IBM

PL/I Programming

I0103

Textbook 2

Independent Study Program

First Edition (July 1980)

Contents

Topic **11**

```
                                 I         S                               P
                                                                             D
                           A                          A
         E                           D                           T
               Y                                P                            Y          I
            D                        U                        E     T
                  M          D                            N                    M  D
             O                          G                               O          P
  U    P           D     E       U    P                 E          P         D
               I                     R                               A           T
             I           N    T     Y            I           N     T   Y           I        DE
  OG               P           T          OG    M        E          T        OG   M      P       T      D
        E          N       TU         O           E           ST         R             D          N      UD
      E      D               Y     OG          E     D           D     RO            D        N         ST        O
  D             NT        DY    R     AM      D           NT        D    R     AM      D           NT          P O
 ND         EN         D   P      R  M   ND          ENT        D   P      R  M  IND         ENT        D     RO      M
   E      N       U       P       A     IN       E       N     TU      P      R       IN       E       N       U      R        IN
  NDE      ST             GR            EP  ND                       RA                          U                     E
 D   T  STU      PR              D    ND        TU     PR    R         D     ND          TU  Y       O          ND
      T          R  G        D    EN  E       T           R  G          D   EN          TU          R    R  M            END
  ST       P  O          I        PE  D   T  ST          P  O            N    PE  D    T  STU              A     ND      N
 U  Y     ROGR                     NT  S  UDY    ROGR                    NT  S  U  Y     ROG                  EN       T
 D    PROGRAM        E          N         UDY  PRO  RA          E        N       UD   PRO  RA     ND  PE            S
  PROG  AM   N  EPE  DE      STU      PR      R        N  E  END    T  ST        PR  G  AM    N  EPE         T     T  D
 R  GR     INDEPEN  EN     S      Y  PR  GR      I  DEPE     ENT  ST  DY  PR  GR       INDEP    DENT  S      U
 GRAM  IN    P  ND  N     S     D     PR  GRAM  I       P  ND  NT  S     D     PRO  R  M  I      E  EN      T  STU      PR
    NDEP  NDEN     S  UDY  P  OGRAM     NDEP  NDENT    TUDY  PR  GRAM  IND  PEN  ENT    TUDY        O
 INDEPEN  ENT  S  UDY  PROG  AM  IND     ENDE  T  S  UDY     ROGRAM  I  DEPENDEN     STUDY  PROGRA
INDEPENDENT  STU  Y  PROGRAM  IN  EPE  DENT  ST  D    P  OGRAM  INDE  ENDENT  STUDY  PROGRAM
DEPENDENT  STUDY    ROGRAM  INDEPENDE  T  STUDY  PR  GRAM  INDEPENDENT  STUDY  PROGRAM  IN
PENDENT  STUDY  PROGRAM  INDEPENDENT  STUDY  PROGRAM  INDEPENDENT  STUDY  PROGRAM  INDE
NDENT  STUDY  PROGRAM  INDEPENDENT  STUDY  PROGRAM  INDEPENDENT  STUDY  PROGRAM  INDEPE
ENT  STUDY  PROGRAM  INDEPENDENT  STUDY  PROGRAM  INDEPENDENT  STUDY  PROGRAM  INDEPEND
T  STUDY  PROGRAM  INDEPENDENT  STUDY  PROGRAM  INDEPENDENT  STUDY  PROGRAM  INDEPENDEN
STUDY  PROGRAM  INDEPENDENT  STUDY  PROGRAM  INDEPENDENT  STUDY  PROGRAM  INDEPENDENT
UDY  PROGRAM  INDEPENDENT  STUDY  PROGRAM  INDEPENDENT  STUDY  PROGRAM  INDEPENDENT  ST
Y  PROGRAM  INDEPENDENT  STUDY  PROGRAM  INDEPENDENT  STUDY  PROGRAM  INDEPENDENT  STUD
PROGRAM  INDEPENDENT  STUDY  PROGRAM  INDEPENDENT  STUDY  PROGRAM  INDEPENDENT  STUDY
OGRAM  INDEPENDENT  STUDY  PROGRAM  INDEPENDENT  STUDY  PROGRAM  INDEPENDENT  STUDY  PR
RAM  INDEPENDENT  STUDY  PROGRAM  INDEPENDENT  STUDY  PROGRAM  INDEPENDENT  STUDY  PROG
M  INDEPENDENT  STUDY  PROGRAM  INDEPENDENT  STUDY  PROGRAM  INDEPENDENT  STUDY  PROGRA
```

Topic 11

Regional Organization

In this topic you will learn about the three Regional organizations in PL/I: REGIONAL (1), REGIONAL (2) (OS/VS only) and REGIONAL (3). You will learn about the physical layout of each organization on disk, the reasons for using each of the three types and how to write programs which use Regional data sets. If you are already familiar with Regional organization you can leave out the associated sections within this topic and read the sections relating to PL/I.

Objectives

At the end of this topic you should be able to:

- understand the differences between the three Regional organizations
- state the advantages of each type of Regional organization and when each type would be used
- write PL/I programs which use Regional data sets
- understand the circumstances in which the KEY condition will be raised.

Introduction

Regional organization is used when **direct** access to the data set is more important than accessing the data set in a logical sequential order. Records are loaded onto the data set by being directed to a particular 'slot' on the data set, relative to the beginning. A 'slot' or region can either be the size of one record (REGIONAL (1) and (2)) or the size of one track of the disk (REGIONAL (3)). Regions always start at Region 0.

The records on the data set will be retrieved directly using some field associated with, and usually located within, each record. This field will be called a 'control field' in this topic and might be a personnel number, a part number for spare parts, a branch number etc. The region number for a particular record could be the same as this control field, in which case the maximum region number, and hence the size of the data set, would be determined by the maximum control number. However, it is more usual **first** to allocate a suitable space for the data set - this decides how many regions there will be - and then to manipulate the record control fields so that they fall within this range of regions. In order to subsequently retrieve records from the data set, the same manipulation of the control field must be performed and the resulting region searched.

Let's see how this is done for each of the Regional organizations.

REGIONAL(1)

Suppose we have ten records to load onto a REGIONAL(1) data set. The control numbers are 101, 145, 132, 169, 178, 109, 188, 155, 111 and 190. We could simply take these control numbers as region numbers and create the data set. This would be very wasteful of disk space because the data set must start at region 0 and all regions up to the maximum region number must be represented. So we must find a way of compressing the data set size. One commonly used method is to allocate a number of regions to the data set (slightly more than there are records), divide the control number by the number of allocated regions and take the remainder. (Other methods can be used to generate the region number - see Exercise 6 at the end of this topic - but the 'remainder' method is the most commonly used).

Let's allocate 16 regions to our data set (region 0 to region 15) and suppose that there will be 4 records per track. If we divided each number by 16, then by definition, every remainder thus generated would fall within the range 0 through 15. For our particular set of records we would generate regions 5, 1, 4, 9, 2, 13, 12, 11, 15, 14 respectively.

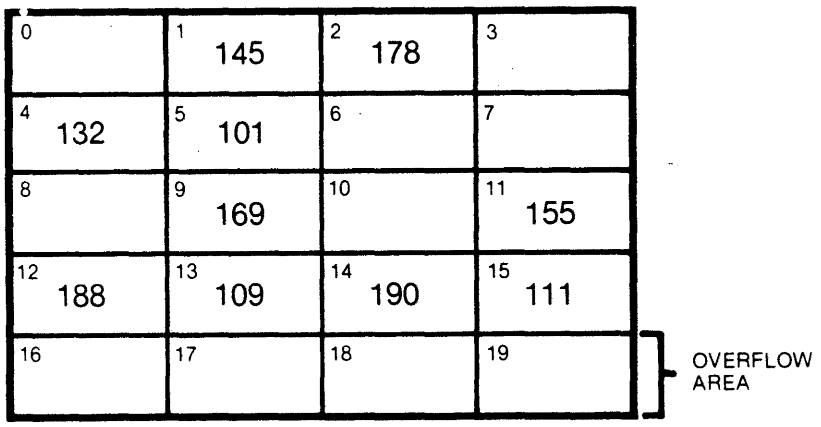
So we would write record 101 in slot (region) 5, record 145 in slot (region) 1 etc.

0	1 145	2 178	3
4 132	5 101	6	7
8	9 169	10	11 155
12 188	13 109	14 190	15 111

You can see that it would be easy to find two or more records wanting to go in the same region. For example, suppose we had to write record 117 onto the data set; that would also want to go into region 5.

Any record trying to go to a region in which a record already resides, is called a 'synonym'. (Increasing the number of regions in the data set might reduce the number of synonyms but it would increase the size of the data set; a compromise has to be reached).

Our program must be capable of recognizing that the region is already in use, and we must therefore arrange to write the 'synonym' in an 'overflow' area. So in this particular case, we might well have reserved an extra track full (i.e. an extra four regions, numbered 16, 17, 18, 19) into which we could write records which 'overflowed'.



Dummy Records in REGIONAL(1)

It is the programmer's responsibility to recognize 'synonyms' and to direct such records to an overflow area. In order to help him/her do this every 'unused' region in the REGIONAL(1) data set is filled with a 'dummy' record by the Regional routines, when the data set is created. A 'dummy' record contains (8)'1' B in the first byte.

How does this help the programmer? Well, the program could read from a region before writing to it. And if the record read was a dummy record? Then the program could write to the region concerned knowing that the region was currently unused. And if the record read was not a dummy record? Then the program would have to write the record in an overflow region - but which overflow region (16, 17, 18 or 19)? A suitable choice would be the first overflow region which was currently unused. This would be found by reading from each overflow region in turn until a dummy record was retrieved that would indicate the first unused overflow region.

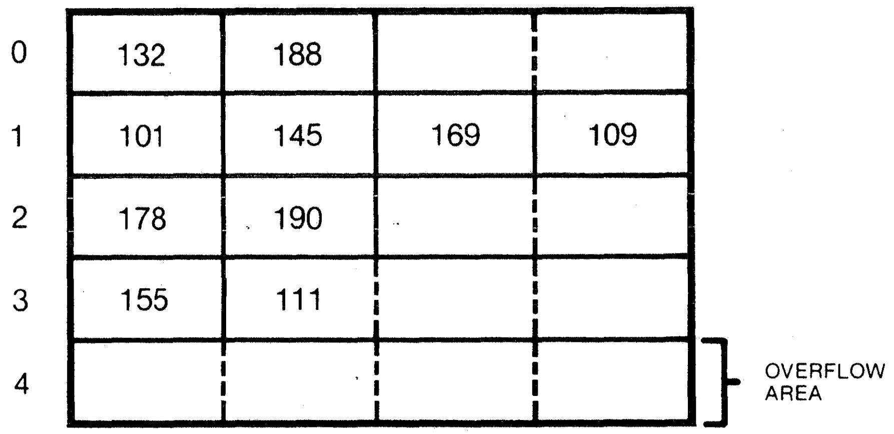
So far we have been talking about creating the REGIONAL(1) data set. The same method of converting the control number to a region number must be used on subsequent retrieval. If the record read does not have the control number you require, then perhaps the record required was a 'synonym'. Therefore search the overflow regions in turn until you find the record with the control number you want. (There is always the possibility that the record might not be there at all).

REGIONAL(3)

REGIONAL(3) data sets are similar to Regional(1) data sets except that a Region is a track - i.e. it can hold more than a single record.

Let's take the same records as before and load them this time onto a REGIONAL(3) data set - to which we have allocated 5 tracks numbered 0 through 4. We intend to use the final track, i.e. track 4, for overflow records, therefore records must initially only be directed to tracks 0 through 3. We can achieve this by dividing the control numbers (101, 145, 132, 169, 178, 109, 188, 155, 111 and 190) by 4 and taking the remainder. This will produce regions 1, 1, 0, 1, 2, 1, 0, 3, 3 and 2 respectively.

Records are written in the next available space in the region specified. Several records can exist in the same region,. therefore each record must have a **recorded key** (immediately preceding the data) so that the REGIONAL(3) routines can distinguish between different records in the same region. This recorded key is usually the control number.



OS/VS Considerations

If we subsequently wished to add a further record to **track 1,** the OS/VS PL/I routines are in fact capable of continuing to search along **track 2** until they find a vacant space. Indeed the whole data set could be searched if necessary until a vacant slot was found. However, this could result in time being wasted, not only to load the file, but in subsequent retrieval. So it is possible to limit the number of tracks to be searched via a Job Control subparameter on the DD card for the data set.

This subparameter is called the LIMCT subparameter and has the form LIMCT=n, where n is the number of tracks to be searched. If LIMCT is omitted the search will continue to the end of the data set and then from the beginning to the original track searched.

The Use of the KEY Condition

If no space is found when creating the REGIONAL(3) data set or a record is not found on retrieval, then the KEY condition is raised. The KEY condition is one of several 'on-conditions' which PL/I recognizes as abnormal situations. The default action for most on-conditions is to print a message and terminate the program but this default action can be overridden by the programmer (see Topic 19). In particular the programmer can investigate the reason for raising the KEY condition and take appropriate action.

For example if you are reading the REGIONAL(3) data set and the KEY condition is raised because the record cannot be found, then read from the overflow region(s). If the KEY condition is still raised because the record cannot be found in the overflow region(s) then the record does not exist on the data set. (You might decide to print an appropriate message under these circumstances).

The KEY condition is equally useful on creation. If the KEY condition were raised because there was no room to write a record in the region specified, then you should arrange to write the record in an overflow region.

The KEY condition is not raised under the same circumstances for REGIONAL(1) data sets.

For details about PL/I on-conditions (of which the KEY condition is one) and how to 'handle' them, see Topic 19. Even at this stage, however, you should appreciate that the KEY condition makes programming for REGIONAL(3) data sets much easier than programming for REGIONAL(1) data sets.

Comparison of REGIONAL(1) and REGIONAL(3) Data Sets

REGIONAL(1) organization provides the fastest means of directly retrieving a record from a data set (provided that there are not many overflows). REGIONAL(1) records are F- format with no recorded keys - keys are not needed, because each region corresponds to a single record.

REGIONAL(3) organization supports the following record formats: F, U, V (OS/VS only) and VS (OS/VS only). Each record has a recorded key (because each region will probably contain more than one record). REGIONAL(3) is not quite as fast as REGIONAL(1) for direct processing (think about the reason for this) but it is more convenient to program because of the KEY condition).

Programs which create REGIONAL(1) data sets are device independent in that the REGIONAL(1) data set will always be the same size, regardless of the device. Programs which create REGIONAL(3) data sets are not device independent in this manner, because the number of records per track (i.e. region) will vary from device to device. For example, if we have to load 100 records on to a device which can hold 20 records per track then we might allocate 6 tracks (regions 0 through 5) and divide the control number by 5 taking the remainder as the REGIONAL(3) region number; region 5 would be used as an overflow region. In order to load the same records on to a device which only holds 10 records per track we might allocate 12 tracks (regions 0 through 11) and device the control number by 10 taking the remainder as the REGIONAL(3) region number; regions 10 and 11 would be used as overflow regions. In other words, the program would have to be different for different devices. REGIONAL(1) data sets do not have this disadvantage.

REGIONAL(2) organization (OS/VS only) combines the device independence of REGIONAL(1) with the ease of coding of REGIONAL(3).

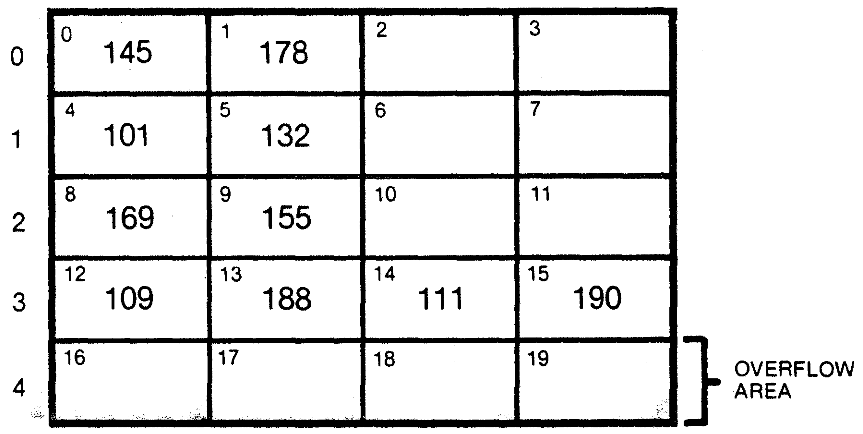
REGIONAL(2) - OS/VS Only

REGIONAL(2) data sets are a combination of REGIONAL(1) and REGIONAL(3) data sets. The region that you calculate **in the program** is a relative **record** slot (as in REGIONAL(1)) - but **at execution time,** when the type of device being used is known, the PL/I REGIONAL(2) routines convert this relative **record** slot to a relative **track** slot (as in REGIONAL(3)). The record goes in the next available space in the relative track.

Let's take the same set of records again and create the REGIONAL(2) data set on a device which will hold four records per track. We have allocated five tracks - 4 data tracks and one overflow. The second column shows the relative record slot calculated in the program by dividing by 16 and taking the remainder. The third column shown the relative track number at execution time.

Record number	**Remainder after dividing by 16**	**Eventual relative track number**
101	**5**	**1**
145	**1**	**0**
132	**4**	**1**
169	**9**	**2**
178	**2**	**0**
109	**13**	**3**
188	**12**	**3**
155	**11**	**2**
111	**15**	**3**
190	**14**	**3**

This is what the data set would look like:



The same rules apply to REGIONAL(2) data sets as to REGIONAL(3): if the original region (track) is full, then the next one, and so on, will be searched until a space is found. Once again, this search can be limited by the LIMCT subparameter on the Job Control DD card.

Keys For REGIONAL(1)

The following is an example of a **direct** read statement for a Regional(1) data set:

```
  READ FILE(REG1) INTO(AREA) KEY('ØØØØØ619');
```

The expression in brackets after the KEY option, is called the **source key** the key used within the program. Regional(1) source keys specify a region number. So, the above statement will read from region 619 or the REG1 data set. The region number is assumed to be eight **characters** long. If the source key is longer than eight characters, the right most eight characters are taken. If the source key is shorter than eight characters, blanks (interpreted as zeroes) are assumed on the left. If the expression after the KEY option is not in character format, it will be converted to character format before being used.

Keys For REGIONAL(2) and REGIONAL(3)

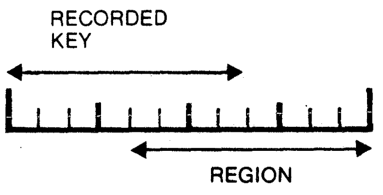
The following is an example of a direct read statement for a REGIONAL(2) or REGIONAL(3) data set:

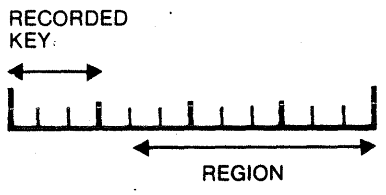
```
  READ FILE(REG2OR3) INTO(AREA) KEY('999912345999');
```

The expression in brackets after the KEY option, is called the **source** key - the key used within the program. Regional(2) and Regional(3) source keys specify a region number (the last eight characters) and a **recorded key.**

The recorded key starts at the left of the source key; its length is specified in the KEY-LENGTH environment option of the file declaration (OS/VS and DOS/VS) or in the DD KEYLEN subparameter portion of each record on the data set - it is **external** to the data portion - and is used to distinguish different records in the same region. (Recorded keys are not necessary in REGIONAL(1)).

The region number is 12345999 (the rightmost eight characters). A KEYLENGTH specification of 4 would indicate that the recorded key was 9999 (the leftmost 4 characters). A KEYLENGTH specification of 12 gives a recorded key of 999912345999. The recorded key and the region number can overlap within the source key or they can be separated by 'redundant bytes' within the source key:





The recorded key is normally the control number and so the source key would be a concatenation of the control number with an 8-byte region number (declared as PIC'(8)9' or CHAR(8)).

For example:

```
DCL REG23 FILE RECORD DIRECT INPUT
               ENV(.....KEYLENGTH(6)....);
DCL REG       PIC'(8)9';
DCL CTLNO     PIC'(6)9';

READ FILE(REG23) INTO(AREA) KEY(CTLNO||REG);
```

If the region specification is not in character format it will be converted to character and the rightmost eight bytes used as the region number.

More information about source keys, recorded keys and Regional organization is contained in the Language Reference Manual and the Programmer's Guide under the appropriate headings.

Declaring a REGIONAL FILE - DOS/VS

An example of a file declaration for the Regional organization is shown below:

```
      DCL REGFIL FILE DIRECT KEYED
                 ENV(KEYLENGTH(n) EXTENTNUMBER(n)
                     REGIONAL(n)...........);
```

Note the following points:

KEYED	must be specified (or implied) if you are using keys when processing or creating the data set, i.e. the options KEY, KEYTO (see later) and KEYFROM (see later) attached to the I/O statement.
DIRECT	specify if processing directly; DIRECT implies KEYED but not vice versa.

The following are options of the ENVIRONMENT attribute:

KEYLENGTH(n)	specifies the length of the recorded key for Regional(3); not valid for Regional(1).
REGIONAL(1) or REGIONAL(3)	
EXTENTNUMBER(n)	this specifies the number of extents of the data set; the default is 1.

Declaring a REGIONAL FILE - OS/VS

An example of a file declaration for the Regional organization is shown below:

```
         DCL REGFIL FILE DIRECT KEYED
                    ENV(KEYLENGTH(n) REGIONAL(n)....);
```

KEYED	must be specified (or implied) if you are using keys when processing or creating the data set, i.e. the options KEY, KEYTO (see later) and KEYFROM (see later) attached to the I/O statement.
DIRECT	specify if processing directly; DIRECT implies KEYED but no vice versa.

The following are options of the ENVIRONMENT attribute:

KEYLENGTH(n)	specifies the length of the recorded key for REGIONAL(2) and REGIONAL(3); not valid for REGIONAL(1). This value can be specified in the KEYLEN subparameter of the DD statement:

```
//REGFIL DD DCB=(KEYLEN=9)
```

REGIONAL(1), REGIONAL(2) or REGIONAL(3)

TRKOFL	applies to REGIONAL(1) and REGIONAL(2) only. This feature allows the record size to exceed the size of a track. You must include this function when the operating system is generated and you must have a special feature on the DASD control unit. This option enables better space utilization on disk.

Dummy Records

Dummy records in REGIONAL(1) have (8)'1'B in the first byte of the **data.** In OS/VS, REGIONAL(2) and REGIONAL(3) (fixed format) dummy records consist of a 'dummy' **recorded key** with (8)'1'B in the first byte, and the first byte of the **data** contains the sequence number of the record on the track. There are no dummy records in DOS/VS REGIONAL(3) organization.

Creation of REGIONAL Data Sets

Regional data sets can be created either sequentially or directly.

In **sequential creation** the records must be written to the data set in ascending region number. REGIONAL(3) organization obviously allows consecutive records to have the same region number, but not REGIONAL(1) or REGIONAL(2). In REGIONAL(1) and REGIONAL(2) sequential creation, any region omitted from the sequence is filled in with a 'dummy' record. Any error in the region number sequence raises the KEY condition.

In **direct creation** records do not have to be written to the data set in a specific order. In REGIONAL(1) and REGIONAL(2) the whole of the data set is preformatted with 'dummy' records when the data set is opened for direct creation (DIRECT OUTPUT). The same preformatting occurs with OS/VS REGIONAL(3) fixed-length-record data sets but not with DOS/VS REGIONAL(3) data sets. Having opened the Regional data sets for DIRECT OUTPUT, records can be written to the data set in any order and they will be placed in their region or in the next available place in that region depending on the type of Regional organization.

PL/I Statements for Processing REGIONAL Data Sets

Let's look at the PL/I statements which processes Regional data sets both sequentially and directly. There is a table in the Language Reference Manual, in the chapter on Record Oriented Transmission, which summarizes the PL/I statements which can be used. Locate this table now. The basic format of the I/O statements is similar to CONSECUTIVE organization, with the addition of several options. The table is divided into three columns: how the associated file can be declared, valid I/O statements to use with the file declared in this way and other options that can also be used. First let's look at the new options.

KEYFROM Option

This is used to specify the **source** key to be used when **creating** the data set. In REGIONAL(1) the source key will be a region number.

In REGIONAL(2) and REGIONAL(3) the source key will normally be a control number concatenated with a region number.

For example:

```
DCL REG       PIC'(8)9';
DCL CTLNO     CHAR(6);

WRITE FILE(REG1) FROM(REC) KEYFROM(REG);
WRITE FILE(REG3) FROM(REC) KEYFROM(CTLNO||REG);
```

KEY Option

This is used to specify the **source** key when **reading** the data set.

