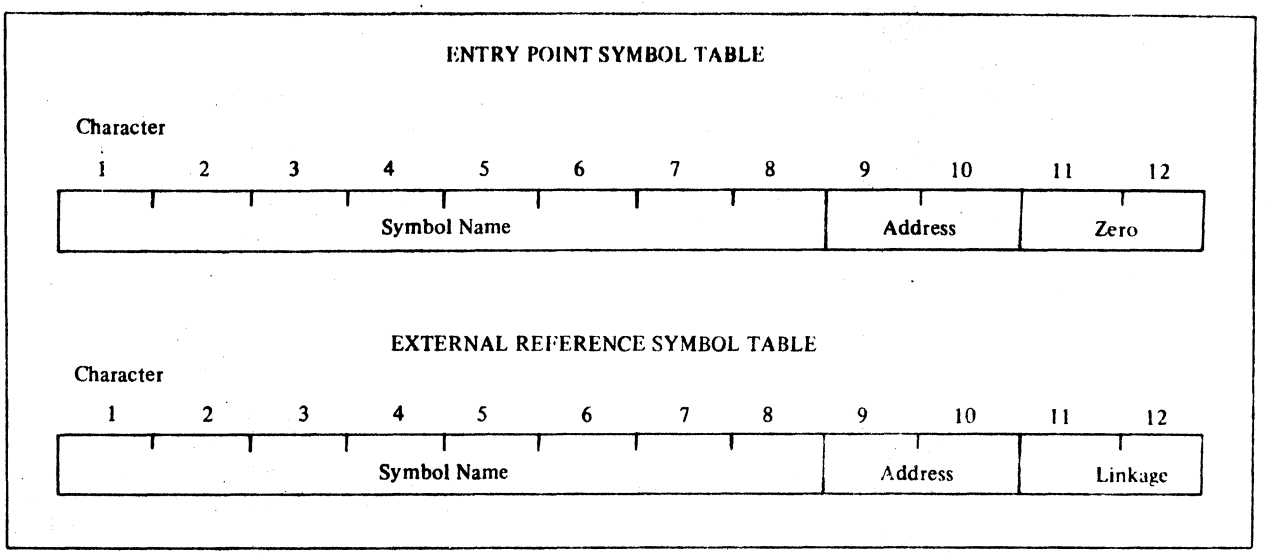
Table 9-3. Symbol Table Format



20079-25

b. Entry Point and External Reference Tables (Card Types 3 and 4). The entry point and external reference tables allow a loader program to satisfy references between programs. The entry table table defines the entry points in a program, and the external reference table defines references within a program to entry points in other programs. The format of these tables is shown in table 9-4. The address field contains the address of the symbol as evaluated at assembly time. The address will be equal to zero unless the symbol is defined within the assembly. If at load time the symbol is encountered in an entry symbol table, this address will be replaced by the entry address. The linkage field indicates the address of the last word which uses the external symbol. The word itself contains the next linkage, unless it is equal to zero, in which case it is the last link.