KEYTO Option

For REGIONAL(1) data sets, this option obtains the region number as an eight byte character string and brings it into storage. If the variable specified in the KEYTO option has a length other than eight characters, then truncation or padding will occur on the **left.**

Example:

```
DCL REGNO     CHAR(5);

READ FILE(REG1) INTO(AREA) KEYTO(REGNO);
```

In this example the first record read will place '00000' in REGNO. Subsequent reads will increment this by 1. It would be more usual (and wiser) to ensure that the KEYTO variable is declared with the CHAR(8) attribute.

For REGIONAL(2) and REGIONAL(3) data sets, the KEYTO option causes the **recorded key** to be assigned to the specified variable. If this variable has the wrong length then truncation or padding with blanks will occur on the **right** (as in normal character assignment).

Example:

```
     DCL RECKEY CHAR(6);
     READ FILE(REG23) INTO(AREA) KEYTO(RECKEY);
```

Suppose the data set REG23 has a keylength of 9 and a record with recorded key '12345ABCD' is read, then '12345A' will be placed in RECKEY. It would be more usual to ensure that the KEYTO variable has the same length as the recorded key length of the data set.

Sequential Processing of REGIONAL Data Sets

Both Move mode and Locate mode processing can be used. Records are retrieved in ascending region number which will not usually be the same as ascending control number because of the way in which region numbers are derived.

REGIONAL(1) - OS/VS and DOS/VS

Dummy records are retrieved and must be ignored within the program.

REGIONAL(2) and REGIONAL(3) - OS/VS Only

Dummy records are not retrieved.

REGIONAL(3) - DOS/VS Only

There are no dummy records.

Direct Processing of REGIONAL Data Sets

Move mode only can be used.

Regional data sets can be created directly (see section on Creating Regional Data Sets).

```
DCL REGOUT FILE RECORD DIRECT OUTPUT
                ENV(REGIONAL(n)......);

WRITE FILE(REGOUT) FROM(AREA) KEYFROM(KEY);
```

n order to read the data set directly declare the file DIRECT INPUT.

```
DCL REGIN FILE RECORD DIRECT INPUT
               ENV(REGIONAL(n).......);

READ FILE(REGIN)INTO(AREA) KEY(KEYFLD);
```

When the file is declared DIRECT UPDATE existing records can be updated directly, using the REWRITE statement, or new records can be added using the WRITE statement.

```
        DCL REGUP FILE RECORD DIRECT UPDATE
                      ENV(REGIONAL(n).......);

        REWRITE FILE(REGUP) FROM(AREA) KEY(OLDKEY);

        WRITE   FILE(REGUP) FROM(AREA) KEYFROM(NEWKEY);
```

Deleting Records - OS/VS Only

In OS/VS it is also possible to logically delete records by converting them to 'dummy' records using the DELETE statement with the file declared as DIRECT UPDATE.

```
        DCL REGFIL FILE DIRECT UPDATE ENV(REGIONAL(n));

        DELETE FILE(REGFIL) KEY(DELKEY);
```

If you try and retrieve a deleted (dummy) record **directly** in REGIONAL(2) or REGIONAL(3) then the KEY condition will be raised. Remember, dummy records in REGIONAL(2) and REGIONAL(3) have (8)'1'B in the first byte of the key, so the key specified in the READ statement will not be found. During **sequential** processing, deleted records are ignored.

Key Condition

This is one of many PL/I on-conditions which are raised when PL/I recognizes an abnormal situation (see topic 19 - but not now). Reasons for raising the KEY condition are shown below:

REGIONAL(1)

- A region is specified which is outside the data set limits
- An error in the region sequence on sequential creation
- A duplicate region during sequential creation.

REGIONAL(2) and REGIONAL(3)

- A region is specified which is outside the data set limits
- An error in the region sequence during sequential creation
- Specified keys cannot be found
- No room to add the record.

Event Option

The EVENT option can be attached to I/O statements for Regional files which are UNBUFFERED SEQUENTIAL or DIRECT (which is automatically unbuffered). This option causes control to be passed immediately to the next statement in the program without waiting for the completion of the associated I/O activity. The main use of the EVENT option is to overlap processing with I/O when processing directly. This overlap is automatic for **sequential** processing if more than one buffer is used but there are no buffers when **directly** processing Regional data sets. The following coding is one way in which this overlap of processing and I/O might be achieved for DIRECT files.

```
DCL REGFIL FILE DIRECT INPUT ENV(REGIONAL(n)........);
DO WHILE(SWITCH);
     READ FILE(REGFIL) INTO(AREA1) KEY(KEY1);
     READ FILE(REGFIL) INTO(AREA2) KEY(KEY2) EVENT(X);
     /* process record in AREA1 */
     WAIT(X); /* wait for record to be read into AREA2 */
     /* process record in AREA2 */
END;
```

The first READ does not use the EVENT option and so control does **not** pass to the second READ until the first READ has taken place. So, while the second READ is taking place, we can process the record read into AREA1. When we wish to process the record in AREA2 we must issue a WAIT statement. In our example X is an EVENT VARIABLE which is contextually declared as such. An event variable links the WAIT statement to a particular I/O event.

EXERCISES

1. With references to the Regional data sets relevant to your system.
   (a) which provides the fastest means of direct retrieval?
   (b) which is the most difficult to program?
   (c) are there any other advantages/disadvantages of the different Regional data set organizations?
2. 'ABCDEFG1234' is the **source key** of a record which is to be written to a REGIONAL(3) data set. A keylength of 8 has been specified. What will be the **recorded key?**
3. Under what conditions will Regional dummy records be ignored? Specify any differences between the different Regional organizations.
4. (OS/VS only) 'ABCD00123456' is specified as the source key of a Regional(2) record. Where will the record be stored on the data set?
5. Declare a suitable file and write the necessary statements to replace a record in region 100 of a REGIONAL(1) data set with a record containing all blanks. Assume a record size of 80.
6. A REGIONAL(3) data set is to be created. It will contain approximately 3500 records with part numbers ranging from 1 to 10,000. The region number is to be calculated by dividing the part number by 3 and taking the integer result as the region. The records to be loaded on to the data set are read in from a card file which is not in any particular order. Write the program to create the Regional(3) data set. The structure of the records is shown below. The recorded key is to be the part number (PARTN). You may omit coding to deal with overflows.

```
     DCL 1 INPT,
           2 PARTN     PIC'(5)9',
           2 REST      CHAR(75);
```

7. You are creating a REGIONAL(1) data set directly. The record size is 80 bytes. Write the necessary declarations and statements to load records on to the data set. Assume that the records to be written are contained in an input file and make any other necessary assumptions.

Answers

1. (a) REGIONAL(1). The access method goes straight to a record (provided there are not any overflows) rather than to a track and then to a record as in Regional(3) (and Regional(2) in OS/VS).

   (b) REGIONAL(1). Regional(3) (and Regional(2) in OS/VS) are easier: the KEY condition can be used on **input** to detect record not found (therefore read from overflow region(s)) and on **output** to detect no room to write in the region specified (therefore write to the overflow region(s)).

   (c) Programs which create REGIONAL(1) data sets are device independent. Regional(3) data sets will usually save disk space because records go in the next available space within a region. Regional(2) (OS/VS only) combines the above two advantages.

2. 'ABCDEFG1'.

3. REGIONAL(1) dummy records are always retrieved. It is up to the programmer to ignore them within the program. REGIONAL(3) and REGIONAL(2) dummy records are ignored on sequential processing. If you attempt to retrieve directly a deleted (i.e. dummy) record from a REGIONAL(2) or REGIONAL(3) data set, using the original key of the record, then the KEY condition is raised.

4. (OS/VS only). In the first available space found, starting from the beginning of the track in which region 123456 is located (subject to the LIMCT restrictions).

5.

```
DCL REG1 FILE DIRECT UPDATE KEYED ENV(REGIONAL(n).....);
DCL DATA CHAR(8Ø);
DATA=' ';
REWRITE FILE(REG1) FROM(DATA) KEYFROM('ØØØØØ1ØØ');
```

OS/VS solution

6. Notice how the JCL is specified.

```
REGPROG: PROC OPTIONS(MAIN);
         DCL 1 INPT,
               2 PARTN PIC'(5)9',
               2 REST  CHAR(75);
         DCL REG3 FILE RECORD DIRECT KEYED OUTPUT
                       ENV(REGIONAL(3) KEYLENGTH(5));
         DCL REGION PIC'(8)9';
         DCL CARDIN FILE RECORD;
         DCL EOF  BIT(1) INIT ('Ø'B);
         ON ENDFILE(CARDIN) EOF='1'B;

         READ FILE(CARDIN) INTO(INPT);
         DO WHILE(¬EOF);
             REGION=PARTN/3;
             WRITE FILE(REG3) FROM(INPT) KEYFROM(PARTN||REGION);
             READ FILE(CARDIN) INTO(INPT);
         END;
END;
```

```
//GO.REG3 DD DSN=S42ØIFLE,UNIT=333Ø,SPACE=(CYL,11),VOL=SER=3ØWORK,
//            DISP=(NEW,DELETE),DCB=(RECFM=F,BLKSIZE=8Ø,DSORG=DA)
```

The KEYLENGTH could have been specified in the DCB parameter of the DD statement as KEYLEN=5.

DOS/VS Solution

6. Notice how the JCL is specified.

```
REGPROG:PROC OPTIONS (MAIN);
        DCL 1 INPT,
              2 PARTN     PIC'(5)9',
              2 REST      CHAR(75);

        DCL REG3 FILE RECORD DIRECT KEYED OUTPUT
                          ENV(MEDIUM(SYSØ1Ø,333Ø) F RECSIZE(8Ø)
                              KEYLENGTH(5) EXTENTNUMBER(1) REGIONAL(3));
        DCL CARDIN FILE RECORD INPUT
                           ENV(MEDIUM(SYSIPT,2540) RECSIZE(8Ø) F );

        DCL REGION PIC'(8)9';
        DCL EOF BIT(1) INIT('Ø'B);
        ON ENDFILE(CARDIN) EOF='1'B;

        READ FILE(CARDIN) INTO(INPT);
        DO WHILE(¬EOF);
             REGION=PARTN/3;
             WRITE FILE(REG3) FROM(INPT) KEYFROM(PARTN||REGION);
             READ  FILE(CARDIN) INTO(INPT);
        END;
END;
```

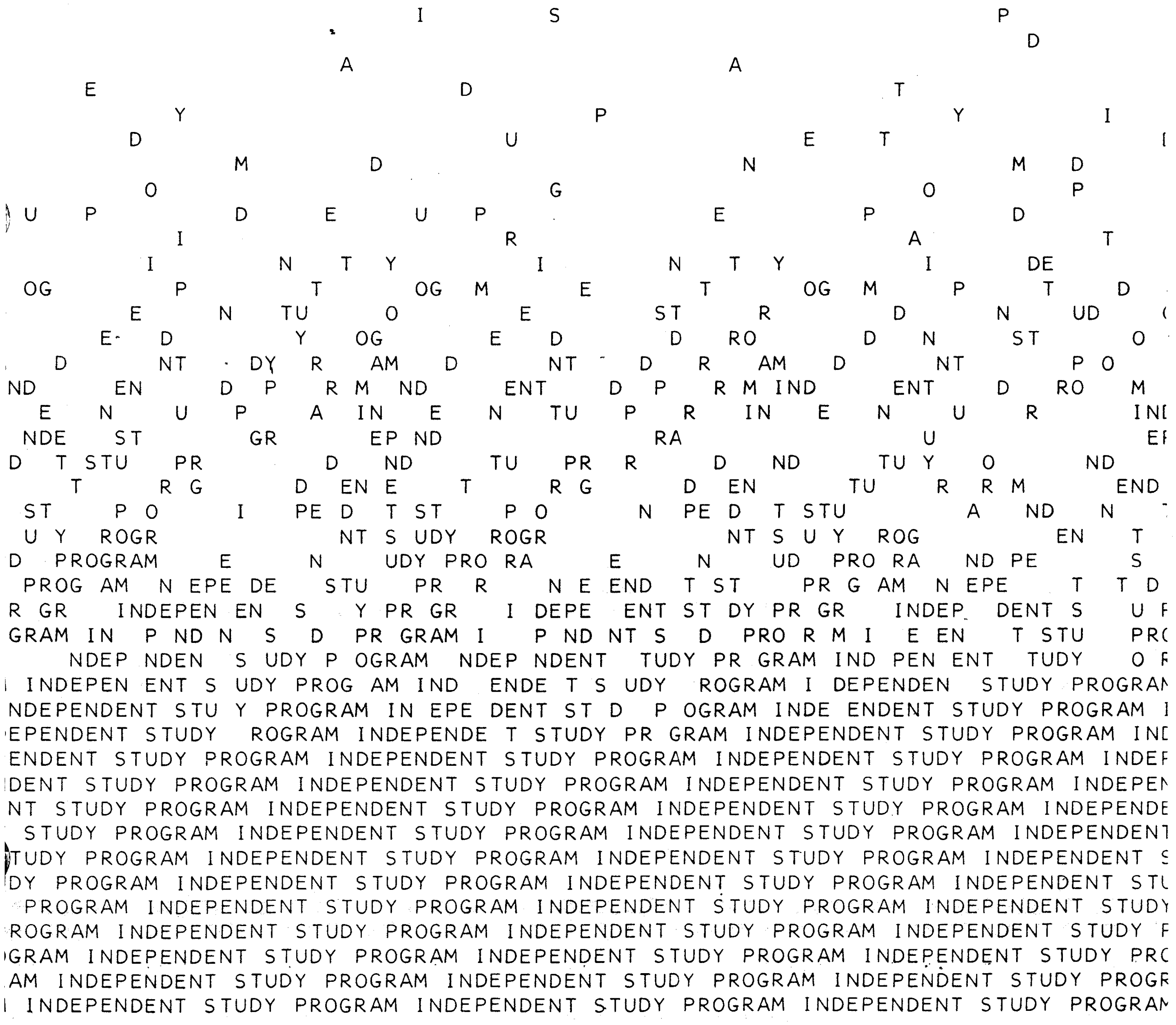
```
// ASSGN SYSØ1Ø,333Ø,VOL=3ØWORK,SHR
// DLBL REG3,'S42Ø1FLE',Ø,DA
// EXTENT SYSØ1Ø,3ØWORK,1,1,19Ø,2ØØ
```

7.

```
DCL INFILE FILE RECORD INPUT SEQUENTIAL ENV(......);
DCL REG1 FILE DIRECT UPDATE ENV(REGIONAL(1)....);
DCL REG1OUT FILE DIRECT OUTPUT ENV(REGIONAL(1)....);
DCL 1 STR,
      2 FIRST    BIT(8),
      2 REST     CHAR(79);
DCL AREA CHAR(8Ø);

OPEN FILE(INFILE),FILE(REG1OUT),FILE(REG1) TITLE('REG1OUT');

READ FILE(INFILE) INTO(AREA);
DO WHILE(/* NOT END OF INPUT FILE */);
        /* WORK OUT REGION NUMBER*/
        READ FILE(REG1) INTO(STR) KEY(/* REGION NUMBER */);
        IF FIRST=(8)'1'B  /* DUMMY RECORD ? */
        THEN WRITE FILE(REG1) FROM(AREA)
                              KEYFROM(/*REGION NUMBER */);
        ELSE /* READ FROM SUCCESSIVE OVERFLOW REGIONS */
             /* UNTIL YOU FIND AN 'EMPTY' ONE AND    */
             /* THEN WRITE RECORD TO THAT REGION     */
END /* DOWHILE */ ;
```

Notice how the TITLE option is used to associate two files with the one data set. This is done so that the data set can be preformatted with dummy records by opening the file REG1OUT. The same data set is subsequently processed using REG1.

Topic 12



Topic 12

Indexed Organization

In this topic you will learn about the organization of INDEXED data sets and the reasons for using them. In particular you will learn to declare INDEXED files and write PL/I statements to access them. If you are already familiar with indexed organization you can leave out the associated sections within this topic and read the sections relating to PL/I.

Objectives

At the end of this topic you should be able to:

- understand the reasons for using INDEXED organization
- write PL/I programs which process INDEXED data sets
- understand the circumstances in which the KEY condition will be raised
- describe the optimization options concerned with INDEXED data sets.

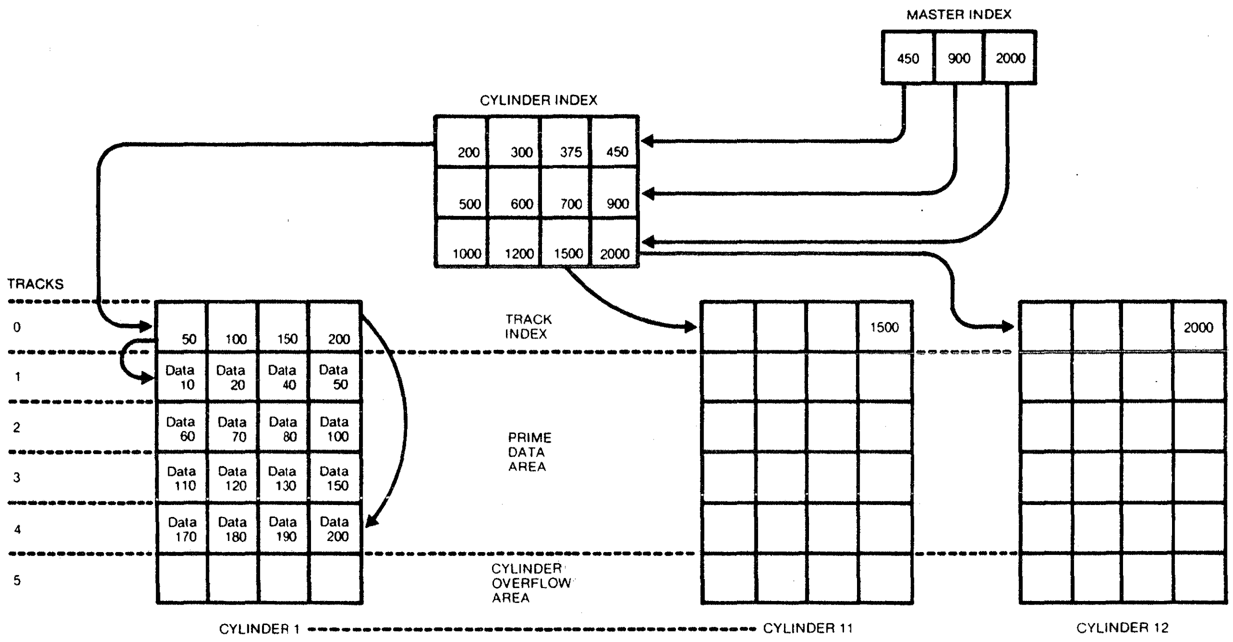
Introduction

INDEXED data sets are used when it is required to process data sets directly (using keys) and also sequentially in ascending key sequence. However INDEXED access routines are neither so fast at sequential processing as CONSECUTIVE routines, nor so fast at direct processing as REGIONAL routines.

Direct processing is achieved by associating each record with a **recorded key** (see later); the access routines then look up this key in indexes in order to locate the record.

Indexed Organization

INDEXED data sets can only be created sequentially. Records must be presorted into ascending key sequence and then written to the data set one after another. The following diagram illustrates an INDEXED data set of twelve cylinders. (For the sake of simplicity there are only 6 tracks per cylinder).



The various components of the data set are labeled in the diagram and are described below. (Note that the indexes are built automatically by the INDEXED routines on creation of the data set).

Prime Data Area

The Prime Data Area is the area of the data set to which records are originally written in ascending key sequence. In OS/VS this is specified in the SPACE parameter of the DD statement. In DOS/VS it is specified by an EXTENT statement.

Track Index

There is a Track Index for each cylinder of the data set. The Track Index for a particular cylinder contains an entry for each track on that cylinder. One of the items of information in each entry is the highest key on the associated track. The Track Index resides on the first track of the cylinder; any remaining space on the first track is used by the Prime Data Area. (The first track is track 0).

Cylinder Index

If the data set contains more than one cylinder a higher level index - the Cylinder Index - is created on output. This contains the highest key on each cylinder. In DOS/VS the Cylinder Index requires a separate EXTENT card with sequence number '1' and type code '4'.

Master Index

This is optional. It is an index to the Cylinder Index. Use it only if the Cylinder Index is quite large. It is specified in OS/VS by the subparameter OPTCD=M on the DD statement. In DOS/VS it is specified by the INDEXMULTIPLE option of the ENVIRONMENT attribute of the file declaration.

In DOS/VS a separate EXTENT statement is required for the Master Index with sequence number '0' and type number '4'.

In OS/VS the Cylinder Index and Master Index (if used) together constitute the Index Area; this Index Area can be specified either on a separate DD statement or the Index Area can be combined with the Prime Data Area on a single DD statement.

Cylinder Overflow Area

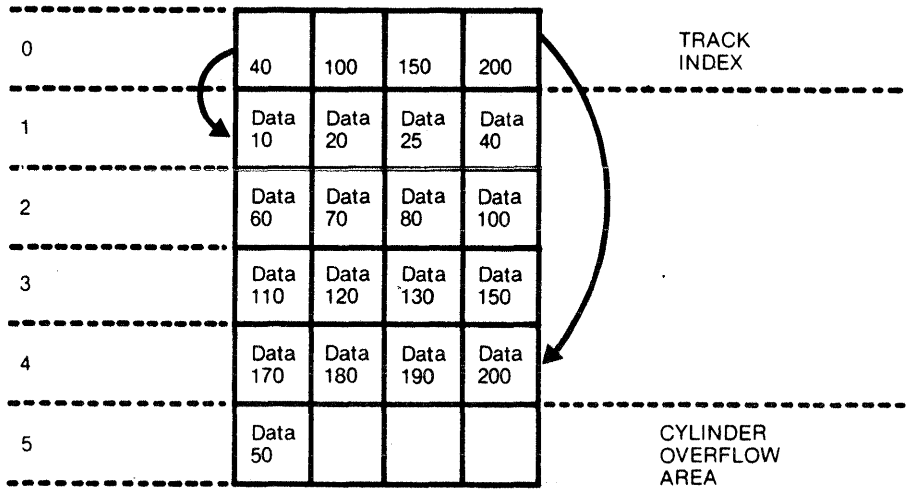
This is the number of tracks per cylinder of the data set which are set aside for overflows. In OS/VS this is specified by the subparameter OPTCD=Y on the DD statement; the subparameter CYLOFL=n is used to specify the number of tracks to be used. In DOS/VS the number of overflow tracks is specified by an option of the ENVIRONMENT attribute when declaring the file: OFLTRACKS(n). The Cylinder Overflow Area is sometimes called the Embedded Overflow Area.

Independent Overflow Area

This is a separate area of disk storage which can be used in case the Cylinder Overflow Areas become full. It is specified in OS/VS by coding the subparameter OPTCD=I on the DD statement; a separate DD statement can be used for the Independent Overflow Area or it may be combined with the Prime Data Area and Index Area on one DD statement (see the 'Programmers Guide' under the section on INDEXED organization). In DOS/VS the Independent Overflow Area is specified on a separate EXTENT card with type code '2'.

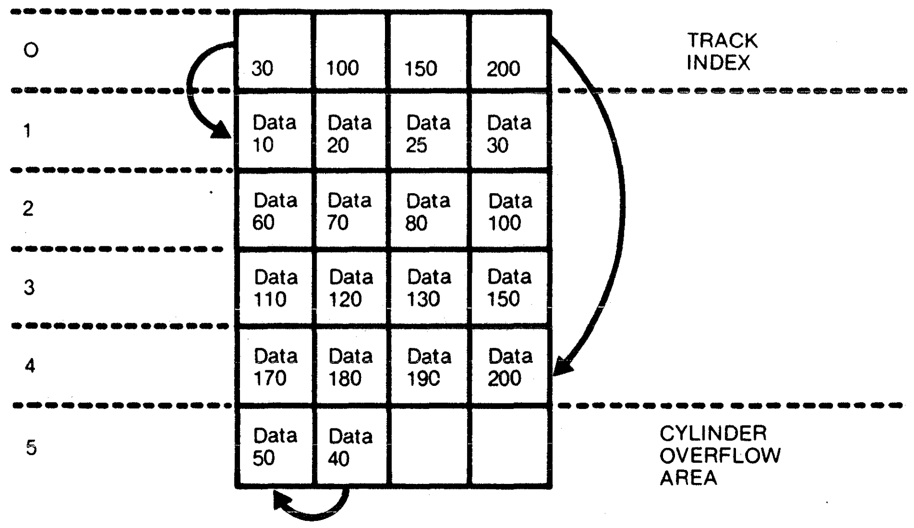
Addition of Records

On creation records are placed in the Prime Data Area. Subsequently records can be **added** to the data set in their key sequence position but this will always cause a record to 'overflow'; the INDEXED routines will place the overflow record in the Cylinder Overflow Area (if there is one, and space is available) or in the Independent Overflow Area. For example, the addition of a record with key 25 would cause the following alteration to the first cylinder of the data set:



Notice that the record (key 25) has been added in its 'logical' key-sequence position in the data set; this has caused a record (key 50) to overflow. INDEXED routines place the overflow record in the Cylinder Overflow Area (in this case) and also update the Track Index. The INDEXED routines also make a note, in the Track Index record for track 1, that a record with key 50 has overflowed from track 1 and they write the disk address of this record in the Track Index record for track 1.

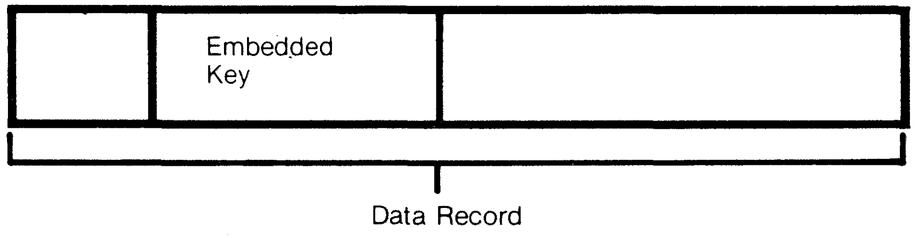
Consider the situation again after the addition of a record with key 30.



The INDEXED routines make a note (in the Track Index record for track 1) that 50 is the highest key to have overflowed from track 1 but that 40 is the next key in sequence after the highest key on track 1 (which is 30). Also the INDEXED routines write the disk address of the record with key 40 in the Track Index record for track 1. In the overflow area, the INDEXED routines set a pointer to point from record 40 to record 50. Thus the records of the INDEXED data set can still be accessed in ascending key sequence after the addition of records 25 and 30. Records 10, 20, 25 and 30 are retrieved and then a chain of pointers enables records 40 and 50 to be retrieved. We will not consider the addition of any more records but you should be able to see that, as more and more records are added to the data set, the average retrieval time, for both sequential and direct processing, will be increased. Eventually the data set may have to be recreated, eliminating the overflow records, in order to regain the original performance.

Keys in Indexed Data Sets

Each INDEXED record is associated with a **recorded key** - this key appears with the record on the data set. The recorded key is usually 'embedded' within the record.



(The system also has its own key which is external to each block on the data set - but this need not concern us here).

The **source key** is the character string or character variable which is used in READ/WRITE statements in order to retrieve/write the record.

DOS/VS Considerations

INDEXED data sets can contain fixed blocked or unblocked records. If the records are blocked, each logical record must contain an embedded key. The location of this embedded key within the data is specified by the KEYLOC(n) option of the ENVIRONMENT attribute and the length of the key is specified in the KEYLENGTH(n) option.

OS/VS Considerations

INDEXED data sets can contain either fixed length or variable length records, blocked or unblocked. If you are creating a blocked data set and the keys associated with each logical record are not a part of the data of the record (i.e. non-embedded keys) then the non-embedded key will be attached to the front of the data of each logical record, provided that you specify RKP=0 in the DD statement. (The record length will be equal to the sum of the data and key lengths).

RKP means Relative Key Position. An RKP value of 1 or greater indicates that there **is** an embedded key within the data. RKP=1 specifies that the embedded key starts at the second byte; RKP=3 specifies the **fourth** byte. RKP=0 means that the key is **not** embedded in the data. For **embedded** keys the RKP specification can be overridden by the KEYLOC(n) option in the file declaration. KEYLOC(1) indicates an embedded key starting in the **first** byte of the data. If KEYLOC(0) is specified then the RKP specification is used. If no RKP value is specified, RDP=0 is assumed.

For variable length records the RKP specification includes the four byte control field. So RKP=4, for a variable length record data set, would indicate an embedded key starting immediately after the four byte control field.

Declaring an Indexed File - OS/VS Considerations

An example of an OS/VS declaration for an INDEXED file which is to be read directly is shown below:

```
DCL ISFILE FILE DIRECT KEYED INPUT
           ENV(KEYLOC(n) INDEXED KEYLENGTH(n));
```

The options KEYLOC and KEYLENGTH are usually replaced by the DCB subparameters RKP and KEYLEN in the DD statement. DIRECT implies KEYED but not vice versa. Use KEYED whenever you want to use keys, i.e. when using the KEY, KEYTO or KEYFROM options (see later).

Declaring an Indexed File - DOS/VS Considerations

An example of a DOS/VS declaration for an INDEXED file which is to be read directly is shown below:

```
DCL ISFILE FILE DIRECT KEYED INPUT
           ENV(KEYLOC(n) INDEXED KEYLENGTH(n)
               EXTENTNUMBER(n) OFLTRACKS(n) MEDIUM(...)
               INDEXMULTIPLE HIGHINDEX(n)    );
```

DIRECT implies KEYED but not vice versa. Use KEYED whenever you want to use keys, i.e. when using the KEY, KEYTO and KEYFROM options (see later).

Most of the additional ENVIRONMENT options have been described in the preceding text. EXTENTNUMBER(n) specifies the number of extents of the INDEXED data set: one for the Prime Data Area, one for the Cylinder/Master Index Area and one for the Independent Overflow Area. The default is EXTENTNUMBER(2) - no Independent Overflow Area. HIGHINDEX(n) is required if the Cylinder/Master Index Area is on a different device **type** to the Prime Data Area. HIGHINDEX (3330) specifies that the index is on a 3330 device but the prime data is on some other device **type.**

PL/I Statements for Processing Indexed Data Sets

There is a table in the Language Reference Manual which summarizes the various ways in which INDEXED data sets can be used. Look up this table in the chapter called 'Record-Oriented Transmission'. Notice that an INDEXED file cannot be declared DIRECT OUTPUT - i.e. there is no direct creation, only sequential.

The following options may be new to you.

KEY Option

This is used to specify the key to be used when reading the data set. This is called the **source** key. It can be a character string or a character variable.

```
  READ FILE(ISFILE) INTO(AREA) KEY('123ABC');
```

KEYFROM Option

This is used to specify the source key to be used when creating the data set. It can be a character string or a character variable.

```
  DCL KEY CHAR(6);
  .
  WRITE FILE(ISFILE) FROM(AREA) KEYFROM(KEY);
```

KEYTO option

This brings the recorded key into the specified variable along with the record itself. This option is not required if the key is embedded within the data of the record. It is useful for records with non-embedded keys.

```
 DCL KEY CHAR(6);
 •
 READ FILE(ISFILE) INTO(AREA) KEYTO(KEY);
```

Sequential Processing of Indexed Data Sets

The associated file must be buffered. Locate mode or move mode processing can be used. The data set must be created sequentially by writing records in ascending key sequence. When reading the data set sequentially, records are retrieved in this ascending key sequence starting with the first record on the data set. Sequential processing can start at a particular record by using the KEY option; subsequent READ statements without the KEY option will read the next record in the data set.

```
 DCL ISFILE FILE RECORD SEQUENTIAL KEYED
             ENV(INDEXED KEYLENGTH(4).....);

 READ FILE(ISFILE) INTO(AREA) KEY('1234');
 •
 READ FILE(ISFILE) INTO(AREA);
```

GENKEY Option

The first READ statement reads a particular record into AREA. The next READ statement reads the very next record on the data set into AREA. It is possible to start processing records in a particular group (i.e. records whose keys start with the same characters) by using the GENKEY option of the ENVIRONMENT attribute of the file. This specifies that you will be using a short key (generic key) to specify the start of the group of records you want to read. If GENKEY is not specified and a short key is used then the short key would be padded out on the **right** with blanks to the length specified in the KEYLENGTH option for the file (or in the KEYLEN DCB subparameter in OS/VS).

```
DCL ISFILE FILE RECORD SEQUENTIAL KEYED
            ENV(INDEXED GENKEY KEYLENGTH(9) .....);
READ FILE(ISFILE) INTO(AREA) KEY('1ØØ');
•
READ FILE(ISFILE) INTO(AREA);
```

The first READ statement reads the first record which starts with the characters '100'. The next READ statement reads the very next record on the data set. This is very useful for processing groups of records on the data set but it is up to the programmer(you) to detect the end of each group.

It is possible to sequentially add records on the end of an Indexed data set provided that there is space at the end and that the added records have keys which are higher than any already existing on the data set. The associated file would be declared SEQUENTIAL OUTPUT. In order to add records in the middle of the data set DIRECT processing must be used.

Records can be updated when the associated file is declared SEQUENTIAL UPDATE by reading in the record, updating it and issuing the REWRITE statement.

OS/VS Considerations

In OS/VS records can be 'logically' deleted, during sequential processing, by reading in a record and issuing a DELETE statement. The effect of the DELETE statement is to write (8)'1'B in the first byte of the data (see later section on deleted records). The DELETE statement can only be used if OPTCD=L is specified in the DD statement that defined the data set when it was created.

Direct Processing of Indexed Data Sets

Only move mode processing can be used.

When the associated file is declared as DIRECT UPDATE records can be added in the middle of the data set by using the WRITE statement or already existing records can be updated using the REWRITE statement. The REWRITE statement can be used to update a record on the data set without pre-reading it - this requires the use of the KEY option.

OS/VS Considerations

Records on the data set can also be flagged as deleted when the associated file is declared DIRECT UPDATE. The DELETE statement writes (8)'1'B into the first byte of the data. This 'logical' deletion is only possible if OPTCD=L is specified in the DD statement that defined the data set when it was created. Deleted records are ignored during subsequent sequential processing. If you try and retrieve a deleted record directly then the KEY condition will be raised (see later).

```
DCL ISFILE DIRECT UPDATE ENV(INDEXED KEYLENGTH(9)....);
.
DELETE FILE(ISFILE) KEY('123456789');
```

KEY Condition

This is just one of many conditions which are raised when PL/I recognizes an abnormal situation. (For more information on other PL/I On-conditions and how to handle them see the topic 'Handling Exceptional Conditions' - but not now!). The main reason for raising the KEY condition with Indexed data sets are:

- a key sequence error on creating the data set
- no room to add a record (overflow areas full)

- an attempt to add a record with a key which already exists on the data set (duplicate record)
- no record on the data set with the key specified in the READ statement.

EVENT Option

This can be attached to I/O statements for direct processing of INDEXED data sets. When EVENT is used control passes immediately to the next statement in the program, rather than waiting for the I/O operation to complete.

It can be used to simulate the overlap of processing and I/O which takes place automatically in sequential processing if you specify more than one buffer. (It is not possible to have buffers when processing INDEXED data sets directly). The WAIT statement is used to halt processing until the associated event has taken place. The following coding illustrates a possible method of overlapping processing and I/O by using the EVENT option, the WAIT Statement and two I/O areas.

```
 DO WHILE(SW);
       READ FILE(ISFILE) INTO(AREA1) KEY(FLD1);
       READ FILE(ISFILE) INTO(AREA2) KEY(FLD2) EVENT(X);

       /* PROCESS RECORD IN AREA1 WHILE */
       /* RECORD IS READ INTO AREA2     */

       WAIT(X); /* WAIT FOR RECORD IN AREA2 */

       /* NOW PROCESS RECORD IN AREA2 */

 END;
```

The first READ does not have the EVENT option and so control does **not** pass to the second READ until the first READ has taken place. So, while the second READ is taking place, we can process the record in AREA1. When we wish to process the record read into AREA2 we must issue a WAIT statement to make sure that the READ has completed. In our example, X is an EVENT VARIABLE which is contextually declared as such; it could also have been explicitly declared. An event variable links the WAIT statement to a particular I/O event.

OPTIMIZATION Options

These are options of the ENVIRONMENT attribute of the file which can be specified to reduce program size or reduce execution time of the program.

INDEXAREA - OS/VS

This reduces execution time. The highest level index is brought into storage and searched there rather than on disk. The programmer can limit the amount of storage allowed for this purpose by specifying a size - INDEXAREA(n). If the highest level index is larger than the specified size then it is **not** brought into main storage.

INDEXAREA - DOS/VS

This reduces execution time by bringing in the cylinder index into storage and searching it there rather than on disk. The programmer can specify that only part of the cylinder index should be brought in at one time by adding a size value - INDEXAREA(n). If no size specification is used, then the whole of the cylinder index is brought in to storage.

NOWRITE

When a file is declared DIRECT UPDATE a program can add new records to the associated data set (using the WRITE statement) and update existing records (using the REWRITE statement). In OS/VS records can also be logically deleted (using the DELETE statement). If you only want to update records directly and not add them, then specify NOWRITE in the file declaration. This will prevent the inclusion of those access modules which add records to the data set, thus reducing the size of your program.

ADDBUFF

This reduces execution time when you are adding records directly. With ADDBUFF the addition of a record takes place by reading in the track concerned, adding the record in its logical place, writing the track back to disk and writing to the overflow area the record which was 'pushed off' the track to make room for the new record. If you do not want the whole track to be brought into storage (because you do not have enough room) then you can specify a size on the option: ADDBUFF(n). Obviously it is more efficient if the whole track is brought into storage. Without the ADDBUFF option, the addition is performed 'out on disk' by a process which involves more disk accesses but less computer storage.

Exercises

1. What is the purpose of the master index?
2. (OS/VS only). Suppose that KEYLOC(0) and RKP=3 have been specified for the records of an FB-format data set. Will the keys be embedded or non-embedded?
3. File X is associated with a data set containing records, which consist of a 5 byte embedded key followed by 10 bytes of character data. File Z is similar, but the keys are not embedded.
   a) Write a move mode input statement to read a record sequentially from the data set associated with file X, so that the data goes into a filed called DATA and the key into a field called KEY. Include the declarations of these fields.
   b) Repeat the above for file Z.
4. What is wrong with the following declaration?

```
DCL DUD FILE DIRECT UPDATE
             ENV(GENKEY INDEXED ADDBUFF INDEXAREA(2ØØØ));
```

5. Suppose you want to use the statement

```
        READ FILE(INDXD) INTO (AREA) KEY('X');
```

   to 'position' an indexed data set so that you can read it sequentially **starting** with the first record whose key **begins** with 'X'. What must you specify in the ENV attribute of the associated file, and why?
6. Declare a file ISET, associated with an indexed data set, and write suitable I/O statements to perform the following:
   a) Read the record in the data set with the key '168' into RECAREA
   b) XAREA contains a **new** record whose key is in KEYAREA. Add this record to the data set.
   c) (OS/VS only) Delete the record whose key is '515'.
   d) Replace the record whose key is '212' with a new record which is in NEWAREA.
7. Write the coding to create an INDEXED data set on a 3330 disk with serial number 30WORK. The records are 80 bytes long. The key is a 10 byte part number which starts at the fourth byte of the record. There is to be an Independent Overflow Area created for the data set and on each cylinder two tracks are to be reserved for overflow records if records are subsequently added to the data set. The records are to be in blocks of 10. Create the INDEXED data set from a card file which has already been sorted into ascending key sequence.
8. If you have read through the topics CONSECUTIVE, REGIONAL and INDEXED organization, then summarize the advantages/disadvantages of each type and indicate when each would be used.

Answers

1. A Master Index contains entries which enable the system to select the correct track of the Cylinder Index, prior to accessing a record. A Master Index is normally recommended when the Cylinder Index occupies more than three tracks.

2. (OS/VS only). Since KEYLOC(0) has been specified, the RKP specification will be used. An **embedded** key will therefore be used, starting at the **fourth** byte of each record.

3. a)

```
     DCL 1 FIELD,
           2 KEY  CHAR(5),
           2 DATA CHAR(1Ø);

     READ FILE(X) INTO(FIELD);
```

b)

```
     DCL DATA CHAR(1Ø);
     DCL KEY  CHAR(5);

     READ FILE(Z) INTO(DATA) KEYTO(KEY);
```

4. GENKEY may only be specified with a SEQUENTIAL file. Otherwise this declaration is satisfactory.

5. GENKEY must be specified to prevent the key from being padded to the right with blanks, and the result being treated as a specific key.

6.

```
 DCL ISET FILE DIRECT UPDATE KEYED
            ENV(INDEXED KEYLENGTH(3)........);

 READ FILE(ISET) INTO(RECAREA) KEY('168');
 WRITE FILE(ISET) FROM(XAREA) KEYFROM(KEYAREA);

 DELETE FILE(ISET) KEY('515'); /* OS/VS ONLY */

 REWRITE FILE(ISET) FROM(NEWAREA) KEY('212');
```

OS/VS Solution

7.

```
 ISPROC:PROC OPTIONS(MAIN);
        DCL ISFILE FILE KEYED RECORD OUTPUT ENV(INDEXED);
        DCL CARDIN FILE RECORD;
        DCL 1 CARD,
              2 DATA1  CHAR(3),
              2 KEYFLD CHAR(1Ø),
              2 DATA2  CHAR(67);

        DCL EOF BIT(1) INIT('Ø'B);
        ON ENDFILE(CARDIN) EOF='1'B;

        READ FILE(CARDIN) INTO(CARD);
        DO WHILE(¬EOF);
             WRITE FILE(ISFILE) FROM(CARD) KEYFROM(KEYFLD);
             READ FILE(CARDIN) INTO(CARD);
        END;
 END;
```

```
//GO.ISFILE   DD DSNAME=ISFILE(INDEX),UNIT=333Ø,SPACE=(CYL,1),
//               DCB=(RECFM=FB,BLKSIZE=8ØØ,DSORG=IS,KEYLEN=1Ø,RKP=3,
//               OPTCD=LIY,CYLOFL=2),DISP=(NEW,KEEP),VOLUME=SER=3ØWORK
//            DD DSNAME=ISFILE(PRIME),UNIT=333Ø,SPACE=(CYL,4),
//               DCB=(DSORG=IS,DISP=(NEW,KEEP),VOLUME=SER=3ØWORK
//            DD DSNAME=ISFILE(OVFLOW),UNIT=333Ø,SPACE=(CYL,4),
//               DCB=DSORG=IS,DISP=(NEW,KEEP),VOLUME=SER=3ØWORK
```

DOS/VS Solution

7.

```
  ISPROC:PROC OPTIONS(MAIN);
         DCL ISFILE FILE RECORD OUTPUT KEYED
                    ENV(INDEXED KEYLENGTH(1Ø) KEYLOC(4)
                        FB RECSIZE(8Ø) BLKSIZE(8ØØ)
                        MEDIUM(SYSØØ8,333Ø)
                        OFLTRACKS(2) EXTENTNUMBER(3)  );

         DCL CARDIN FILE RECORD INPUT
                    ENV(MEDIUM(SYSIPT,254Ø) F RECSIZE(8Ø));

         DCL 1 CARD,
               2 DATA1  CHAR(3),
               2 KEYFLD CHAR(1Ø),
               2 DATA2  CHAR(67);
         DCL EOF BIT(1) INIT('Ø'B);

         ON ENDFILE(CARDIN) EOF='1'B;

         READ FILE(CARDIN) INTO(CARD);
         DO WHILE(¬EOF);
              WRITE FILE(ISFILE)FROM(CARD) KEYFROM(KEYFLD);
              READ FILE(CARDIN) INTO(CARD);
         END;
  END;
```

```
// ASSGN SYSØØ8,333Ø,VOL=3ØWORK,SHR
// DLBL ISFILE,'ISFILE',,ISC
// EXTENT SYSØØ8,3ØWORK,4,1,1994,1
// EXTENT SYSØØ8,3ØWORK,1,2,19ØØ,19
// EXTENT SYSØØ8,3ØWORK,2,3,1996,1
```

8. CONSECUTIVE organization is used when you are only interested in processing the data set sequentially, one record after another. REGIONAL organization is used when you are primarily interested in retrieving records directly in no particular sequence. REGIONAL data sets can also be processed sequentially but this will be in ascending region number, which is not usually the same as ascending key sequence. INDEXED organization provides sequential processing by ascending key sequence **and** direct processing via a recorded key. INDEXED is neither as efficient at sequential processing as CONSECUTIVE nor as efficient at direct processing as REGIONAL. In addition INDEXED organization will generally occupy more space on disk than the other two organizations because of the space required for the indexes.

Topic 13

Topic 13

Virtual Storage Access Method

In this topic you will learn about the three VSAM data set organizations and the reasons for using them. If you are already familiar with VSAM organization then you can leave out the associated sections within this topic and read the sections relating to PL/I.

Objectives

At the end of this topic you should be able to:

- understand the basic ideas of VSAM data set organization
- describe the three types of VSAM data set organization (key sequenced data sets, entry sequenced data sets and relative record data sets) and under what circumstances they would be used.
- write PL/I statements to declare and access VSAM data sets
- understand and make use of alternative indexes
- understand the circumstances in which the KEY condition will be raised.

Introduction

Virtual Storage Access Method (VSAM) is a way of organizing and accessing data on disk. VSAM is compatible across operating systems and across disk devices.

There are three types of VSAM organization.

The most important is the KSDS (key sequenced data set). This is intended for applications in which records are processed directly by key (using indexes) and also sequentially in ascending/descending key sequence. It is, therefore, intended for similar applications to those for which INDEXED organization is used but VSAM has been designed to minimize some of the drawbacks of INDEXED organization (namely the increase in average retrieval time resulting from the addition of new records to an INDEXED data set).

The entry sequenced data set (ESDS) is intended for applications in which records are processed one after another without reference to keys. CONSECUTIVE data sets are used similarly but an ESDS has the advantage that records can be added at the end of the ESDS without the necessity of recreation (also, it is possible to process an ESDS directly, as we shall see).

The relative record data set (RRDS) would be used in applications which require rapid direct access. Direct access is quicker with an RRDS than a KSDS. The records are retrieved directly from the RRDS by specifying the 'position' of the record within the data set (first, fifth, hundredth). The RRDS is thus intended for similar application areas to those for which REGIONAL(1) organization is used.

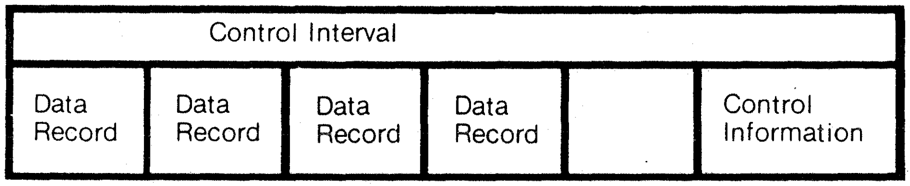
VSAM Data Spaces, VSAM Data Sets and VSAM Catalog

VSAM data sets are allocated in an area on disk called a **VSAM data space.** There can be several VSAM data sets in this VSAM data space; there may be only one: the VSAM data space may even be empty. The VSAM data space and the vsam data sets are defined using a special utility AMS (Access Method Services) and information about the data space and the data set is placed in a **VSAM catalog** - this is a central pool of information which contains, among other things, the names of each VSAM data set, its maximum record size and where it is situated. VSAM data sets consist of **control areas** which are subdivided into **control intervals.**

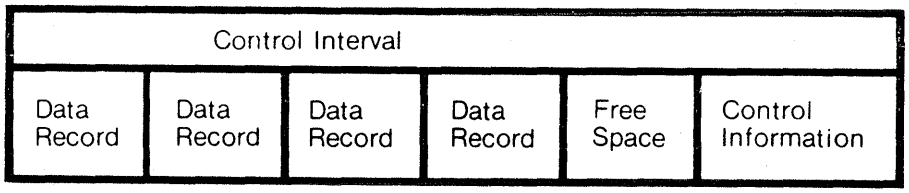
Control Intervals

In other data set organizations the physical record (or block) is the unit of data transfer between computer storage and the peripheral device containing the data set. In VSAM the unit of data transfer is the **control interval.**

Data records are placed in the control interval starting from the left-hand side and information about the length of the record is placed in the control interval starting from the right-hand side.