Table 9-4. Entry Point and External Reference Tables Format



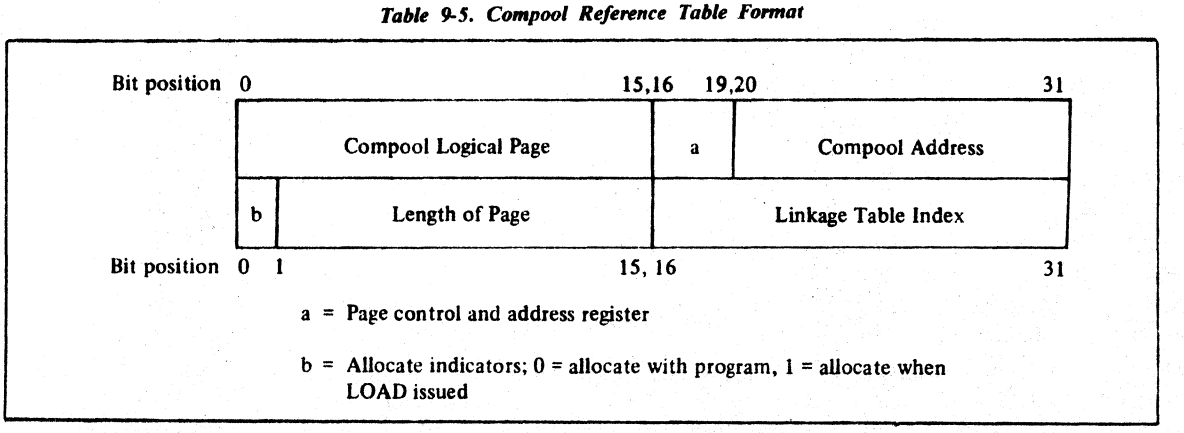
20079-26

c. Program-Access Table Card (Type 5). This table lists the programs which are called or loaded by the object program. The format of the program-access table is shown in the following chart.

0 Bit Positions 31
Program Name (First 4 characters)
Program Name (Last 4 characters)

d. Compool Reference Table Card (Type 6). The Compool reference table contains the page numbers of Compool references along with the index into the Compool reference linkage table. This table, along with the linkage table (type 7 cards), is used to modify the references in the object program to relocated Compool symbols. If the Compool reference has not been relocated, there is no modification required at the time of object program execution. The format of the relocatable Compool reference table is shown in Table 9-5. If the allocate indicator is equal to allocate with program, the space for the Compool reference is allocated at the time of loading the program. If the allocate indicator is equal to allocate when LOAD issued, no allocation is made when the program is loaded. The Compool reference table also contains the operand 'D' field and the page displacement associated with fixed-location Compool references made by the object program. These page references are stored in the Compool-page table maintained by the operating system. When a level is assigned to the object program, either within the object deck or through a procedure call, the page control and address registers for that level are configured to enable the program to access the specified pages.

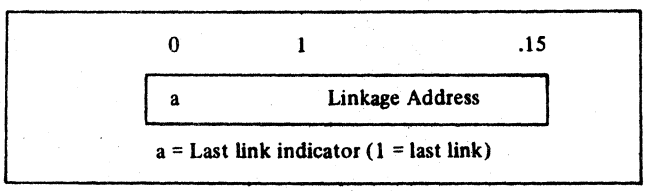
Table 9-5. Compool Reference Table Format



20079-27

e. Compool References Linkage Table Card (Type 7). The Compool reference linkage table contains the address (relative to the beginning of the program) of the reference to Compool-defined entities. The most significant bit of each link is a last link indicator, which if equal to one, indicates the last link in a reference chain. The format of the relocatable Compool reference linkage table is shown in table 9-6. When the Compool reference is loaded, its assigned load location is compared to its Compool location. If there is a difference, this difference is added to all of the references indicated by the linkage.

Table 9-6. Compool Reference Linkage Table Format



20079-28

f. End Card (Type 9). This card indicates the end of the object deck. It also specifies the point in the program to which control may be transferred after the program is loaded. The format of the end card is shown in table 9-7. If the entry point field is blank, the entry point is the first word of the program.

Table 9-7. End Card Format

Columns	Content
1	9
3-10	Assembly module name
12-15	Entry point (hex) or blank
17-18	Program level (decimal) or blank

20079-29

CHAPTER 10
ASSEMBLY LISTING

10-1. Introduction

The listing produced by the Assembler consists of the following parts:

a. A listing of the source cards.

b. A list of the types of errors encountered in the assembly.

c. A listing of the entry points, external references, Compool, page references, and program accessed.

d. A cross-reference listing.

e. Error diagnostics.

10-2. Source Card and AN/GYK-12 Machine Code Listing

This portion of the assembly listing contains the source cards (exactly as input) along with the machine code and addresses generated (if any). The content of this portion of the assembly listing is as specified in table 10-1.

10-3. Prologue

This portion of the assembly listing contains the contents of the object deck, except for the data and tag table cards (types 0 and 2, respectively).

10-4. External Symbol Dictionary

This listing of the type 1, 3, 4, and 9 cards (refer to table 10-2) is provided at the end of each assembly listing.