An ESDS and a KSDS can have variable length records but an RRDS must have fixed length records. In a KSDS it is possible (and usual) to have a certain amount of 'free space' within each control interval. The purpose of free space will be explained later although you can probably see a use for it already. Thus a control interval in a KSDS might look like this:



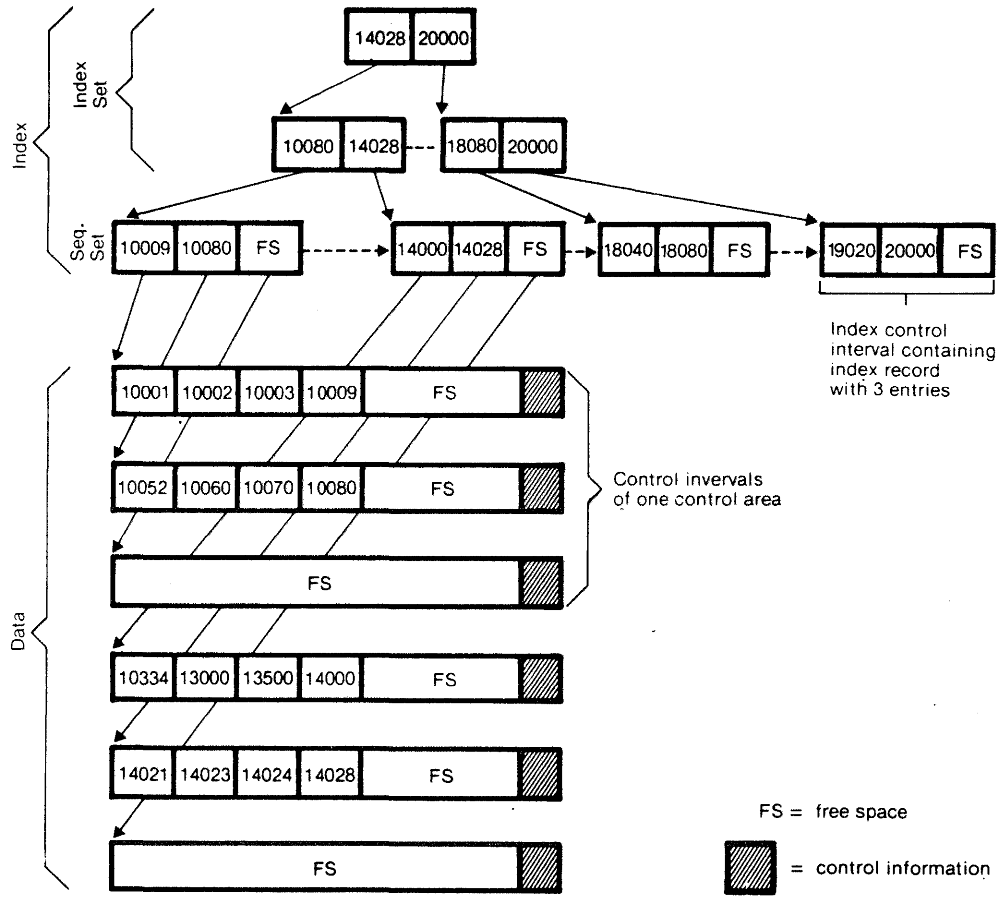
The size of the control interval can vary from 1/2 K to 32K. This size can be specified when the data set is setup - a typical size is 2K.

Control Areas

Control Intervals are grouped into units called Control Areas. VSAM works out the Control Area size. The maximum control area size is one cylinder. For a KSDS it is possible (and usual) to leave a number of control intervals within each control area completely free of data records. Can you see a use for this? If not, it will be explained later.

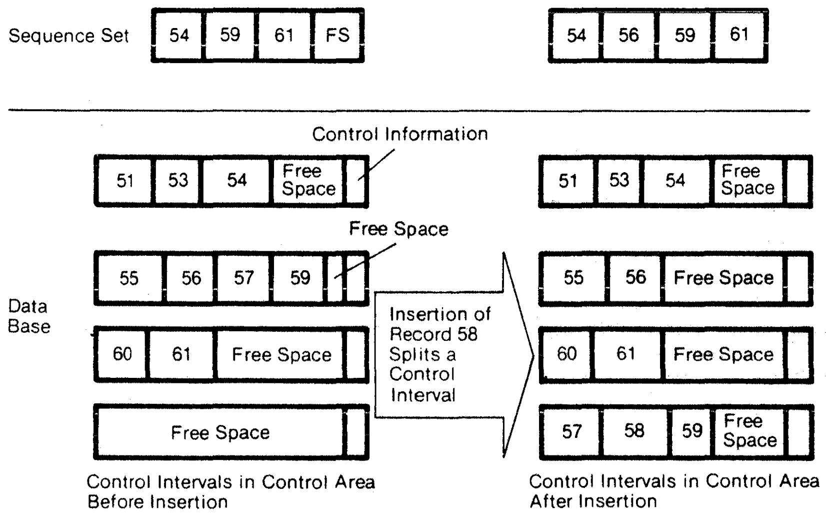
KEY Sequenced Data Sets

These consist of an index and data portion, each of which is divided into control intervals. The lowest level of index is the **sequence set** which contains the highest key for each control interval in the data set. Higher levels of index are used to refer to the sequence set. These higher levels are collectively called the index set - the highest level of all contains only one control interval. In the example below a control area in the data portion contains three control intervals - this is smaller than normal, but simplifies the diagram.


Index
Index Set
Seq. Set
Data
14028 20000
10080 14028
18080 20000
10009 10080 FS
14000 14028 FS
18040 18080 FS
19020 20000 FS
Index control interval containing index record with 3 entries
10001 10002 10003 10009 FS
10052 10060 10070 10080 FS
FS
10334 13000 13500 14000 FS
14021 14023 14024 14028 FS
FS
Control invervals of one control area
FS = free space
= control information

Adding a Record to a KSDS

A record can be added to a KSDS in its logical (key-sequence) position by moving other records about and reducing the free space within the affected control interval. There is no loss in performance. Sooner or later, however, the free space in the control interval will be used up. The diagram below illustrates a VSAM KSDS into which we wish to add a new record with key 58 but there is not enough room in the control interval to add this record.

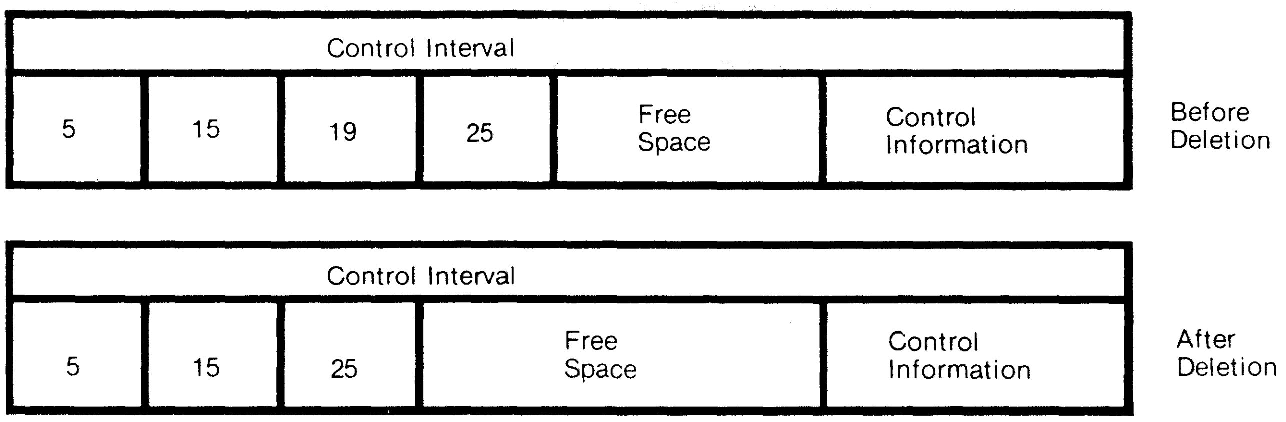


What happens? A **control interval split** takes place and roughly half the data in the original control interval is moved to the completely 'free' control interval at the end of the control area. This is fine and once again there is little or no loss in performance but what happens when there are no more 'free' control intervals within the control area? A **control area split** takes place and roughly half the control intervals in the original control area are moved to a completely new control area at the end of the data set. There is little or no loss in performance.

You can see that VSAM reorganizes the data set whenever it is necessary but as far as you are concerned, as a programmer, you are just adding another record. Obviously when the data set is originally defined (using AMS) a certain amount of thought has to go into deciding suitable percentages of free space within a control interval, within a control area and within the data set as a whole. This should be part of the planning process, taking into account how the data set is going to be used - better to plan before than reorganize afterwards.

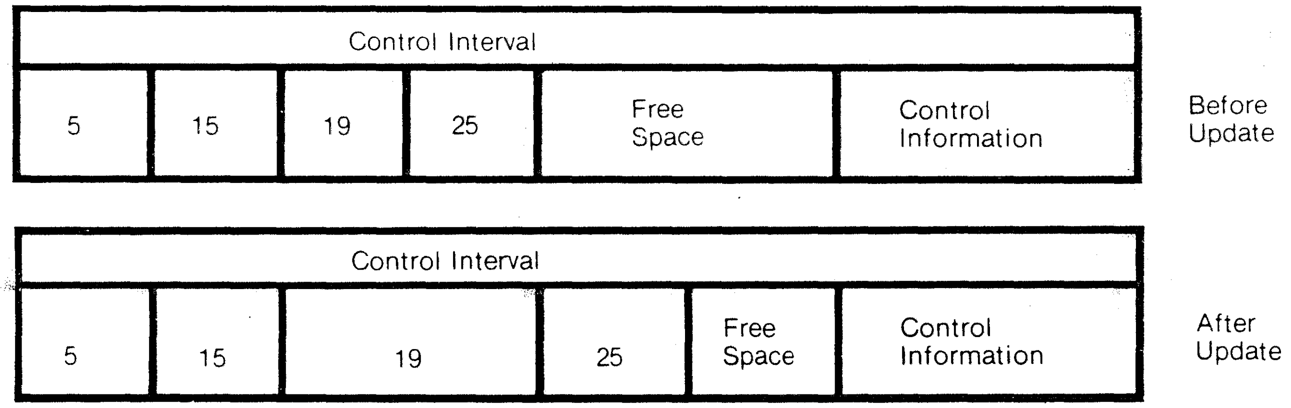
VSAM Deletions on a KSDS

A VSAM DELETE statement causes a record to be **physically** deleted from the data set. This is obviously useful because it increases the amount of free space within the control interval, enabling more records to be added later on. The diagrams below show a control interval before and after the deletion of the record with key 19.



Updating Records on a KSDS

Records can be updated in all three VSAM organizations but only in a KSDS can records be updated and written back to the data set with a different length. This is because only a KSDS has 'free space'. The diagrams below show a control interval before and after the updating of record with key 19, which is written back to the data set with a different length.



Entry Sequenced Data Sets

An ESDS consists of control intervals and control areas but there is no index and no free space.

An ESDS is essentially processed sequentially although access by Relative Byte Address is also possible (see later). Records can be added at the end of the data set by declaring the File DIRECT UPDATE and issuing a WRITE statement. You cannot delete records in an ESDS.

Relative Record Data Sets

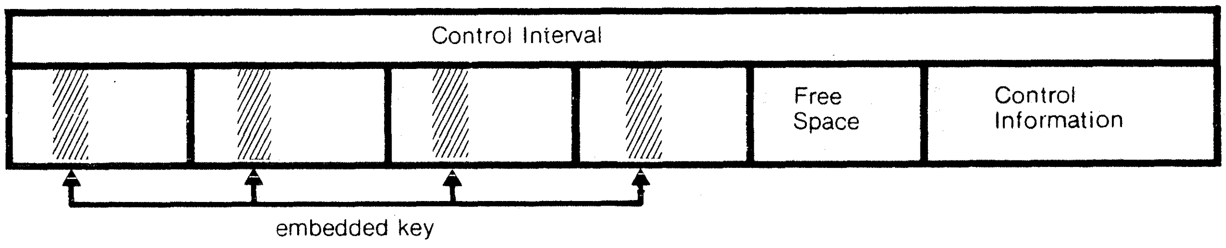
An RRDS consists of control intervals and control areas but there is no index and no free space. An RRDS can be processed sequentially or directly by relative record number.

When the RRDS is originally created some relative record positions can be left empty so that subsequent additions to the RRDS can be inserted in the data set.

Records can be deleted from the data set - but this merely causes the associated relative record slot to be made empty (this is not the same as a VSAM deletion on a KSDS).

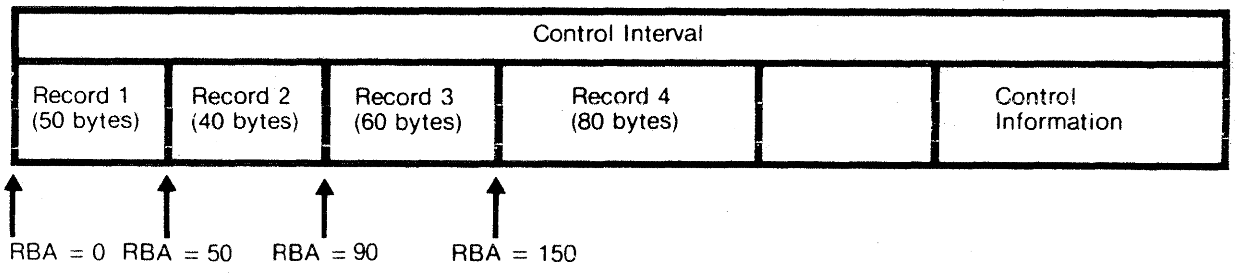
VSAM Keys

For a **KSDS** the keys **must** be embedded within the data record. They must be fixed length and in a fixed position within the data record.



In an **RRDS** records are identified by a relative record number which starts at 1 and is incremented by one for each succeeding record. This relative record number can be used as a key to process the RRDS directly. The relative **record number is a character string of length 8 and must represent an unsigned fixed decimal constant.**

An ESDS is essentially intended for sequential processing but it can be processed using the Relative Byte Address (RBA) of the record as a key. The RBA is the displacement of the record, in bytes, from the start of the data set.



The RBA can be obtained by means of the KEYTO option (see later) when creating or reading the data set. This RBA can subsequently be used as a key to process the data set.

Declaring a VSAM File

File attributes (such as DIRECT, SEQUENTIAL, INPUT etc.) were discussed in Topic 5 and they should be coded appropriately. However, because VSAM has a catalog (containing information such as key length and key position within the record) less information need be specified within the ENVIRONMENT attribute. For instance, you do not need to code the KEYLENGTH option of the ENVIRONMENT attribute but, if you do, then it will be checked against information in the VSAM catalog for the file concerned; the UNDEFINEDFILE condition will be raised if you specify the KEYLENGTH value incorrectly.

A VSAM data set can be accessed using a file with the attribute ENV(VSAM) or ENV(INDEXED) - see later section on VSAM/ISAM compatibility interface.

Also, VSAM gives you the option of having password protection for its data sets. The password, if used, is placed in the VSAM catalog when the data set is originally set up (using the AMS utility). Subsequently any programs which want to use the data set must specify the correct password in the PASSWORD option of the ENV attribute of the associated file. (There are several levels of password protection and there is more information in the

Programmer's Guide). The password specified can be a character string or a character variable.

Example:

```
  DCL VFILE FILE RECORD DIRECT KEYED INPUT
                 ENV(VSAM PASSWORD('SESAME'));
```

If the password is incorrect the operator can be given a chance to correct it. The number of attempts to specify the correct password is defined by AMS when setting up the data set. Failure to specify the correct password after the defined number of attempts results in the UNDEFINEDFILE condition (mentioned in a previous topic).

The file attribute KEYED must be used whenever you are using keys, i.e. when using the KEY, KEYTO or KEYFROM options of the I/O statements (see later). The file attribute DIRECT implies KEYED but KEYED does not imply DIRECT.

DOS/VS Consideration

The MEDIUM option of the ENVIRONMENT attribute is not necessary for a VSAM file.

PL/I Statements for Processing VSAM Data Sets

There are several tables in the Language Reference Manual which summarize the various ways in which VSAM data sets can be used. Look up this table in the chapter called 'Record Oriented Transmission'.

The following options may be new to you.

KEY Option

This is used to specify the key to be used when reading the data set **directly.** This key can be an embedded key (for a KSDS), a relative record number (for an RRDS) or an RBA (for an ESDS or, more rarely, a KSDS). The example below is reading directly from a KSDS with a six byte embedded key:

```
      READ FILE(KSDS) INTO(AREA) KEY('37Ø168');
```

KEYFROM Option

This is used to specify the key to be used when creating the data set. This key can be an embedded key (for a KSDS) or a relative record number (for an RRDS) - but not an RBA. The example below is writing a record to an RRDS using a relative record number. If a record with the same relative record number has already been written to the data set then the KEY condition will be raised (see later).

```
  DCL RELRECNO CHAR(8);
  RELRECNO='ØØØØØ123';

  WRITE FILE(RRDS) FROM(AREA) KEYFROM(RELRECNO);
```

KEYTO Option

This brings the key of the record into the specified variable in storage. This key will be an embedded key (for a KSDS), a relative record number (for an RRDS) or an RBA (for an ESDS). The KEYTO option, if used at all, would normally be used on input but it can be used on output. The following example uses the KEYTO option to recover the RBA of a record as it is written:

```
    DCL RBA CHAR(4);
    WRITE FILE(ESDS) FROM (AREA) KEYTO(RBA);
```

The tables show all possible ways to declare and use the VSAM file. The following points should be noted.

KSDS

The table to be used is the one which shows statements for 'creating and accessing VSAM data sets via prime or alternate indexes'. The prime index is the index component of the key sequenced data set and is built automatically by VSAM when the data set is created. Alternate indexes will be covered later.

When a KSDS is created, the associated file must be declared KEYED SEQUENTIAL OUTPUT and records must be written to the data set in ascending key sequence.

A file declared as UPDATE can be used to add records to the data set (WRITE statement), update records on the data set (REWRITE statement) or delete records from the data set (DELETE statement).

ESDS

An ESDS can only be declared SEQUENTIAL although KEYED access is possible using Relative Byte Addresses. If you retrieve a particular record using a key then subsequent sequential processing continues from this new position in the data set. Records can be added at the end of the data set by declaring the associated file UPDATE and issuing a WRITE statement.

RRDS

An RRDS can be created using a SEQUENTIAL OUTPUT file or a DIRECT OUTPUT file. In direct creation records are placed in the relative record slot specified by the KEYFROM option. In sequential creation records are placed in successive record slots unless the KEY-FROM option is used to direct a record to a particular relative record slot.

The Use of Sequential Keyed Files

It is possible to start sequential processing from a particular record in the data set (KSDS, ESDS, or RRDS). For example, using a KSDS,

```
     DCL VSAM FILE RECORD SEQUENTIAL KEYED ENV(VSAM);
     •
     READ FILE(VSAM) INTO(AREA) KEY('123.999.1');
     •
     READ FILE(VSAM) INTO(AREA);
```

The first READ reads a particular record; the second READ reads the next record after that record in the data set.

In the example above the key is 9 characters long. If you want to start processing from a particular 'group' of records (for example records whose keys start with '123') then it is not sufficient merely to specify a key of '123' in the KEY option. The 'start key' would be padded on the right with blanks to a length of 9 characters - the specified key length - and a VSAM routine would search for a key '123ƀƀƀƀƀƀ'. There would probably not be a record with such a key on the data set.

GENKEY Option

To prevent this padding of the start key use the GENKEY (generic key) option of the ENVIRONMENT attribute.

Example:

```
     DCL VSAM FILE RECORD SEQUENTIAL KEYED
                     ENV(VSAM GENKEY);

     READ FILE(VSAM) INTO(AREA) KEY('123');
     •
     READ FILE(VSAM) INTO(AREA);
```

The first READ statement reads the first record on the data set beginning with '123'. The second READ statement reads the next record after the one just read.

KEY Condition

This is just one of many conditions which are raised when PL/I recognizes an abnormal situation. (For more information on PL/I ON-conditions and how to handle them see Topic 19 - but not now!). The main reasons for raising the KEY condition with keyed VSAM data sets are:

KSDS

- a key sequence error on creating the data set
- attempting to add a record with a key which already exists on the data set (duplicate key)
- no record on the data set with the key specified in the READ statement (also applies to generic key)
- no room to add a record

ESDS

- the RBA, specified as the key, is not a valid RBA on the data set

RRDS

- attempting to add a record to a relative record position which already contains a record.

BUFFERED and UNBUFFERED Files

The BUFFERED attribute specifies that data records should pass through an area of storage within the program before being written to a data set (output) or being read by the program (input). This allows LOCATE mode processing to be used but has the disadvantage that extra storage is required for the program. Other IBM access methods have always supported buffers for sequential and direct processing but not for direct processing. VSAM supports buffering for sequential and direct processing.

```
      DCL VSAM FILE DIRECT INPUT BUFFERED
                    ENV(VSAM);

      READ FILE(VSAM) SET(PTR) KEY(KEYFLD);
```

EVENT Option

This can be attached to I/O statements for processing of UNBUFFERED VSAM data sets. The main use of EVENT is for overlapping of processing and I/O during direct processing - this takes place automatically during sequential processing if you specify more than one buffer. When EVENT is used control passes immediately to the next statement in the program, rather than waiting for the I/O operation to complete. The WAIT statement is used to halt processing until the associated event has taken place. The following coding illustrates a possible method of overlapping processing and I/O by using the EVENT option, the WAIT statement and two I/O areas.

```
  DO WHILE(SW);
        READ FILE(VSAM) INTO(AREA1) KEY(FLD1);
        READ FILE(VSAM) INTO(AREA2) KEY(FLD2) EVENT(X);
        /* PROCESS RECORD IN AREA1 WHILE   */
        /* NEXT RECORD IS READ INTO AREA2  */
        WAIT(X);/* WAIT FOR COMPLETION OF 2ND READ */
        /* NOW PROCESS RECORD IN AREA2     */
  END;
```

The first READ does not have the EVENT option and so control does **not** pass to the second READ until the first READ has taken place. So, while the second READ is taking place, we can process the record read into AREA1. When we wish to process the record read into AREA2 we must issue a WAIT statement to make sure that the READ has completed. In our example X is an EVENT VARIABLE which is contextually declared as such; it could also have been explicitly declared.

VSAM/ISAM Compatibility

This section will be of interest to those people who currently use Indexed Sequential Access Method (ISAM) which is the access method for PL/I INDEXED files. VSAM can be used for all applications in which ISAM is used, but VSAM has many additional features. You may already have ISAM but wish to change to VSAM. If so, there is no need to rewrite all your previous INDEXED programs so that files have the attribute ENV(VSAM). The way in which PL/I handles this situation depends on the operating system being used.

OS/VS Considerations

Consider a program using an INDEXED file to access a VSAM data set. PL/I will provide access to the VSAM data set as if you were actually using a file with attribute ENV(VSAM) - this is 'native' access and no compatibility interface is involved. However you may wish to force the use of the VSAM/ISAM compatibility interface in order to do an ISAM (logical) deletion as opposed to a VSAM (physical) deletion or because the INDEXED program uses records with non-embedded keys (not possible on a VSAM KSDS). In order to force the use of the VSAM/ISAM compatibility interface you must code either RECFM=F/FB/V/VB or OPTCD=L in the AMP parameter of the DD statement for the data set. The AMP (Access Method Parameter) parameter replaces the DCB parameter used for other access methods.

DOS/VS Considerations

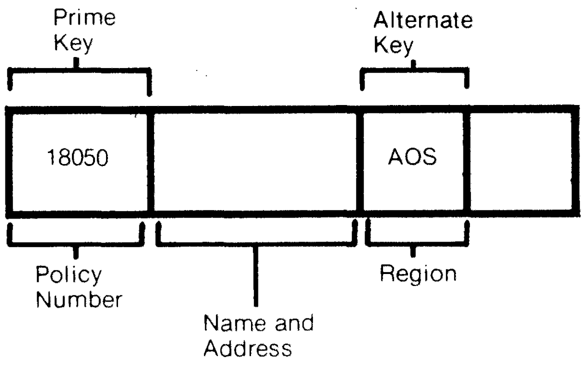
PL/I will automatically invoke the VSAM/ISAM interface whenever you try and access a VSAM data set using a file with attribute ENV(INDEXED). Using the interface limits the processing of the VSAM data set to those situations which apply for an INDEXED data set, e.g. no variable length records and no DELETE statement.

Alternate Indexes and Paths

The index of a KSDS is called the **prime index.**

It is possible to define one or more **alternate indexes** over the base KSDS (referred to as the **base cluster)** in order to access the KSDS by a different key within the record.

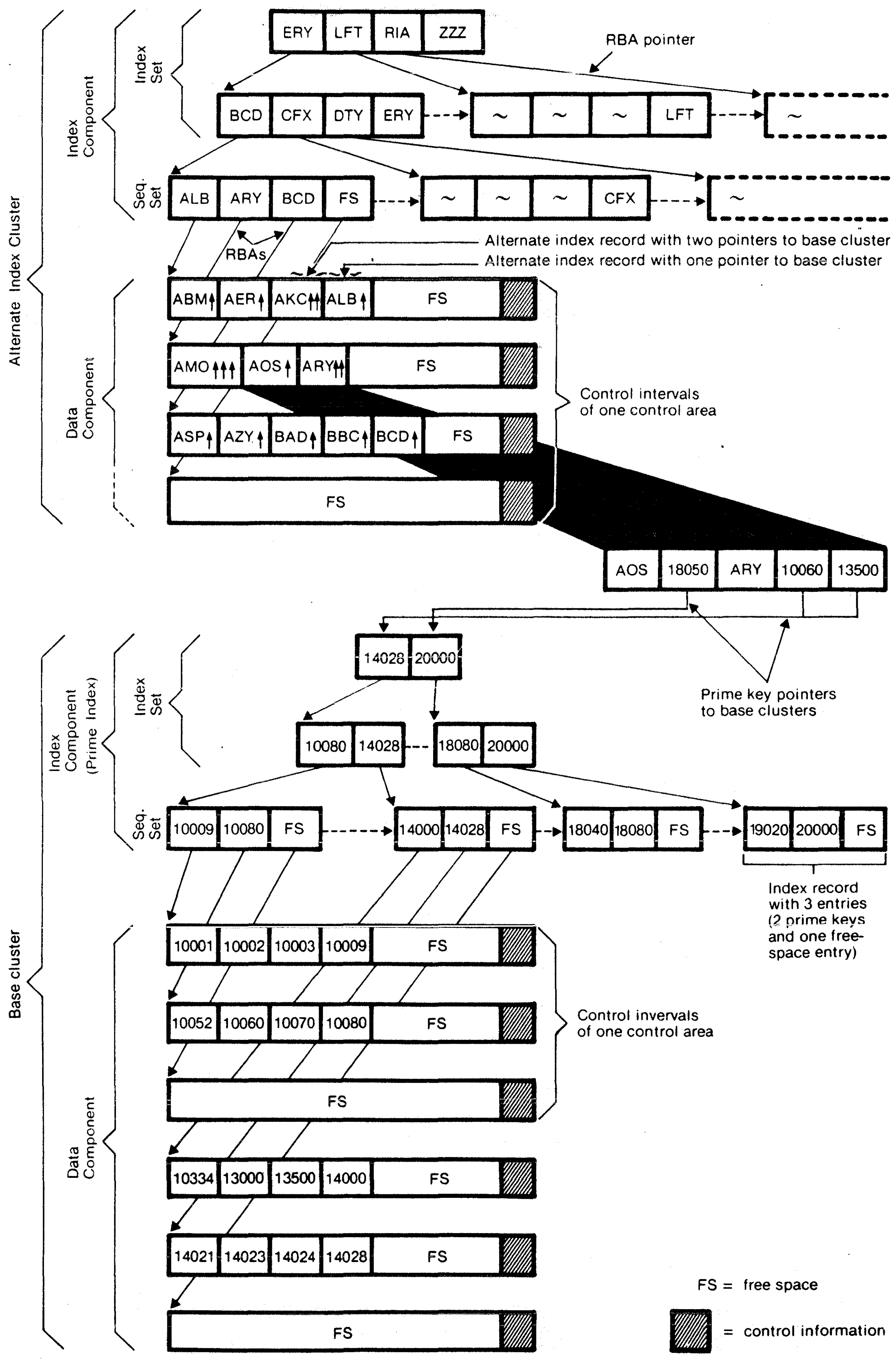
For example an insurance company might keep records containing information such as policy number, name and address and region.



Record in base KSDS
showing prime key and alternate key

The prime key is the policy number but sometimes the company wishes to access the records by region. So an alternate index is built to relate the alternate keys to the prime keys. The relation between the alternate index and the base data is shown on the next page.

You can see from the diagram that in the region AOS there is only one policy number (18050) and in the region ARY there are two policy numbers (10060 and 13500).

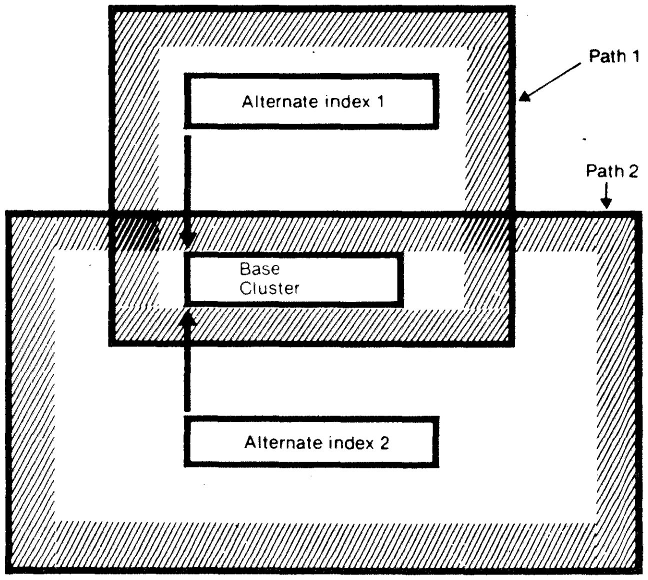

Alternate Index Cluster
Index Component
Index Set
Seq. Set
Data Component
ERY LFT RIA ZZZ
RBA pointer
BCD CFX DTY ERY
~ ~ ~ LFT
ALB ARY BCD FS
~ ~ ~ CFX
RBAs
Alternate index record with two pointers to base cluster
Alternate index record with one pointer to base cluster
ABM AER AKC ALB FS
AMO AOS ARY FS
ASP AZY BAD BBC BCD FS
FS
Control intervals of one control area
AOS 18050 ARY 10060 13500
Prime key pointers to base clusters
Base cluster
Index Component (Prime Index)
14028 20000
10080 14028 18080 20000
10009 10080 FS
14000 14028 FS
18040 18080 FS
19020 20000 FS
Index record with 3 entries (2 prime keys and one free-space entry)
10001 10002 10003 10009 FS
10052 10060 10070 10080 FS
Control invervals of one control area
10334 13000 13500 14000 FS
14021 14023 14024 14028 FS
FS = free space
▨ = control information

The alternate index is itself a VSAM KSDS in which each record contains the alternate key followed by the associated prime key (or keys) of the data in the base KSDS. An alternate index is **unique** if it contains only one prime key per alternate key, it is **non-unique** if it contains more than one prime key per alternate key. **In our example the alternate index is non-unique.**

The alternate index is defined and (usually) built using the AMS utility.

The alternate index and its associated base cluster must be 'linked' together before the base cluster can be accessed by alternate key. This link between alternate index and base cluster is called a path. A path is merely an entry in the VSAM catalog and it is defined using the AMS utility.

In PL/I terms a path is treated as a file. Opening a path opens both the alternate index and its associated base cluster. A base cluster might have two alternate indexes (see diagram below). To process the base cluster using alternate index 1 you should open Path 1.



You might find this use of the 'path' to be a little strange at first. In fact it is very useful - by opening the path you are specifying that you want to process the base cluster via the alternate index; if you open the alternate index you are specifying that you want to process the alternate index on its own.

An alternate index can also be defined on an ESDS. The alternate key would be some field within the record and the prime key would be an RBA for the associated record. **The alternate index would be defined and built using the AMS utility. Once the path linking the alternate index to the base ESDS has been defined, using AMS, then the ESDS can be processed directly using the alternate key.**

SAMEKEY Builtin Function

The SAMEKEY builtin Function is useful for processing base data sets via a **non-unique** alternative index. SAMEKEY returns '1'B if the I/O operation has been successful and there are more records on the data set with the same alternate key; otherwise SAMEKEY returns '0'B.

The following coding shows an example of the use of SAMEKEY. The file KSDS is keyed on the first six characters (BOOKKEY). The key for the alternate index starts on the 42nd byte and is 22 bytes long (AUTHOR). Notice that the key length and key location do not have to be specified in the declarations - such information is contained in the VSAM catalog.

```
 BOGART:PROC OPTIONS(MAIN)
        DCL KSDS FILE RECORD KEYED DIRECT
                      ENV(VSAM PASSWORD('READ'));
        DCL PATH FILE RECORD KEYED SEQUENTIAL
                      ENV(VSAM PASSWORD('READPATH'));
        DCL PRFILE FILE RECORD SEQUENTIAL OUTPUT
                     ENV(/* OPTIONS DEPEND ON
                           OPERATING SYSTEM  */ );
        DCL 1 INREC,
              2 BOOKKEY   CHAR(6),
              2 NAME      CHAR(35),
              2 AUTHOR    CHAR(22);
        DCL EOFPATH BIT(1) INIT('Ø'B);

        /* THE NEXT THREE STATEMENTS ARE ON-STATEMENTS--SEE TOPIC 19 */

        ON KEY(KSDS) BEGIN;
                          PUT LIST  ('KEY CONDITION RAISED');
                          GOTO END;
                     END;
        ON KEY(PATH) EOFPATH='1'B;
        ON ENDFILE(PATH) EOFPATH='1'B;
        OPEN FILE(KSDS),FILE(PATH);
        READ FILE(KSDS) INTO(INREC) KEY('SFØØØ1');
        READ FILE(PATH) INTO(INREC) KEY(AUTHOR);
        DO WHILE(¬EOFPATH);
              WRITE FILE(PRFILE) FROM(INREC);
              IF ¬SAMEKEY(PATH) THEN LEAVE;
              ELSE READ FILE(PATH) INTO (INREC);
        END;

 END:   END BOGART;
```

The above program reads a record with key 'SF0001' from KSDS file and then prints out all records whose authors are the same as the author of 'SF0001'. Remember that when you access the path you are accessing the base KSDS in alternate index sequence. The alternate index would have been defined and built using AMS; the path would have been defined using AMS.

Some Further Options

The following are options of the ENVIRONMENT attribute for the file. There are other options, mentioned in the Language Reference Manual, but they are not important at this stage.

BKWD Option

This is used to process the file sequentially in **descending** key sequence. When a file with the BKWD option is opened the associated data set is positioned at the end.

SKIP Option

This option applies to a KSDS accessed via a SEQUENTIAL file. It should be used to improve performance whenever records are accessed by key but the access is mostly in ascending key sequence. When the SKIP option is specified VSAM attempts to retrieve records by key using the **sequence set** only. If the records are in ascending key sequence this will always be possible. If records are not in ascending key sequence then all levels of the index component will have to be used and access will take longer. (For DIRECT files VSAM always accesses all levels of the index component). It is never a mistake to specify the SKIP option but its incorrect use may slow down processing.

Exercises

1. Discuss the reasons for using a VSAM KSDS, ESDS or RRDS?
2. Is anything wrong with the declarations below?

```
DCL PASS CHAR(6);
DCL DUDLEY FILE DIRECT UPDATE
              ENV(GENKEY VSAM PASSWORD(PASS));
```

3. Suppose you wish to use the statement

```
READ FILE(MORE) INTO(AREA) KEY('X');
```

   to read the first record whose key **starts** with 'X'.

   What would you specify in the ENVIRONMENT attribute, and why?

   How would you declare the file and code the statement to read the next record after the one just read?

4. Is it possible to insert a record in a VSAM RRDS?

   Under what circumstances? Write the declaration of the file and the statement which would add a record to relative record slot 5.

5. An ESDS has been created (using AMS) containing transaction records for a whole week. At the last minute, some further records have to be added to the ESDS before processing it. Does the data set have to be recreat~~ing~~ed to incorporate these new records in a new extended data set? How else could the records be incorporated?

6. A KSDS consists of records with the following structure

```
        DCL 1 INSTR,
              2 EMPNO          CHAR(6),
              2 EMPNAME        CHAR(22),
              2 DEPCODE        CHAR(2),
              2 DEPARTMENT     CHAR(8);
```

   The prime key of the KSDS is EMPNO. An alternate index has been defined and built using AMS; the alternate key is DEPCODE. A path has been defined linking the base KSDS to the alternate index. Assume that the necessary JCL has been provided. Write the necessary declarations and statements to read all records which have DEPCODE of 'C1' and update the DEPARTMENT to 'COMPUTER'. Assume no password protection.

Answers

1. A **KSDS** is used in applications which process directly by embedded key and also sequentially in ascending/descending key sequence. INDEXED organization also provides this facility but VSAM KSDS organization has the advantage of not requiring frequent reorganization to maintain the initial performance. Thus a VSAM KSDS would be especially useful in applications in which there were many additions during the life of the data set.

   An **ESDS** is used in applications which mostly process the records in a sequential manner. CONSECUTIVE organization also provides this facility but an ESDS can additionally be processed by key (RBA), if desired, and records can also be added to the end of an ESDS without recreating it.

   An **RRDS** would be used to give fast direct access by relative record 'slot' within the data set without the time overhead of looking up an index (as in KSDS organization) nor the space overhead of storing such an index on disk.

2. GENKEY can only be used with a SEQUENTIAL file. The PASSWORD option has been used perfectly correctly; obviously the field PASS should contain the valid password character string before the file DUDLEY is opened.

3. GENKEY. Without GENKEY the key ('X') would be padded with blanks to the right for a length equal to the keylength of the file and then that particular key would be searched for. If the padded-out key were not there then the KEY condition would be raised. GENKEY stops the padding out of the key. Assuming no password protection, the file would be declared as

```
DCL MORE FILE SEQUENTIAL KEYED ENV(GENKEY VSAM);
```

   The statement to retrieve the next record after the one retrieved by generic key would be

```
READ FILE(MORE) INTO(AREA);
```

4. Yes, provided that the relative record slot into which you are trying to add the record is empty (either because you left the particular slot empty during creation or because a record has subsequently been deleted from that slot).

```
DCL RRDS FILE DIRECT UPDATE KEYED ENV(VSAM);
DCL RELRECNO   PIC'(8)9';
 .
RELRECNO = 5;
WRITE FILE(RRDS) FROM(AREA) KEY(RELRECNO);
```

5. The ESDS does not have to be recreated. Records can be added at the end of the ESDS as follows:

```
DCL ESDS FILE SEQUENTIAL UPDATE ENV(VSAM);
•
WRITE FILE(ESDS) FROM(NEWREC);
```

6.

```
DCL PATH FILE RECORD SEQUENTIAL KEYED UPDATE ENV(VSAM);
DCL EOFPATH BIT(1) INIT('Ø'B);
DCL 1 INSTR,
        2 EMPNO         CHAR(6),
        2 EMPNAME       CHAR(22),
        2 DEPCODE       CHAR(2),
        2 DEPARTMENT    CHAR(8);

ON KEY(PATH) EOFPATH = '1'B;          /* SEE TOPIC 19 */
ON ENDFILE(PATH) EOFPATH ='1'B;  /* SEE TOPIC 19 */

READ FILE(PATH) INTO(INSTR) KEY('C1');
DO WHILE(¬EOFPATH);
          DEPARTMENT='COMPUTER';
          REWRITE FILE(PATH);
          IF ¬SAMEKEY(PATH) THEN LEAVE;
          ELSE READ FILE(PATH) INTO(INSTR);
END;
```

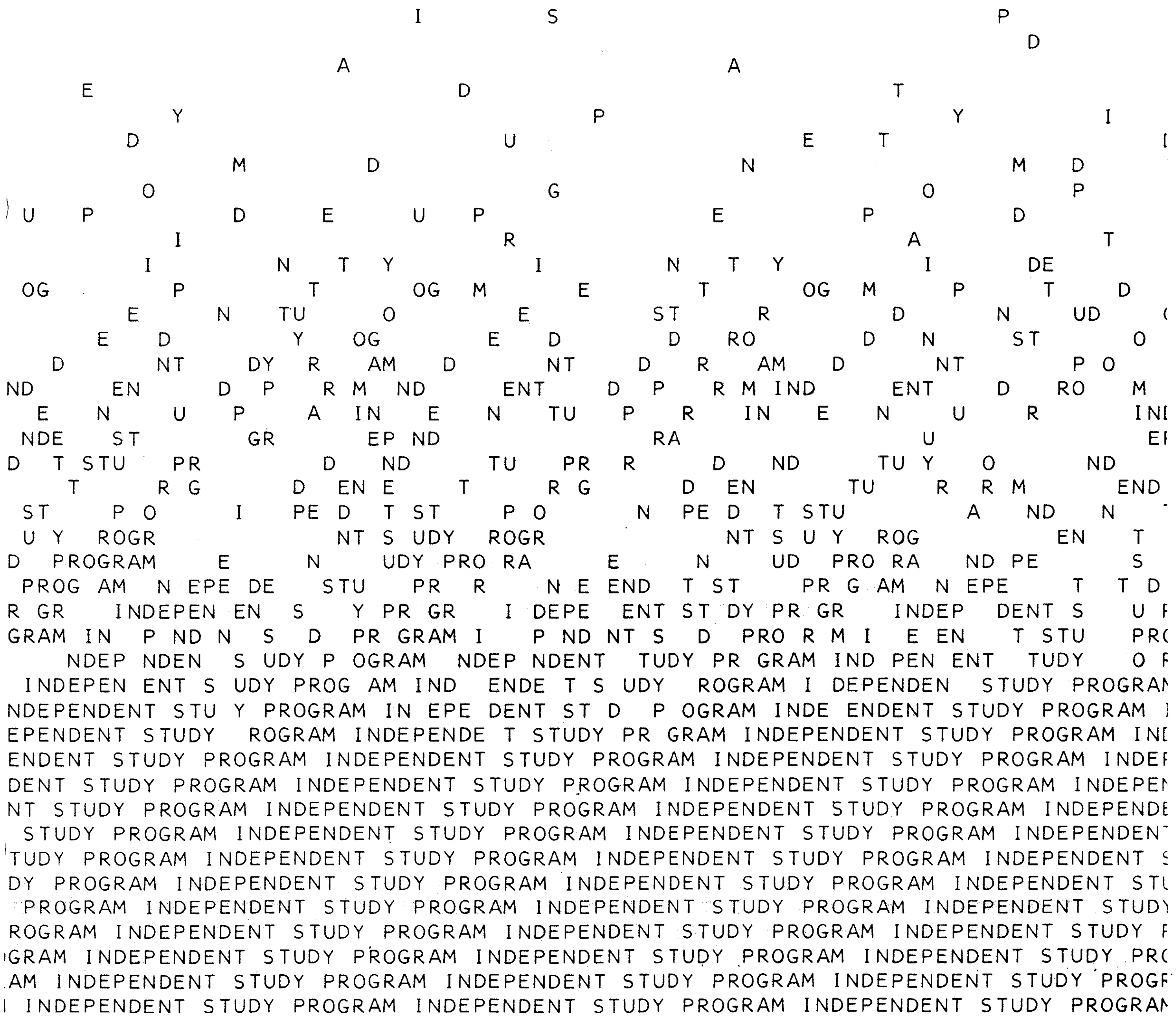
OS/VS JCL required,

```
//PATH DD DSNAME=VSAMPATH
```

DOS/VS JCL required,

```
//  DLBL PATH,'VSAMPATH',,VSAM
//  EXTENT SYSNNN,VOLID
```

Topic 14



Topic 14

Variable Length Records

In this topic you will learn about the various sorts of variable length records you are likely to meet, their format and methods of processing. You will also learn how to use variable length character strings.

Objectives

At the end of the topic you should be able to:

- describe the format of variable length records
- describe the creation of variable length records by using different length structures
- describe how the REFER option may be used to create and retrieve variable length records consisting of a fixed length header part followed by a variable number of fixed length trailer parts
- describe the implementation of VARYING character and bit strings
- describe the function of the SCALARVARYING option of the ENVIRONMENT Attribute.

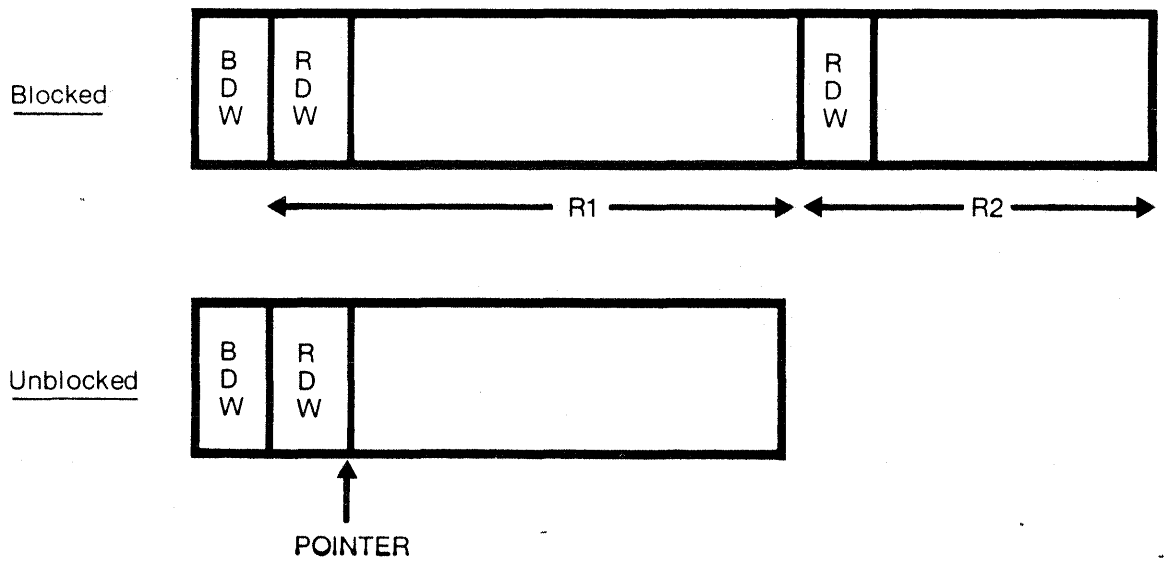
Introduction

Variable length records are used mainly for the purpose of saving external storage space. Their use, however, generally increases execution time because they must be created with control bytes which specify the size of the logical record and the length of the physical block. These control bytes are supplied by the system on output and interpreted by the system when the data set is subsequently read.

In theory, records which contain different sized fields could be handled as fixed length records by making each record the maximum possible size and ignoring or padding unused portions. This technique might even be practicable for data sets where the number of records and the range of records lengths are relatively small. However, in typical cases it is possible to make considerable external space savings by using variable length records. The significance of these savings increases with the range of record sizes, the number of records in the data set and the cost of the associated I/O device.

Record Formats

Variable length records can either be blocked or unblocked and their formats are shown in the following diagram.



The RECORD DESCRIPTOR WORD (RDW) is a 4 byte control field which precedes each record and contains information as to the length of the record.

The BLOCK DESCRIPTOR WORD (BDW) is a 4 byte control field at the beginning of each block and contains information as to the length of that block.

The control bytes which precede the 'data' portion of each logical I/O record are **never** included in the declaration of a structure (or element) which describes this record. During the creation of variable length records, the necessary control bytes are automatically prefixed to each logical record in the output buffer by compiler-generated code. Similarly, when these records are subsequently read into storage, the control bytes are transmitted but they are not made available to the programmer. Thus a READ SET statement causes the pointer to be set to the first **data** byte of a record in the input buffer, while a READ INTO statement will transfer only the data portion of a record from the buffer to the named variable.

Record / Block Size

Although control bytes are of no concern within the program, they must be specified in the maximum logical record size or block size in the ENVIRONMENT attribute or the DD card (OS/VS) as follows:

RECSIZE = Max DATA size + RDW

BLKSIZE = RECSIZE * (number of records) + BDW

In the next few sections we shall discuss specific ways of processing variable length records in PL/I. Normally they will appear in one of three different ways.

1. A multiple of record types, each type being different in length.
2. A record consisting of a fixed length header part and a variable number of fixed length trailer parts.
3. Records consisting of variable length character strings.

These ways will be considered separately.

The Use of Different Length Structures

One situation in which variable length records are used is where a data set contains multiple record types, each type being different in length. This can conveniently be handled by declaring a different structure to represent each type of record and specifying a WRITE or LOCATE statement for each structure.

For example, suppose a data set consists of two types of record:

Name and Address Records	-	80 bytes
Transaction Records	-	120 bytes

then we can declare a structure for each type:

```
DCL 1 NA DEF TRAN,              /* 8Ø BYTE N/A RECORD */
      2 TYPE_N CHAR(1),
      2 NA_DETAILS CHAR(79);

DCL 1 TRAN,                     /* 12Ø BYTE TRANSACTION RECORD */
      2 TYPE_T CHAR(1),
      2 TRANS_DETAILS CHAR(119);
```

Each record contains a field called TYPE which denotes whether it is a name/address record or a transaction record.