10-5. Cross-Reference and Set/Used Listing

The cross-reference and set/used listing is a listing of all references to symbols within the assembly module. Included is the symbol name, its value, and overall set/used indicator, the addresses within the module which reference the symbol, and a set or used indicator for each reference. This indicator appears immediately in front of each reference address and consists of an asterisk for set and a blank for used. SET is defined as a reference by an instruction which is capable of changing the value of a word (or bit) in core memory. USED is defined as a reference by an instruction which is not capable of changing core memory. The overall set/used indicator may be NONE, SET, USED, or BOTH. NONE means that the symbol is defined in the module but not referenced. BOTH means that a symbol is both set and used in the module. An example of a cross reference and set/used listing appears in Appendix A.

10-6. Assembly Error Indications

The assembly error indications are produced by the assembler as information to the user whenever the assembler detects incorrect source input.

a. Source Statement Assembly Error Messages. As a source program is assembled, it is analyzed for actual or potential errors in the use of the Assembler language. Errors which are detected are flagged in the assembly listing by the appearance of the word 'ERROR' followed by one or more of the letters E, I, L, M, O, P, and U, which denote the types of errors encountered in analyzing the source statement (refer to paragraph 10-6c below). This message appears in the line of assembly listing, immediately following the erroneous source statement.

b. Error and Error Symbol Count Message. At the end of each assembly listing, a count of the assembly errors encountered during the assembly and a count of the number of error symbols appearing in the listing are provided. The reason for providing both counts is to inform the programmer that there are errors for which a symbol is not present. (This occurs only when the same error symbol would have to appear more than once for the same source statement.)

c. Explanation of Errors Encountered During Assembly. Following the error counts is an explanation of the type of errors encountered during the assembly process. The explanations appear in the listing as specified in table 10-3.

Table 10-1. Assembly Listing

Content	Remarks
Error line	Refer to paragraph 10-6
Address	For machine instructions, GEN, and BSS assembler instructions the address is printed in hexadecimal
Function	For all machine instructions, the function code is printed in hexadecimal
H field	For all machine instructions except MIU and MIL, the H field is printed in hexadecimal
I (immediate) field	For MIU and MIL machine instructions, the immediate field is printed in hexadecimal
M field	For all machine instructions except MIU and MIL, the mode is printed in hexadecimal
S field	For all machine instructions except MIU and MIL, the index is printed in hexadecimal
D-A-W field	For all machine instructions, the D-A-W field is printed in hexadecimal
R field	For shift and format machine instructions, the tally, or destination register is printed in hexadecimal
T field	For shift and format instructions, the shift option code is printed in hexadecimal
K field	For shift and format instructions, the number of shifts is printed in hexadecimal
Generate instructions	A maximum of four half-words are printed per line. If more half-words are generated by one GEN statement, additional lines, each containing a maximum of four half-words, are printed if the DATA option has been specified on the OPT card. If less than four half-words are to be printed on a line, they are left justified within the allocated columns
EQU, BSS, ORG, and LOC instructions	The evaluation of the operand field is printed in hexadecimal under the D-A-W column
Statement number	The number of each source statement is printed in decimal
Macro-expansion statement indicator	A "plus" sign precedes the source statement for macro-instruction
Source statement	The source statement is printed in its entirety
Sequence error indicator	An asterisk following the source statement indicates a sequence error

20079-30

Table 10-2. External Symbol Dictionary Information

Card	Remarks
Type 1 (LOAD)	Contains the assembly module name, the location of the first relocatable instruction, the length of the module, and the length of local data used by the module
Type 3 (ENTRY)	Contain the entry names and their assembled locations
Type 4 (EXTRN)	Contain the external reference names, their assumed values, and the address of the last link in the chain of references
Type 9 (START)	Contain the module starting location (in hexadecimal) and its program level (in decimal)

20079-32

Table 10-3. Assembly Error Indications

Error code	Error code explanations
E (erroneous operand instructions) I (illegal character or name field entry)	Operand cannot be interpreted Illegal character encountered in operand field or a compool-defined symbol encountered in name field
L (machine field limit surpassed)	Limit error. A subfheid has exceeded its size limitation
M (multiple name field symbol definition)	A symbol appears in the name field of more than one source statement
O (undefined operation code)	An operation code was encountered which was neither a machine or extended mnemonic; nor a macro-instruction
P (possible error caused by forced offset) U (undefined symbol in operand field)	Possible error caused by forced offset Symbol encountered in operand field which has not been defined in name field or COMPOOL, and which has not been declared external

20079-33