To write a name/address record we can code:

```
        WRITE FILE(NATRAN) FROM(NA);
```

and to write a transaction record:

```
        WRITE FILE(NATRAN) FROM(TRAN);
```

Records can subsequently be retrieved by moving them into the structure TRAN and then the record description used depends upon the value of the first byte as follows:

```
        READ FILE(NATRAN) INTO(TRAN);
        IF TYPE_N = 'N' THEN DO;
                               .
                               .
                               .
                               END;
        IF TYPE_T = 'T' THEN DO;
                               .
                               .
                               .
                               END;
```

Alternatively it is possible to create and retrieve records in LOCATE mode by a similar means. In this case the structure would have to be based as follows:

```
 DCL 1 NA BASED(P),
       2 TYPE_N CHAR(1),
       2 NA_DETAILS CHAR(79);

 DCL 1 TRAN BASED(P),
       2 TYPE_T CHAR(1),
       2 TRANS_DETAILS CHAR(119);
```

The creation and retrieval statements would be:

```
LOCATE TRAN FILE(NATRAN);

READ FILE(NATRAN) SET(P);
IF TYPE_N = 'N' THEN DO;
                     .
                     .
                     .
                     END;
IF TYPE_T = 'T' THEN DO;
                     .
                     .
                     .
                     END;
```

Record Condition

RECORD CONDITION was introduced in Topic 9 'Input and Output - further considerations'. Then we were dealing with fixed length records. The situations when it could arise for variable length records are as follows:

On input the work area is smaller than the stated record size.

On output the stated record size is smaller than the word area.

In other words, it is always possible to put something into somewhere larger than itself but not into somewhere smaller. In the latter case, the RECORD condition would be raised.

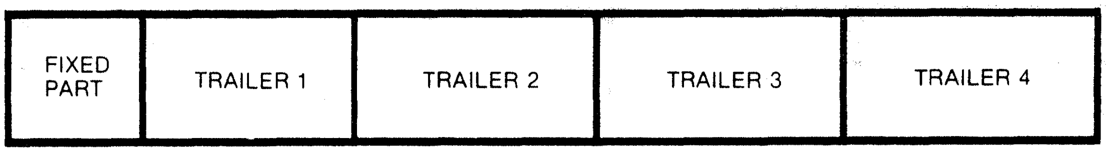
The technique of processing variable length records by using different length structures is relatively simple. It is not necessary for the programmer to hold the length of each record within the record itself: this can be established on the basis of record type.

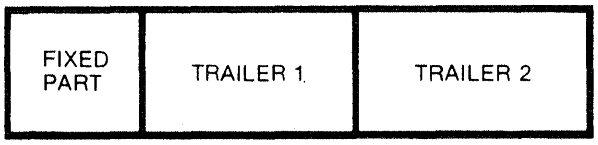
This technique is suitable for data sets containing records of relatively few different lengths.

Based Variable With the Refer Option

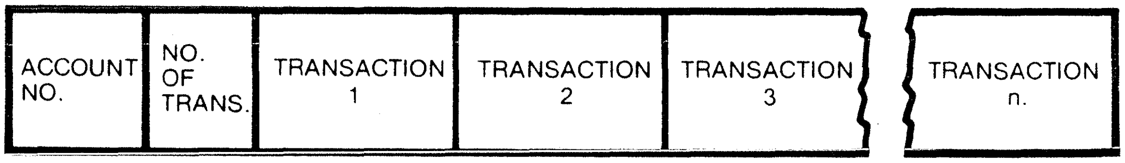
Self Defining Data

It is sometimes necessary for the programmer to create a variable length record in such a way that it contains an indication of its own length (in addition to the control bytes, which are inserted by the system and which are not normally accessed within a PL/I program). This kind of record is known as a **self-defining record.** The following diagram illustrates a typical example:





This represents a very common type of variable-length record; each record contains a fixed length header section followed by a fixed length trailer portion which is repeated a variable number of times. An example might be a record consisting of a customer's bank account number, followed by a variable number of transactions. To facilitate processing the programmer should create a field within the record which contains the number of transactions. For example:



Handling this type of record poses a problem. Consider a possible declaration of this record, assuming that the maximum number of transactions was 50.

```
DCL 1 C_TRAN,
      2 ACCOUNT CHAR(1Ø),
      2 NO_OF_TRANS FIXED DEC(3),
      2 TRANS(5Ø) CHAR(2Ø);
```

If MOVE mode input/output were used, then a possible record creation statement would be

```
        WRITE FILE(LEDGER) FROM(C_TRAN);
```

However this would always give fixed length records, each one being the maximum size of the structure C__TRAN. The alternative would be to have 50 structures each one having a different size for the array TRANS and then writing out from the appropriate structure. Obviously this involves a tremendous amount of coding and also of storage.

If LOCATE mode input/output were used the structure C__TRAN would have to be based on a pointer and the record created by means of the statement

```
         LOCATE C_TRAN FILE(LEDGER);
```

At this stage the pointer is pointing to the start of the data part of the record but no information has been put into the control bytes about the length of the record. In fact, the length may not even be known! There is no way in which PL/I can add this information later on. Hence using LOCATE mode, another problem arises i.e. there is no way of assigning values to the control bytes to reflect the lengths of records.

To overcome these problems, PL/I provides a feature called the REFER option and we will discuss it now.

The REFER Option

The REFER option is used in the declaration of a **based structure**: it specifies the length of a string variable or the bound of an array.

Let's declare a structure which describes the customer transaction record outlined above:

```
 DCL 1 C_TRAN BASED(P),
       2 ACCOUNT CHAR(10),
       2 NO_OF_TRANS FIXED DEC(3),
       2 TRANS (TRANS_CT REFER (NO_OF_TRANS)) CHAR(20);

 DCL TRANS_CT FIXED BIN(15);
```

TRANS is an array within the major structure C__TRAN; it describes the transactions made by this customer. It is not possible to declare explicitly the bounds of the array TRANS, because they are variable. Look closely at how they are declared:

TRANS (TRANS__CT REFER (NO__OF__TRANS))

TRANS is a one-dimensional array of which the upper bound is described by means of the REFER option which specifies two variables - one inside the structure CTRAN and the other outside. TRANS__CT is the variable which is declared **outside the structure.** In this variable the programmer keeps a count of the number of transactions relating to the current record.

Assuming that this count has a value, consider what happens when the programmer, in order to write out the record, issues the statement:

```
LOCATE C_TRAN FILE(LEDGER);
```

This statement allocates C__TRAN in the buffer and sets P to identify the position of this allocation. Data can now be assigned to C__TRAN. The number of transactions which occur in the record is taken from the variable in the REFER specification **which is declared outside the structure,** i.e. TRANS__CT. The implementation uses this value to calculate the logical record length which is automatically prefixes to C__TRAN in the buffer, i.e. **the RDW has been given a value.** Furthermore, the execution of the LOCATE statement causes **the value in TRANS__CT to be assigned to the variable NO__OF__TRANS.** This is the other variable used in the REFER specification and it is declared **inside the structure.** Thus we now have an indication of the number of transactions which occur in a record **within the record** - and this is provided **automatically** as a consequence of using the REFER option.

Notice that it is not possible to assign a value to NO__OF__TRANS before the LOCATE statement is executed because at that stage, it does not have any storage allocated to it. This is the whole point of specifying two variables in the REFER option. The variable **outside** the structure provides the information which determines how much storage in the buffer is to be associated with the current record; the variable inside the structure receives this information automatically when storage is allocated thus making the record self-defining.

To summarize, in order to create records of the self defining type, a value must be given to the upper bound of the array by means of the REFER option before issuing LOCATE and then building up the current record by assignment statements.

For example:

```
TRANS_CT = N;             /* CURRENT NO. OF TRANSACTIONS */
LOCATE C_TRAN FILE(LEDGER);
C_TRAN.ACCOUNT = CURRENT_AC_NO;
DO I = 1 TO NO_OF_TRANS;
   TRANS(I) = CURRENT_TRANS;
END;
```

This record will be physically written to the associated data set when the next LOCATE statement is issued (assuming unblocked records).

Record Processing

Now consider how the variable length records created above might subsequently be read and processed. The same structure (C__TRAN) used to create a record can be associated with a READ statement in order to retrieve the record. In this case the value for the upper bound of the array TRANS is taken from the variable NO__OF__TRANS **inside** the structure (i.e. from the record itself). Processing is simple because the programmer can interrogate the field NO__OF__TRANS directly to determine how many transactions there are (this is specified in that part of the record effectively overlaid by NO__OF__TRANS). Note that, on input, the value of TRANS__CT has no effect, and that the value of NO__OF__TRANS is not assigned to TRANS__CT. The programmer does not need to place a value in TRANS__CT before issuing the READ statement. This is necessary only before allocation in the output buffer.

Because of this it would be possible to dispense with the use of the REFER option in the structure named in the READ statement. For instance, the array TRANS in the structure C__TRAN could be declared with **fixed** bounds equal to the maximum number of transactions that could occur in one record (it is permissible to process a variable length record by means of a fixed length structure which is equal to the maximum possible record length). In this case the programmer can still obtain the length of the variable part of the record by interrogating the field NO__OF__TRANS.

We have now seen that, in order to create variable length records (of the type described in this section) by use of the REFER option, the programmer must:

1. Keep a count of the number of transactions relating to the current record in an independent variable (X).
2. Describe the variable length record by means of a structure which contains a field (Y) for holding the number of trailers in this record and which uses the REFER option. Y is linked to X by means of this REFER option.
3. Allocate the correct amount of space in the output buffer by means of a LOCATE statement. This statement causes the value in X to be assigned to Y.
4. Assign data to the buffer, using a DO loop (with Y as the control variable) in order to write the trailers.

On subsequent retrieval of each record the field Y can be interrogated to determine the number of trailers.

Varying Length Character Strings

The third type of variable length records are those which consist of variable length character strings. Before looking at processing these sorts of records we need to discuss the VARYING attribute.

The VARYING Attribute

We are familiar with the notion that string data is usually declared with a length specification. It is also possible to specify the VARYING attribute for both CHARACTER and BIT data. In this case a length specification is still normally supplied, but this is interpreted as a **maximum** length for the variable rather than a fixed length. For example:

```
 DCL X CHAR(1ØØ) VARYING;
```

This declares X to be a variable length character string with a maximum length of 100.

The compiler prefixed two bytes to a VARYING string at the time that storage is allocated for it. These two bytes are used to contain the **current length** of the string (this is maintained automatically during the program execution by compiler-generated code). The programmer does **not** have to include the two length bytes in the length specification when he declares a VARYING string. The current length of a VARYING string is taken to be the length of the data most recently assigned to it. For example:

```
 DCL X CHAR(1ØØ) VARYING;
 DCL A CHAR(25);
     X = A;
```

After this assignment, X will have, for practical purposes, a length of 25. Note that padding to the right with blanks does not occur (unlike assignments to fixed length strings). This is the chief characteristic of a VARYING string - it takes both the value and the length of the string assigned to it.

Let's consider another example - one which also illustrates that the result of concatenating a VARYING string with any other string (fixed-length or VARYING) is a VARYING string:

```
DCL NAME CHAR(2Ø) VARYING;
DCL (ADDR1,ADDR2,ADDR3) CHAR(2Ø) VARYING;
DCL NAME_ADDR CHAR(8Ø) VARYING;
   NAME = 'MR JONES ';
   ADDR1 = '13 HIGH ROAD ';
   ADDR2 = 'ANYTOWN ';
   ADDR3 = 'COUNTY';
   NAME_ADDR = NAME||ADDR1||ADDR2||ADDR3;
```

The field NAME__ADDR after the assignment has a practical length of 36 bytes and contains the string constant

'MR JONES 13 HIGH ROAD ANYTOWN COUNTY'

This is one of the advantages of using variable length character strings. If fixed length fields were used it would have been necessary to take substrings of the complete strings containing the relevant bytes and then concatenate them together.

LENGTH Built-in Function

Built-in functions are explained in detail in Topic 18, 'Subroutines and Functions'. One of them is the LENGTH built-in function which is useful for obtaining the current length of a VARYING string. For example, the effect of coding

```
         I = LENGTH(NAME_ADDR);
```

would be to assign the value 36 to the variable I. The LENGTH built-in function returns the current length of the specified string as a fixed binary integer.

The SCALARVARYING Option

We have seen that the compiler adds two length bytes to the beginning of a VARYING string but that the programmer need not be concerned with these within his PL/I program. Remember that it is **never** necessary to allow for the two length bytes when declaring a VARYING string.

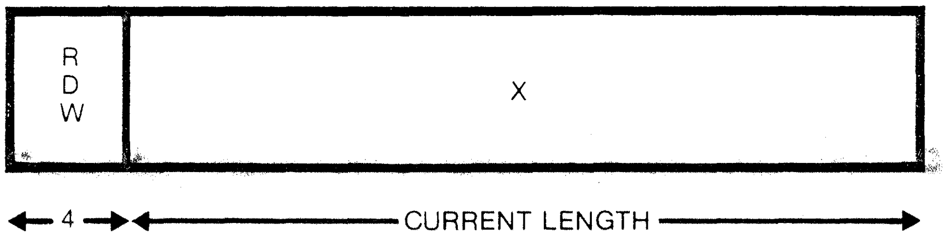
The programmer can, however, specify that these two bytes are to be included in the transmission when writing a record from a VARYING string. This is necessary when using locate mode output statements.

In order to include the length prefix in the transmission of a VARYING string, the programmer must specify the **SCALARVARYING option** in the ENVIRONMENT attribute when declaring the associated file. Let's now consider the effect of this option, first with respect to move mode I/O.

Suppose that a programmer creates a variable length record in a VARYING string called X and transmits this record by means of a file called F as follows:

```
DCL X CHAR(1ØØ) VARYING;
DCL F FILE RECORD OUTPUT ENV(V ........);

/* ASSIGN VALUE TO X */

WRITE FILE(F) FROM(X);
```

Then the record would be transmitted without the length prefix in the following format:

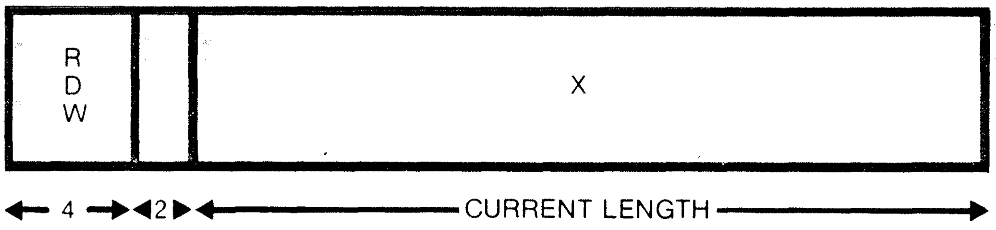


In this diagram the field RDW represents the logical record length which is automatically constructed by data management (this field contains the 'control bytes' that we discussed earlier).

Now suppose that the programmer specifies the SCALARVARYING option in the file declaration, viz:

```
DCL F FILE RECORD OUTPUT ENV(V, SCALARVARYING ..........);
```

This would cause the VARYING string to be transmitted **with its length prefix,** in the following format:



For subsequent retrieval of this record, the SCALARVARYING option must be specified in the associated file declaration. This informs the system that the first two 'data' bytes of the record constitute a length prefix, and are to be interpreted as such for execution of the READ statement, which must transmit the record into a VARYING string field.

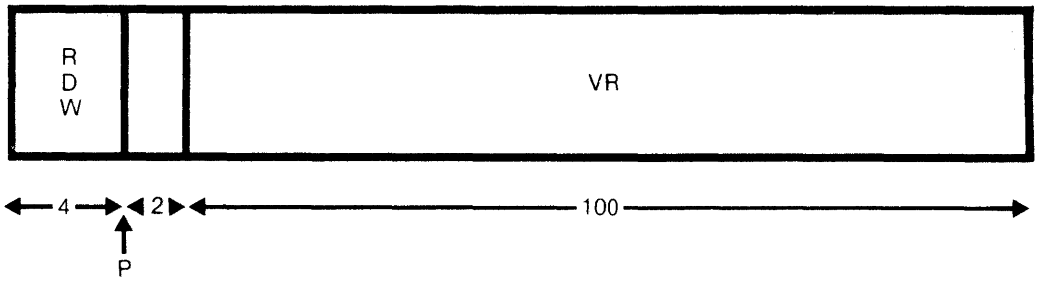
Note that if SCALARVARYING is not specified when reading records into a VARYING string (thus indicating that the records do not contain a length prefix), the system is able to calculate the length prefix to be appended to the VARYING string in each case from the contents of the RDW field. Thus for MOVE mode processing the SCALARVARYING option is redundant. It just duplicates the information given in the Record Descriptor Word.

The SCALARVARYING option is primarily intended for locate mode I/O, where it **must** be specified. This is a consequence of the fact that locate mode causes **the maximum length of the VARYING string** to be transmitted. This is because the LOCATE is executed before any values have been assigned to the output string and hence there is no indication as to what the length of the string will be.

Consider the following example:

```
DCL F FILE RECORD OUTPUT ENV(V,SCALARVARYING ........);
DCL VR CHAR(1ØØ) VARYING BASED(P);
LOCATE VR FILE(F);
```

Here the transmission of the length prefix associated with the string VR is specified by means of the SCALARVARYING option. The effect of the LOCATE statement would be to allocate enough storage in the output buffer to accommodate the maximum size of VR and to set P to the beginning of the length prefix:



After each LOCATE statement data can be assigned to VR, thus constructing the current record and setting the corresponding value of the length prefix. If the current record does not fully occupy the allocated space, the unused positions are transmitted with the record. It is for this reason that the length prefix must be included in the transmission: this prefix enables the system to determine what portion of the record contains genuine data on subsequent retrieval. Note that the transmission of a VARYING string by means of the LOCATE statement effectively creates fixed-length records. Therefore this technique should only be used in special cases.

Summary

Variable length records are used to save external storage space. We have discussed three basic types of variable-length records in this topic. They are:

1. Multiple record types, each type being a fixed length. These can be created by writing to the data set from different length structures (each one corresponding to a different record type).
2. Records which contain a fixed length header section followed by a fixed length trailer section which is repeated a variable number of times. This type of record may be conveniently created by using the REFER option to describe the number of occurrences of the trailer section in each record.
3. Records in which all or several fields are variable. These may be created by concatenation of a series of VARYING strings. If these are to be outputted using LOCATE mode processing then the SCALARVARYING option of the ENVIRONMENT must be used.

Exercises

1. (a) What is the main advantage of using variable-length records?

   (b) Why does the use of variable-length records generally increase execution time?

2. 'The control bytes which the system prefixes to a variable-length record to describe its length need **never** concern the programmer.' Explain why this is not true.

3. A data set is to contain three types of records - one 65 bytes, one 75 bytes, and the remaining one 100 bytes. Suggest a suitable method for creating this data set.

4. What is meant by a self-defining record?

5.

```
DCL 1 C_TRAN BASED(P),
      2 ACCOUNT CHAR(1Ø),
      2 NO_OF_TRANS FIXED DEC(3),
      2 TRANS(TRANS_CT REFER(NO_OF_TRANS)),
        3 TYPE CHAR(1),
        3 ITEM CHAR(15),
        3 AMT FIXED DEC(7);
DCL TRANS_CT FIXED DEC(3);

READ FILE(IN) SET(P);
```

   Write statements which will examine the TYPE field in every transaction and, where this field contains the character 'A', increase the AMT field by the value 2.

6. The maximum possible size of a variable-length record is 100. Records are constructed by concatenation of VARYING strings.

   Declare a VARYING string called VARY in which the programmer can build each record, and state what maximum logical record length must be specified in the ENVIRONMENT attribute (or DD card) when the data set is created (or retrieved).

   Do this

   (a) when SCALARVARYING is specified.

   (b) when SCALARVARYING is not specified.

Answers

1. (a) The saving in external storage space.

   (b) Control bytes must be created and prefixed to a variable length record by the system.

2. They must be allowed for when specifying the maximum logical record size and the block size.

3. Declare three different structures, one for each type of record.

4. One which contains information about its own fields (e.g. length) within itself.

5.

```
 DO I = 1 TO NO_OF_TRANS;
      IF TYPE(I) = 'A' THEN AMT(I) = AMT(I) + 2;
 END;
```

6. (a)

```
 DCL VARY CHAR(1ØØ) VARYING;
```

Logical record length = RDW + length prefix + VARY
= 106

(b)

```
 DCL VARY CHAR(1ØØ) VARYING;
```

Logical record length = RDW + VARY
= 104

Note that the programmer does not need to allow for either the system control bytes, or the length prefix when declaring VARY.

Topic 15

Topic 15

Stream INPUT/OUTPUT

As well as record-oriented input/output, PL/I supports an entirely different approach, namely stream-orientated input/output. This is useful for the input of test data, and the debugging of programs and also provides ease of control of layout of printed output. This topic describes the declarations of files for stream-orientated input/output and the basic input/output statements.

Objectives

At the end of this topic you should be able to:

- code valid I/O statements - using the standard system files or programmer defined files - for all three types of STREAM I/O
- describe when the ENDPAGE and NAME conditions arise
- describe the function of the COUNT, LINENO, DATAFIELD built-in functions.

Introduction

Stream-oriented input/output regards input and output data sets as a continuous stream of character information without record boundaries and thus can handle only consecutively organized data sets. The variables which are to receive input, or from which output is to be transmitted, do not have to be in adjacent parts of a structure or array. Any necessary conversion between the types of the variables and the character format of the data held externally will be done automatically on input or output. The input information will be distributed to the various variables which are to receive it, and the output information will be collected from the various variables from which it originates. Because of the conversion during input/output, stream input/output is less efficient in terms of execution time and is not recommended for normal production programs.

File Declarations

Explicit

Files can be declared explicitly as stream by adding the attribute STREAM in the declaration statement, i.e.

```
 DCL STR STREAM OUTPUT ENV(·······);
```

Note that STREAM is the alternative to RECORD and that if neither are implied or specified then STREAM is the default.

Contextual

If the file referred to in a stream input/output statement is not declared elsewhere then it is assumed to have the STREAM attribute, i.e.

```
 PUT FILE(STR) LIST(X);
```

Implicit

Both DOS/VS and OS/VS have standard system input and output files. These are called SYSIN and SYSPRINT respectively. If a filename is omitted in a GET statement (a stream input statement) then it is assumed that SYSIN is being used and similarly, for a PUT statement (a stream output statement), SYSPRINT is assumed. The default attributes are as follows:

SYSIN (DOS/VS): FILE STREAM INPUT EXTERNAL
ENV(F RECSIZE(80) MEDIUM(SYSIPT))

SYSIN (OS/VS): FILE STREAM INPUT EXTERNAL
ENV(F RECSIZE(80))

SYSPRINT (DOS/VS): FILE STREAM OUTPUT PRINT EXTERNAL
ENV(F RECSIZE(121) MEDIUM(SYSLST))

SYSPRINT (OS/VS): FILE STREAM OUTPUT PRINT EXTERNAL
ENV(VB RECSIZE(129))

The RECSIZE for SYSPRINT includes one character for a CTLASA control character. This is automatically supplied when transmission statement options (see later) are included in the output statement.

Stream I/O Types

There are three different types of stream oriented input/output. These are mentioned below along with their uses.

DATA-directed

Used mainly for testing and debugging. It provides an easy method of entering test values into a program and for obtaining the value of variables at points throughout the program.

LIST-directed

As for DATA, but is also useful for jobs requiring a small amount of output where the layout of the print file is not important.

EDIT-directed

Used when more control is required over the layout of stream files (particularly print files) and also the layout of initial input of data into a program from punched cards.

If the type of input/output is not specified, the LIST is the default.

LIST-directed I/O

General

When record-oriented input is used, the programmer is required to provide an input area (or based variable) which precisely matches the incoming data-records. List-directed input relieves him of this duty: the statement

```
 GET LIST(VAR);
```

will obtain the next data item from the input stream and assign it to VAR, converting it to the attributes and precision of VAR automatically.

It is, therefore, only necessary for the user of list-directed input to know **which** variables have their values provided in the stream and, if there is more than one variable whose value appears, **in what order** the input stream is arranged.

List-directed output is also greatly simplified. Zero-suppression and formatting are performed automatically, according to predefined rules, and the programmer is charged with the sole duty of ensuring that his output is intelligible.

LIST Directed Input

The data items in the input stream must be character strings in the form of optionally signed valid constants. These strings are separated by commas, one or more blanks, or both. String constants are enclosed in quotes in the input stream, just as they would be written within a program.

Example:

```
-1.4,6.3,-9,8,'ALFA','BETA'
-1.4 6.3 -9 8 'ALFA' 'BETA'
-1.4     6.3     -9      8    'ALFA' 'BETA'
-1.4   ,  6.3   ,  -9    ,  8 ,  'ALFA','BETA'
```

The above four sample streams are all equivalent.

Data items are assigned to the variables specified in an input data list, in sequence; each variable in the list is assigned the subsequent item from the stream until the list is exhausted. The stream is then 'positioned' at the next data item, in preparation for the next GET.

```
 GET LIST(VAR1,VAR2);
```

This statement will, in association with one of the input streams in the above example, cause -1.4 to be assigned to VAR1 and 6.3 to VAR2.

If the next input statement were

```
 GET LIST(VAR3,VAR4);
```

then -9 would be assigned to VAR3 and 8 to VAR4.

Finally,

```
 GET LIST(VAR5,VAR6);
```

would assign 'ALFA' to VAR5 and 'BETA' to VAR6. (VAR5 and VAR6 would have to be declared as character string variables).

Data List

Although in the above example, VAR1 to VAR6 are element variables, other possibilities are unsubscripted/subscripted array names or structure names.

Example:

```
 DCL 1 STR,
       2 A CHAR(5),
       2 B FIXED DEC(3);
 GET LIST(STR);
```

Assuming that the input stream contains:

'ABCDE',900,....

the result of the above GET would be to perform the assignments:

```
A = 'ABCDE'
B = 900
```

Note:

If the data items in the input stream are not of the same data type or precision as the variables to which they are to be transmitted, conversion will automatically take place, as necessary. This is true for all three types of stream-oriented transmission.

LIST-directed Output

As has previously been mentioned, in Topic 9 on 'Input and Output'; the PUT statement is used for stream output. The data items are converted to character form and placed into the output stream, separated by blanks.

When, as is usual, stream output is used for PRINT files, data items are automatically aligned on present tab positions, which are generated by software. (These are at positions 1, 25, 49, 73, 97 and 121 on each line. The positions may be altered if required; the details are in the Programmer's Guide).

Example:

Suppose X, Y and Z are arithmetic variables having the current values 2.6, 6.3 and 5.8 respectively.

```
 PUT LIST(X,Y,Z);
```

This statement will cause

```
2.6     6.3     5.8
```

to be printed.

Example:

```
 ON KEY BEGIN;
          IF ONCODE=51 THEN
             PUT LIST ('RECORD NOT FOUND,KEY=', ONKEY);
        END;
```

If the key condition were raised and the value of ONKEY were 10 (say) then the following would be printed:

RECORD NOT FOUND,KEY= 10

DATA-Directed I/O

General

Data-directed I/O resembles List I/O, except that not only do the values of variables appear in the input or output stream, but the **names** of the variables are also present. By using this technique, the programmer need not be aware of the order of data items in the input stream, since each of these takes the form of a self-identifying assignment (e.g. VAR1=15 might appear in the input stream; 15 **on its own** is **not** permitted as a data-directed input item).

When data is output by data-directed I/O, the variables are automatically transmitted in the form of assignment statements.

DATA-directed Input

The input statement is similar in format to that for list-directed input.

```
 GET DATA(A,B,C);
```

Data List

The items within the parentheses are known as the data list and must be one of the following:

ELEMENT, UNSUBSCRIPTED ARRAY NAME,
STRUCTURE NAME, A DEFINED VARIABLE

The data list can be omitted.

Data Stream

The data in the input stream is in the form of assignment statements and a typical input stream corresponding to the above GET statement might be

```
A=3,B=4;A=6,B=2,C=7;A=6;
```

where semicolons indicate the limit of assignments to be performed by a single GET statement. Thus on the first execution of the GET statement the following assignments will be made:

```
A=3
B=4
```

on the second

```
A=6
B=2
C=7
```

and on the third

```
A=6
```

Notes:

1. Each assignment can be separated either by one or more blanks and/or a comma.
2. Assignment statements need not be in the same order as the items in the data list.
3. Assignments need not be made to all items in the data list. In this case, those not appearing in the stream remain unchanged.

4. In the data stream subscripted array names may be used.

Example:

```
DCL AR(1Ø) FIXED(2);
GET DATA(AR);           /* IS PERMITTED */
GET DATA(AR(1));        /* NOT PERMITTED */
```

A possible valid input stream is

```
AR=1;AR=2;AR(1)=9;AR(9)=5;
```

(The first two assignments are assignments to every element of the array AR).

Note that repeated execution of the valid GET DATA statement will read in all of the input stream even though some of the data refers to elements of the array while the rest refers to all of the array.

It should also be noted that any qualified name which appears in the input stream must be **fully** qualified. Partially qualified names are, however, permitted in data-lists (both for data-directed and list-directed I/O). Example:

```
DCL 1 STR,
      2 A,
        3 B FIXED(2),
        3 C FIXED(2);
```

Valid alternative GET statements:

```
GET DATA(B);
GET DATA(STR.B);
GET DATA(STR.A.B);
```

Valid input stream:

```
STR.A.B=52;
```

Invalid input streams:

```
STR.B=52;
    B=52;
```

NAME Condition

If in the input stream there is an assignment to a field not mentioned in the data list, then the NAME condition will be raised.

Example:

```
 GET DATA(A,B);
```

with a corresponding input stream:

```
A=6,B=7,C=8;
```

DATAFIELD Built-in Function

If the standard system action is not required then a suitable on-unit should be coded. The DATAFIELD built-in function is specifically designed to help deal with the NAME condition.

```
 ON NAME BEGIN;
           PUT LIST(DATAFIELD);
          END;
```

This on-unit, if executed for the above example will produce the output

```
C=8
```

Note that if the data list is omitted on a GET statement then the NAME condition cannot be raised.

DATA-directed Output

In the normal case, when the output is used for PRINT files the output, which is in the form of assignment statements, is aligned on the preset tab positions, as mentioned with LIST output.

Example:

```
 PUT DATA(A,B,C);
```

would cause:

```
A=2,B=9,C=12;
```

to be printed (assuming these are the correct current values).

Data List

As for input with the addition of subscripted array names. The data list can be omitted in which case all variables would be printed in the above format. This is uneconomical both in time and paper and should only be used for debugging very small programs.

Data in the stream

As shown above, in the form of assignment statements terminated by a semicolon. Array elements are subscripted and printed in row major order. Structures are printed element by element and are fully qualified.

Example:

```
DCL AR(2,2) FIXED (1) INIT(1,2,3,4);
DCL 1 STR,
      2 MIN1,
        3 A FIXED(1) INIT(2),
        3 B FIXED(1) INIT(7),
      2 MIN2 FIXED(1) INIT(6);

PUT DATA(AR,STR);
```

will cause the following to be printed:

```
AR(1,1)=1         AR(1,2)=2         AR(2,1)=3         AR(2,2)=4
STR.MIN1.A=2      STR.MIN1.B=7      STR.MIN2=6
```

EDIT-Directed I/O

General

Neither DATA nor LIST directed stream input/output allows the programmer any control over the formatting of the input data or the layout of the output data. EDIT directed stream input/output gives the programmer this control and thus provides a more flexible form of stream transmission.

General format

The general format of the edit directed input statement is:

GET FILE(filename) EDIT(data-list)(format-list);
(PUT will replace GET in the output statement).

For the remainder of this section the FILE option will be omitted, thus implying

```
         FILE(SYSIN)      for input
    and  FILE(SYSPRINT)   for output
```

because these are the files with which all types of stream-oriented transmissions are most commonly used.

In the statement

GET EDIT(data-list)(format-list);

the data-list is a list of variables, whereas the format-list is a set of instructions which describes how these variables are to be obtained from the input stream. The input stream itself is a **continuous stream of characters:** variable names are not included, as they are in data-directed input streams, nor are commas or blanks required, as for list-directed input.

Example:

Supose each card in the input stream is to contain the values of eight variables (A,B,C,D,E,F,G,H) in the form of ten-digit numbers. The first columns might contain

```
     128769889761342678962863 1......

so that   1287698897 is the value of A
and       6134267896 is the value of B, and so on.
```

Notice how there is nothing in the input stream itself to say how the figures are to be interpreted. That is specified by the format list of the input statement:

```
 GET EDIT(A,B,C,D,E,F,G,H)
         (F(1Ø),F(1Ø),F(1Ø),F(1Ø),F(1Ø),F(1Ø),F(1Ø),F(1Ø));
```

which may be abbreviated to:

```
 GET EDIT(A,B,C,D,E,F,G,H) (8 F(1Ø));
```

The F(10) means 'ten characters interpreted according to the F format item'. (The F format item means that the characters are the representation of fixed decimal values; see under the heading 'The Format Items' below).

Observe that each item in the data-list is associated with the item in the corresponding position of the format list, and that the repetition factor 8 is written with at least one space separating it from the format item F(10). Suppose that variable A occupied only the first three columns of each card and variables B through H each occupied eleven columns. Then we should code:

```
 GET EDIT(A,B,C,D,E,F,G,H) (F(3),7 F(11));
```

On output, the format item has a slightly different meaning:

```
 PUT EDIT(X,Y,Z) (3 F(5));
```

This statement means 'put X, Y and Z into the output stream (i.e. on to SYSPRINT), placing X into the next five character positions, Y into the 5 following character positions and Z into the 5 positions after that; before placing them into the stream, however, convert them to the character representation of fixed decimal values and perform any rounding or zero-suppression that is required; also supply a minus sign if necessary.'

The Format Items

The F Item

The general format is

F(w,d,p)

where w specifies the total number of characters in the field, d specifies how many are to the right of the decimal point (d = 0 is assumed if d is omitted), and p is a scaling factor (p is rarely used). It effectively multiplies the value of the corresponding data item by 10 to the power p on output, and multiplies the item in the data stream by this amount on input).

Decimal points for input data may be specified in an F item, as above, or they may actually appear as characters in the input stream. If they are indicated in both of these ways, **the specification in the F item is overridden.**

Input

On input the data stream can consist of decimal digits with optional sign, decimal point and leading and trailing blanks. Any other character will produce an error.

Example

Format Item	**Contents of External Data Field**	**Value Assigned**
F(4,2)	ƀ123	1.23
F(4,2)	1235	12.35
F(4,2)	-123	-1.23
F(4,2)	1.23	1.23
F(4,4)	12.3	12.3
F(4,4,2)	12.3	1230.0
F(4)	ƀ123	123.0
F(4)	123ƀ	123.0
F(4)	-123	-123.0
F(4)	1.23	1.23
F(4)	12.3	12.3

The target variable, be it DECIMAL FIXED/FLOAT or BINARY FIXED/FLOAT must be large enough to receive the value read. Low order digits will be truncated, or low order zeros may be added, as necessary. High order zeros will be added if necessary, but if high order significant digits are truncated, the result will be unpredictable.

Output

When a PUT statement uses the F item, the field width (w) is the number of character positions of the output stream which will be filled; since a decimal point and minus sign will be supplied if necessary, space should be left for them where they occur.

Example:

Format Item	**Value to be Printed**	**Output Produced**
F(8,2)	0.1	ƀƀƀƀ0.10
F(8,2)	-123.456	ƀ-123.46
F(8,2)	101B	ƀƀƀƀ5.00
F(8,2,2)	101B	ƀƀ500.00
F(8,2)	0	ƀƀƀƀ0.00
F(8)	0.1	ƀƀƀƀƀƀƀ0
F(8)	0.9	ƀƀƀƀƀƀƀ1
F(3,1)	-123.456	3.5

Note that rounding is automatic.

The A Item

The general format is

A(w)

Input

The A format item is used for input to character variables. It specifies that the next w characters in the input stream may be any characters, and are to be assigned to the next variable in the data list, e.g.

```
 GET EDIT(EL1) (A(3));
```

This will take the three characters immediately after the last character read (assuming that it is not the first GET statement) and assign them to EL1. If the length of the string read, and the length of the variable do not match, padding or truncation will take place as in assignments.

Example

```
 GET EDIT(MONTH,DAY,YEAR) (A(3),F(2),A(4));
```

Input stream: JAN101976

This has the same effect as the assignments

MONTH	=	JAN
DAY	=	10
YEAR	=	1976

Output

The A format item causes the data list associated with it to be converted to a character string of length w, with truncation or padding on the right hand side, if necessary, and to be copies to the output stream. **The A may be specified without the field width in which case it is taken to be equal to the length of the character string specified in the data list.**

Example:

Format Item	**String to be Printed**	**Output Produced**
A(7)	ALPHA_1	ALPHA_1
A(5)	ALPHA_1	ALPHA
A	ALPHA_1	ALPHA_1
A(9)	ALPHA_1	ALPHA_1ƀƀ
A(6)	BILL'S	BILL'S

The omission of the field width in the A format is useful when a character string constant, as opposed to a character string field, is to be printed. The text will be printed as it appears in the data list, with pairs of quotes within the string reduced to single quotes, i.e.

```
 PUT EDIT('MANAGER''S REPORT') (A);
```

The output from this statement will be

```
MANAGER'S REPORT
```

The P Item

Input

The P item is used to specify ordinary picture editing for stream I/O.

On input, the P item can be used in two ways: to check that an item in the input stream satisfies a specified description, or to define how the arithmetic value of a numeric character string is to be interpreted. The picture characters used are no different from those which appear in DECLARE statements.

Example

Input stream: $56.78923....

Suppose the above stream represents cash amounts followed by customer numbers ($56.7,8923). An appropriate way of reading it in would be as follows:

```
DCL CASH PIC '$99V.9';
DCL CUST CHAR(4);
GET EDIT(CASH,CUST) (P'$99V.9',P'9999');
```

Notice that CUST could have been assigned a value using the A format item but then there would have been no check to ensure that all digits were numeric. If the input stream had contained 89Z3 instead of 8923 the CONVERSION condition would have been raised.

Notice also that we need not have kept the character representation of CASH within the program. CASH would contain the correct arithmetic value if it were declared thus:

```
DCL CASH FIXED DEC(3,1);
```

(The GET statement would not need changing).

Output

On output, the P item enables the programmer to specify picture editing. This is particularly useful for numeric fields.

Example:

```
DCL AMOUNT FIXED DEC(9);
AMOUNT = 562;
PUT EDIT(AMOUNT) (P'$$$ZZZZZZ9');
```

The resultant printout would consist of the following 10 characters:

ƀƀ$ƀƀƀƀ562

Observe how the $ floats and how the zeros are suppressed.

The E, C and B Items

These items are not going to be considered in detail, since their use is rather specialized. Further information can be retrieved from the Language Reference Manual (Section E). It is sufficient to say, at the moment, that E is used for floating point numbers, C for complex numbers ('complex' in its mathematical sense) and B is used for bit strings.

Control Format Items

These are items which allow part of the input stream to be ignored or facilitate the control of printed layout.

The X Item

The X item is used to specify that a number of characters in the input stream are to be ignored, or that a number of blanks are to be inserted into an output stream.

Thus, if the input stream is

```
5813263......
```

the statement

```
 GET EDIT(VAR) (X(3),F(2));
```

has the same effect as the assignment

```
VAR=32;
```

If we say

```
 PUT EDIT(VAR) (X(4),F(2));
```

this specifies that the next six characters of the output stream are to contain four blanks, followed by the value of VAR as two digits.

The SKIP Item

The SKIP format item goes against the normal rule that stream input/output ignores record boundaries. On input a record is a card and on output, it is a line. The general format is:

SKIP(n)

Input

On input it causes the remainder of the current record and also the next n-1 records to be ignored and the next item will be read from the nth record after the current one.

Example:

```
GET EDIT(EL,AR(1,2)) (A(3),SKIP(2),X(4),F(7,2));
```

This statement will cause the next three characters in the input stream to be read to EL, the remainder of the current record, the whole of the next record and the first four characters of the record after that to be ignored, and the next seven characters to be read to AR(1,2).

Output

Note that the instruction:

```
PUT SKIP;
```

has the same meaning as:

```
PUT SKIP(1);
```

Similarly with output, the statement:

```
PUT SKIP(n);
```

causes the rest of the current line to be filled with blanks, the printing of n-1 blank lines and the next item will be printed at the **beginning** of the nth line after the current one.

Example:

```
PUT EDIT(TOTAL,PAGE) (SKIP,A,SKIP(2),F(2));
```

will print the TOTAL field at the beginning of a line and the PAGE field at the beginning of another line with a blank line between.

The PAGE Item

The general format is:

PAGE

The PAGE format item is only appropriate to files with the PRINT attribute and causes a skip to the start of a new page.

Example:

```
 PUT EDIT('PAGE HEADING') (PAGE,A);
```

will cause PAGE HEADING to be printed at the start of the first line of a new page.

The COLUMN Item

The general format is:

COLUMN(n) or COL(n)

Input

The column format item causes scanning to recommence at column n of the current record. If that column has been passed, then the rest of the current record will be ignored and scanning will recommence at column n of the next record.

Example:

```
 GET EDIT(EL,AR(1,2)) (A(3),SKIP(2),COL(5),F(7,2));
```

will have the same effect as the example given for the SKIP format item.

Output

Similarly with output, the COLUMN format item causes spacing to the nth character position on the current line, or the following line if that position is passed.

Example:

```
 PUT EDIT(TOTAL,PAGE) (SKIP,COL(3),A,COL(2Ø),F(2));
```

will print out the TOTAL field starting at column 3 of a new line and the PAGE field starting at column 20 of the same line.

The LINE Item

The general format is:

LINE(n)

The LINE format is also only appropriate to files with the PRINT attribute. It causes a skip to the beginning of the line specified. If that line on the page has not been passed, skipping will be to that line on the same page. If it has been passed, skipping will be to that line on the next page.

Example:

```
 PUT EDIT('PAGE HEADING') (PAGE,LINE(5),COL(54),A);
```

will cause PAGE HEADING to be printed in the center of the fifth line of a new page.

Data List and Format List Length

The number of items in the data list and the number of data format items in the format list need not be the same. The stream input/output operation is controlled by the data list. As each item in the data list is processed, the format list is scanned until the next data format item is found. This is associated with the data list item, and all control format items between the last data format item and the current one will be acted on. When the data list is exhausted, **any remaining format items will be ignored,** including any control format items. Thus:

```
 PUT EDIT('MESSAGE') (A,SKIP,COL(30),A);
```

will produce identical output to:

```
 PUT EDIT('MESSAGE') (A);
```

In the first example, 'MESSAGE' will use the first A format item, and the rest of the format list will be ignored. It will not cause any error.

If there are insufficient data format items in the format list, the format list will be re-used. Re-using the format list will cause no special effects on input or output, such as skipping records. The statement will be executed as if the format list has been repeated an indefinite number of times. When the data list has been exhausted, the rest of the format list will be ignored.

Thus:

```
 GET EDIT(W,X,Y,Z) (A(5),A(2));
```

will have identical effect to:

```
 GET EDIT(W,X,Y,Z) (A(5),A(2),A(5),A(2));
```

Iteration within Format Lists

It is often necessary, when processing several items with similar attributes, to repeat items or groups of items in format lists, as in the last example.

In the example:

```
DCL X FIXED DEC(3);
DCL 1 STR(3),
      2 FLDA CHAR(3),
      2 FLDB (2) FIXED DEC(5,2);
GET EDIT(X,STR) (F(3),A(3),F(5,2),F(5,2),A(3)
                 F(5,2),F(5,2),A(3),F(5,2),F(5,2));
```

the group of items A(3), F(5,2), F(5,2) appears three times, and in each repetition, the item F(5,2) appears twice. Coding may be simplified by using iteration factors, i.e.

```
GET EDIT(X,STR) (F(3),A(3),2 F(5,2),A(3),2 F(5,2)
                 A(3),2 F(5,2));
```

and even further by:

```
GET EDIT(X,STR) (F(3),3 (A(3),2 F(5,2)));
```

Remote Format Lists

When many input/output statements share the same format specifications, there is no need to recode that specification each time. It can be coded as a separate format statement and then referenced within the input/output statement.

Example:

Instead of coding

```
 PUT EDIT(A,B,C) (F(3),SKIP,F(5,2),PAGE,A(6));
 PUT EDIT(P,Q,R) (F(3),SKIP,F(5,2),PAGE,A(6));
 PUT EDIT(X,Y,Z) (F(3),SKIP,F(5,2),PAGE,A(6));
```

we may code

```
 PUT EDIT(A,B,C) (R(CMN));
 PUT EDIT(P,Q,R) (R(CMN));
 PUT EDIT(X,Y,Z) (R(CMN));
 CMN: FORMAT(F(3),SKIP,F(5.2),PAGE,A(6));
```

(The FORMAT statement, whose label is CMN in the above example, is similar to a DECLARE statement in that control automatically bypasses it during sequential execution of instructions. Also, it is forbidden to GO TO it).

Expressions in the Format List

When considering the items in parenthesis within the format list, such as the number of lines to be skipped or the number of characters to be printed, it is possible to have expressions or variables in these positions.

Examples:

Suppose the input stream consists of character fields of variable lengths, each field being preceded by a two byte fixed decimal field specifying the length of the character field then the stream could be read in with the following statement:

```
 GET EDIT(M,FLDA,M,FLDB) (F(2),A(M),F(2),A(M));
```

The value of M is the length of the associated character field.

Example:

```
 DCL 1 REC,
       2 FIXED,
         3 NUM PIC'9',
         3 DATA CHAR(44),
       2 VARIABLE(7) CHAR(5);
```

The above represents a structure consisting of a fixed length portion and up to seven portions of length five. The current number of the latter being given by the value of NUM. The following would print out the present length of the structure.

```
 DCL OUTREC CHAR(80) DEF REC;
 PUT FILE(OUT) EDIT(OUTREC)  (A(45+5*NUM));
```

Note that the statement:

```
 PUT FILE(OUT) EDIT(REC)  (A(45+5*NUM));
```

would not produce the required result. This would print out each element of the structure, each with length 45+5*NUM.

Repetitive Specifications in Data Lists

This facility is mainly of use when processing arrays, but it may be used for processing any items in a data list.

When an array name appears in the data list of a GET or PUT statement, all elements of the array are processed in the order in which they are stored. It may be required to process the array in a different order or else only process part of the array. This can be achieved by using repetitive specifications. Repetitive specifications have a similar format to iterative DO group specifications and are treated as an item in the data list.

....(items DO var=specification)....

where 'items' are any items allowed in a data list, 'var' is an index variable and 'specification' is any specification or specifications allowed for in an iterative DO statement.

Example:

```
 GET EDIT ((EL(I) DO I = 1 TO 3)) (A(3));
```

The repetitive specification is the only item in the list. It must have its own parentheses, and the whole list is enclosed in parentheses. The statement is equivalent to:

```
DO I = 1 TO 3;
  GET EDIT(EL(I)) (A(3));
END;
```

Example:

```
PUT EDIT(('*' DO I = 1 TO 5Ø)) (A);
```

will cause a row of 50 asterisks to be printed.

Transmission Statement Options

With EDIT directed input/output we have already discussed various control format items such as SKIP and COL which enable the positioning of input and output files. However, for DATA and LIST directed input/output there is no direct equivalent. Instead we have TRANSMISSION STATEMENT OPTIONS. These can be used with EDIT directed input/output but as we shall see later, the control format items are more powerful.

The COPY Option

Any GET statement may include the COPY option, which causes the data from the input stream to be copied without modification on the specified file (or on SYSPRINT if no file is specified).

Example:

```
GET DATA(A,B,C) COPY(COPFIL);
```

This not only causes assignments to A, B or C to be made (if there are any in the next group of input data), but also causes these assignments to be written to the file COPFIL.

The statement

```
GET DATA(A,B,C) COPY;
```

would cause the assignments to be written in data-directed format on the SYSPRINT file.

The COPY option is an extremely convenient way of obtaining a hard copy of input data.

The SKIP Option

This option specifies that a number of records or, for PRINT files, a number of lines, are to be skipped. It may optionally be used in conjunction with a Stream I/O specification and is followed by an optional expression (which may be a constant) which is converted to an integer indicating how many records, or lines, are to be skipped. If the expression is omitted, SKIP(1) is assumed.

For non-PRINT files, SKIP may be used either for input or for output. Skipping is always performed before data transmission, regardless of the order in which the options are coded.

Example:

```
 GET SKIP(3) DATA;
 GET DATA SKIP(3);
```

Both the above statements mean the same, viz; 'skip to beginning of the third record on from here and GET DATA'.

For PRINT files

```
 PUT SKIP(Ø);
```

repositions the printer at the beginning of the **current** line, thereby permitting over-printing. SKIP(0) is never needed for non-PRINT files and is not permitted.

The PAGE Option

This may only be used with a PRINT file, and always causes the file to be repositioned at the start of a new page. It may be used on its own, or in conjunction with a stream output specification, in which case it will always be performed before output takes place.

Example:

```
 PUT PAGE;
```

causes SYSPRINT to be repositioned at the top of a new page.

The LINE Option

This, too, may be used only with PRINT files, and **must** be followed by a parenthesized expression or constant which indicates the **absolute** number of the line on the page which is to be skipped to. **(The SKIP option specifies a line number relative to the current one).**

Thus:

```
 PUT LINE(31);
```

would cause the print file to be repositioned on line 31 of the current page (or of the next page if it is already past that line).

Both PAGE and LINE may appear in one statement, in which case PAGE always takes effect first, although they may be coded in any order. LINE may, in common with SKIP and PAGE, be used either on its own, or with a stream output specification.

As mentioned above, EDIT directed input/output control format items are more powerful than transmission statement options. For example the statement:

```
 PUT EDIT(X,Y,Z) (F(1),PAGE,A(2Ø),SKIP(2),F(3));
```

would have to be written as follows using transmission statement options.

```
 PUT EDIT(X) (F(1));
 PUT PAGE EDIT(Y) (A(2Ø));
 PUT SKIP(2) EDIT(Z) (F(3));
```

Instead of one statement, three would be needed.

PAGESIZE and LINESIZE

The default size of each line of a PRINT file is 120 characters, and the number of lines per page is 60. Both may be changed at OPEN time by means of the LINESIZE and PAGESIZE options.

```
 OPEN FILE(SYSPRINT) PAGESIZE(4Ø) LINESIZE(5Ø);
```

will cause pages of 40 lines each to be written, each line being only 50 characters in width.

Note carefully that PAGESIZE may only be used with STREAM PRINT files, and LINESIZE only with STREAM OUTPUT files.

Endpage Condition

This condition will be raised when a PUT statement attempts to write a line beyond the limit set for the current page. **The standard system action is to skip to a new page. For further information see Section H of the Language Reference Manual.**

Stream Handling Built-in Functions

Lineno B.I.F.

This built-in function can only be used for PRINT files. It returns the current line of the specified file. The general format is:

LINENO(x)

where x is the name of a file which must have the PRINT attribute.

Count B.I.F.

The general format is:

COUNT(x)

where x is a file having the STREAM attribute. It returns the number of items transmitted during the last GET and PUT operation.

Example:

```
       DCL IN-COUNT INITIAL(Ø);
 READ: GET FILE(INPT) DATA;
       IN-COUNT = COUNT(INPT) + IN-COUNT;
       IF IN-COUNT > 1ØØ  THEN DO;
                               .
                               .
                               END;
                          ELSE DO;
                               .
                               .
                               END;
```

Thus when one hundred data items have been transmitted, the first 'DO Group' will be executed.

Summary

The full range of PL/I input and output facilities has now been covered, and you should now be in a good position to appreciate the great flexibility which they provide. For efficiency we have Record I/O, while, for ease of programming, Stream I/O is supplied.

Moreover, PL/I does not merely permit a single method of Stream I/O (as does FORTRAN), but has no less than three: data-directed for utmost simplicity of use; list-directed, which provides a little more flexibility but requires correspondingly more programming effort, and edit-directed for the programmer who wants complete control.

Finally, a word of warning: List-directed I/O is **very** inefficient, and data-directed even more so. Data-directed I/O not only incurs execution time penalties, but also lots of storage is taken up by the dictionary of data names which it needs to maintain. They should only be used in very infrequently executed routines. The only form of Stream I/O which should even be considered when any great quantity of data transmission is required is **edit-directed.** When execution time is an important consideration, however, **all** types of Stream I/O should be reserved for such things as 'one-time jobs' and infrequently used error routines.

Exercises

1. What is the minimum file declaration for SYSIN for your operating system?
2. Write a statement that will print the values of V1 and V2 (but not the names) on the system printer.
3. What format would be required to produce the following outputs, assuming that they came from numeric variables?

   (a) 56.78

   (b) -5.67

   (c) CHARACTERS
4. Cards contain in columns 1-2 a number indicating the starting column, within the same card, of a field called ADDR which is 10 bytes long. Write the STREAM input statement to read these cards, using X and/or COLUMN format items.
5. Rewrite the following statements in an abbreviated version:

```
 PUT EDIT(A,B,C) (F(1),SKIP,F(1),SKIP,F(1));
 PUT EDIT(L,M,N,O) (A(1),X(2),A(3),X(2),A(3),X(2),A(3));
```

6. Recode the following in full, without iteration factors, and explain what it means:

```
 PUT PAGE EDIT(X,Y,Z) (A(12),2 (X(2),F(2))) LINE(1Ø);
```

   (Assume that X was declared with the attribute CHAR(12) and that Y and Z were both declared as FIXED DECIMAL(2)).
7. How will

```
 GET DATA COPY;
```

   be expanded by the compiler? What will be its effect?

8.

```
 DCL PRTOUT FILE STREAM OUTPUT ENV(.......);
```

(a) Write a statement which will open this file and specify that each page should contain 20 lines, with 50 characters in each.

(b) Write statements which will print a row of asterisks on the line **below** the bottom line of each page, and then start a new page.

9. A program contains only three variables: X, Y and Z. They all have the attributes FIXED DECIMAL(9). Write the most suitable statements, including any necessary declarations, so that in the event of the raising of the ERROR condition, the values of all three variables will be output to SYSPRINT, for debugging purposes. (Each value should appear **with its name).**

Answers

1. No declarations are needed in DOS/VS or OS/VS for SYSIN or SYSPRINT. They are the standard system files.

2.

```
PUT LIST(V1,V2);
```

3. (a) F(5,2)

   (b) F(5,2)

   (c) A(10), or A if the length of the item were 10 bytes long.

4.

```
GET EDIT(N,ADDR) (COL(1),F(2),COL(N),A(1Ø));
```

The COL(N) can be replaced by X(N-3).

5.

```
PUT EDIT(A,B,C) (F(1),SKIP);
PUT EDIT(L,M,N,O) (A(1),3 (X(2),A(3)));
```

6.

```
PUT PAGE EDIT(SYSPRINT) (X,Y,Z) (A(12),X(2),F(2)
                                 X(2),F(2)) LINE(1Ø));
```

Print the values of X, Y and Z on line ten of a new page of SYSPRINT. X is to be twelve characters long, Y and Z are two characters long and there are to be 2 blanks between X and Y and also Y and Z.

7.

```
GET DATA(SYSIN) COPY(SYSPRINT);
```

The stream input data read by the GET statement will be copied onto SYSPRINT.

8. (a)

```
OPEN FILE(PRTOUT) PAGESIZE(2Ø) LINESIZE(5Ø);
```

(b)

```
ON ENDPAGE(PRTOUT) BEGIN;
                     PUT EDIT(('*' DO I = 1 TO 5Ø)) (A);
                     PUT PAGE;
                   END;
```

9.

```
ON ERROR BEGIN;
               PUT DATA;
             END;
```

Topic 16

INDEPENDENT STUDY PROGRAM

Topic 16

Controlling the Compiler

This topic discusses some of the compiler options available to the programmer, how he can control the use of them and how he can use them to locate and correct source statement errors.

Objectives

At the end of this topic you should be able to:

- control the production of the various items of printed compiler output
- interpret the information contained in the following:

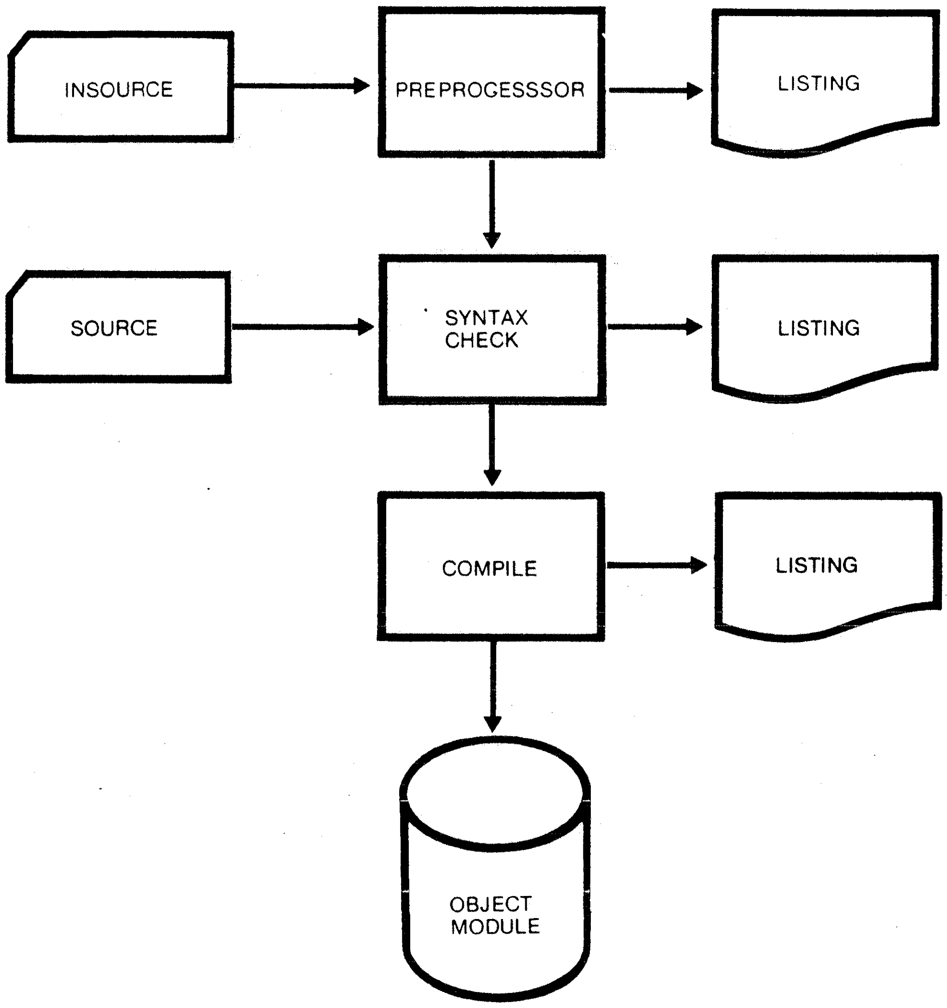
  Source Listing
  Attributes and Cross-reference listing
  Diagnostics
- locate and correct source statement errors diagnosed by the compiler
- code statements to cause the compiler to incorporate text from a library by means of a %INCLUDE statement
- using the Programmer's Guide, list all options that control compiler action, and describe the effect of each.

Introduction

The major purpose of the compiler is to translate PL/I source statements into machine instructions. A set of such instructions is called an object module. However, the compiler does not generate all the machine instructions required to represent the source statements. For various frequently used routines, it inserts references to standard subroutines in the DOS/VS or OS/VS PL/I RESIDENT LIBRARY which are then included by the Linkage Editor or OS/VS loader. Some other routines are loaded at execution time and the storage they occupy is released when they are no longer required. These routines reside in the DOS/VS or OS/VS PL/I TRANSIENT LIBRARY.

While it is processing a PL/I source program, the compiler produces a listing that contains information about the source program and the object module derived from it, together with diagnostic messages relating to errors or other conditions detected during compilation. Much of this information is optional and is supplied either by default or in response to a request, made by including appropriate options either in the compiler-control PROCESS statement that optionally precedes each source module or else as a PARM parameter on the end of the OS/VS EXEC statement. These options and their effects will be studied in this topic.

Stages of Compilation



Action taken by the compiler consists of three stages:

PREPROCESSING

SYNTAX CHECKING

COMPILING

Preprocessor

The initial stage, the preprocessor, is optional. Its function is to alter its input (known as the INSOURCE) according to appropriate commands. It treats this as a string of characters and its functions include:

adding more text from libraries
modifying the statements
omitting statements

The output from the preprocessor is known as the SOURCE which can be fed immediately into the next stage of the compiler or punched out onto cards (SOURCE DECK). Only the first of the above functions will be considered in this chapter.

Syntax Check

The second stage is known as the syntax check. Here the input (SOURCE statements) is checked to ensure that it consists of valid PL/I statements. Various listings are produced at this stage. These will be discussed later in the topic. One of the listings is 'error diagnostics'. These are message giving information concerning possible or certain errors found in the SOURCE statements. They are classified according to severity of error as follows:

I	informatory
W	warning
E	error
S	severe
U	unrecoverable

Full information on the above can be found by referring to the beginning of the OS/VS or DOS/VS PL/I Optimizing Compiler: Message Manual.

Compilation

The final stage of the compiler is to convert the SOURCE statements into machine readable code (known as the OBJECT PROGRAM). This stage can be made dependent upon the level of success of the previous stages, because there is no point in producing an object program if there are errors within the source program.

Compiler Options

A list of all the options is given in the OS/VS or DOS/VS PL/I Optimizing Compiler: Programmers Guide. Refer to the appropriate figure in Chapter 4. As can be seen, the figure is split into three columns, the first giving the name of the option and its alternatives, the second gives an abbreviated name which can be used, and the third gives the default when the compiler is first supplied.

Option Formats

The options take one of three different formats:

1. Keyword/Opposite e.g. AGGREGATE | NOAGGREGATE
2. Keyword + value e.g. MARGINS(m,n,c)
3. Combination of above e.g. MARGINI('C') | NOMARGINI

(Throughout this topic the notation ' | ' will be used to mean 'OR').

Installation Default

The option defaults supplied by IBM may not satisfy the requirements of an installation and so the defaults can be modified to produce installation defaults.

Installation Suppression

An installation can also suppress options so that even if requested they cannot be obtained. Suppressed options can be reinstated for a compilation by means of the CONTROL option.

Programmer Control of Options

Before looking at the options in detail, the methods by which the programmer can control them will be studied. At this stage various options will be mentioned by name although their meaning may not be known. The DOS/VS programmer has only one way of controlling the options and that is by the use of the PROCESS statement. The OS/VS programmer will normally use PARM parameters although in certain circumstances he (or she) may resort to the PROCESS statement (see later in the topic).

PROCESS statement

The general format is:

*** PROCESS options list;**

The rules governing the format are as follows:

1. The PROCESS card - if used - must be the first card the compiler encounters within each compilation.
2. There can be several of them if required (see example 1).
3. The asterisk must be in column 1.
4. The keyword PROCESS can begin in column 2, or there can be blank(s) before it.
5. The options list consists of the options we wish to change for **this** compilation.
6. The options are separated by commas or blanks (see examples 2 and 3).
7. The entire statement must end with a semi-colon.
8. There may be continuation cards (the scan is columns 2 through 72) (see example 3).

Example 1

```
// EXEC PL1OPT
* PROCESS DECK;
* PROCESS MACRO;
```

Example 2

```
* PROCESS AG,MACRO,NOINSOURCE NODECK;
```

Example 3

```
* PROCESS
          AGGREGATE,
          MACRO
          NOINSOURCE;
```

PARM parameters (OS/VS only)

(See examples below)

The parameters are enclosed in quotes and added onto the EXEC statement. Note that in a procedure the PARM field is qualified by the name of the job step in which the compiler is executed i.e. in example 2 we wish the compiler options to apply in the PL/I step of the procedure. Note also that in example 2, the MARGINI option includes quotes and that these need to be doubled up when used as a PARM parameter.

Example 1

```
//STEP EXEC PGM=IELOAA,PARM='OBJECT,NOLIST'
```

Example 2

```
//STEP EXEC CATPROC,PARM.PLI='ESD,MARGINI(''#'')'
```

Scope of Compilation

The normal scope of compilation i.e. (how much will be compiled in one go) is from the first PROCEDURE statement to its respective END statement. Any interval procedures will be compiled. Thus in the following DOS/VS example, procedures A and B will be compiled but not procedure C:

Example

```
//  EXEC PLIOPT
   A: PROC;

         B: PROC;

         END B;
   END A;

   C: PROC;

   END C;
```

In this case the compiler would produce a diagnostic message stating that the logical end of the program (END A) has been found before the end of the source deck.

Multiple Compilations

A program will normally consist of more than one external procedure and hence there is a requirement for the scope of compilation to be increased. One way would be to reinvoke the compiler for each external procedure as follows:

Reinvocation

Example (DOS/VS)

```
// EXEC  PLIOPT
   A:  PROC;

   END A;
/*
// EXEC  PLIOPT
   B:  PROC;

   END B;
/*
/&
```

Example (OS/VS)

```
//STEP1 EXEC PLIXCG
//PLI.SYSIN DD *
   A:PROC;
     .
     .
   END A;
/*
//STEP2 EXEC PLIXCG
//PLI.SYSIN DD *
   B:PROC;
     .
     .
   END B;
/*
```

One disadvantage of this method is that the compiler has to be loaded each time it is required. This is an I/O operation and thus takes time. A quicker method would be to load the compiler once and then alter the options as necessary for each procedure. This can be achieved by inserting PROCESS statements between each individual compilation as follows:

Batched

Example (DOS/VS)

```
// EXEC PLIOPT
* PROCESS DECK,AG;
   A: PROC;
      .
   END A;
* PROCESS NODECK,AG;
   B: PROC;
      .
   END B;
* PROCESS;
   C: PROC;
      .
   END C;
/*
/&
```

Example (OS/VS)

```
//STEP EXEC PLIXCG,PARM.PLI='DECK,AG'
//PLI.SYSIN DD *
   A: PROC;
      .
   END A;
* PROCESS NODECK;
   B: PROC;
      .
   END B;
* PROCESS NODECK,NOAG;
   C: PROC;

   END C;
/*
```

The resultant compilations by the batched method are still as separate as they were under the reinvocation method. It is in the batched method that the OS/VS programmer needs to resort to PROCESS statements. These are necessary to 'delimit' compilations, even if no options are coded on it. After each individual compilation, the options return in DOS/VS to the default options, and in OS/VS to the options set in the EXEC statement. Thus in the examples above the settings of the DECK and AG options are as follows (assuming that NODECK and NOAG are the defaults):

PROCEDURE A:	DECK,AG
PROCEDURE B:	NODECK,AG
PROCEDURE C:	NODECK,NOAG

Compiler Options

Chapter 4 of the OS/VS or DOS/VS PL/I Optimizing Compiler: Programmer's Guide gives a complete meaning of each of the compiler options. Hence only some of the options will be explained in full, giving details of how they would be used. Any others should be looked up as required.

OPTIMIZE(TIME | 2 | 0) | NOOPTIMIZE

OPTIMIZE (TIME | 2) have the same meaning and specify that full optimization of the program is to occur. Thus the resulting program will be very efficient with a fast execution time. This will be at the expense of compilation time which will be substantially increased.

OPTIMIZE(0) | NOOPTIMIZE have the same meaning and specify that a fast compilation time is required at the expense of a less efficient object program.

The latter should be used when testing a program. At this stage the emphasis is on removing errors and correcting the logic of programs. When the programmer is satisfied with his program, it should be compiled once more under OPTIMIZE (TIME | 2) to produce a very efficient production program.

A full discussion on optimization is included in the Language Reference Manual. It includes details of the various ways in which the compiler optimizes programs.

Preprocessor Options

MACRO | NOMACRO

MACRO specifies that the use of the preprocessor is required. NOMACRO specifies that the first stage of compilation will be syntax checking. The full use of the preprocessor will not be covered on this course.

INCLUDE | NOINCLUDE

One of the most used of preprocessor facilities is the inclusion of source text from the library. Typical inclusions would consist of:

a) standard code which will be used repeatedly

b) file declarations

c) complex structures

The aim being to reduce programmer effort (i.e. avoid duplication of coding) and the chance of errors in the coding. INCLUDE specifies that **this and only this** facility of the preprocessor is required. The inclusion can then occur without the preprocessor stage being executed. The inclusion will take place during syntax checking when the appropriate INCLUDE statements are encountered. If MACRO and INCLUDE options are both specified, then the latter has no effect.

Source Text Inclusion

To include source text either MACRO or INCLUDE must be in effect.

Example

```
A: PROC;
   % INCLUDE BKNAME;
   % INCLUDE R(BOOK);
   % INCLUDE BOOK1,R(BOOK2),BOOK3;
END;
```

The whole statement is a normal PL/I statement and thus must lie between columns 2 and 71. Blanks between '%' and 'INCLUDE' are optional. In OS/VS BKNAME would be a member within a partitioned data set defined by the SYSLIB card.

//SYSLIB DD DSN=....

and BOOK would be a member within the data set defined by:

//R DD DSN=

In DOS/VS BKNAME will be in the default PL/I source statement library (P.) while BOOK will be in the R. source statement library.

Included code can be complete procedures, statements or parts of statements. The only non-permissible statement is a PROCESS statement.

Nested Includes

Within an included text it is possible to have %INCLUDE statements. This will lead to nested includes where one set of included text is within another.

Example

```
* PROCESS INCLUDE;
  A:PROC;
    .
    .
    % INCLUDE INTPROC;
    .
    .
  END A;
```

Suppose that INTPROC and DCLS are the names under which the following two sets of statements are cataloged into the P. library (DOS/VS) or the SYSLIB library (OS/VS). INTPROC consists of:

```
  INTPROC:PROC;
          .
          .
          % INCLUDE DCLS;
          .
          .
  END INTPROC;
```

DCLS consists of:

```
  DCL 1 STRIN,
        2 TYPE     CHAR(1),
        2 NAME     CHAR(2Ø),
        2 ADDRESS  CHAR(4Ø),
        2 ACCOUNT  CHAR(5);
```

Then the expanded form of the procedure A will be as follows:

```
  A:PROC;
    .
    .
    INTPROC:PROC;
            .
            .
            DCL 1 STRIN,
                  2 TYPE    CHAR(1),
                  2 NAME    CHAR(2Ø),
                  2 ADDRESS CHAR(4Ø),
                  2 ACCOUNT CHAR(5);
            .
            .
    END INTPROC;
    .
    .
  END A;
```

Debugging Options

These options help the programmer to locate errors within his program and to correct them.

COUNT | NOCOUNT

COUNT specifies that at the end of execution a table will be printed showing how many times each statement/group of statements has been executed.

Below is a program and the associated count table produced when the program is executed with the stated data.

```
FLOW: PROC OPTIONS(MAIN);                                                          1
      DCL (VALUE(1ØØ),PRICE(1ØØ)) CHAR(5));                                        2
      DCL NUMBIN FILE RECORD ENV(MEDIUM(SYSIPT,254Ø) F RECSIZE(8Ø));               3
                                         /* MEDIUM OPTION SINCE PROGRAM
                                            WAS RUN UNDER DOS/VS           */
      DCL EOFSW BIT(1) INIT('Ø'B);                                                 4
      DCL 1 STR,                                                                   5
            2 NO PIC'99',
            2 COST(11) PIC'9999999',
            2 BLANK CHAR(1);
      ON ENDFILE(NUMBIN) SNAP EOFSW = '1'B;                                        6
      J = Ø;                                                                       7
      READ FILE(NUMBIN) INTO(STR);                                                 8
      DO WHILE(¬EOFSW);                                                            9
         ITOTAL = Ø;                                                              1Ø
         J = J + 1;                                                               11
         DO I = 1 TO NO;                                                          12
            ITOTAL = ITOTAL + COST(I);                                            13
         END;                                                                     14
         IF ITOTAL > 1ØØ THEN DO;                                                 15
                                    VALUE(J) = 'LARGE';                           16
                                    PRICE(J) = 'DEAR ';                           17
                              END;                                                18
                         ELSE DO;                                                 19
                                    VALUE(J) = 'SMALL';                           2Ø
                                    PRICE(J) = 'CHEAP';                           21
                              END;                                                22
         READ FILE(NUMBIN) INTO(STR);                                             23
      END;                                                                        24
      END;                                                                        25
```

Data:

```
0500001230000000500000009000000070000025
0200000110000104
```

Count Output:

```
STATEMENT COUNT TABLES
EXECUTED STATEMENTS
PROCEDURE FLOW
                FROM        TO    COUNT
                   1         5        1
                   6                  2
                   7         8        1
                   9                  3
                  10        11        2
                  12        14        7
                  15        18        2
                  23                  2
                  24                  3
                  25                  1
                FLOW     TOTAL       51
    GRAND TOTAL                      51
STATEMENT COUNT TABLES
UNEXECUTED STATEMENTS
PROCEDURE FLOW
                FROM        TO
                  19        22
END OF COUNT TABLES
```

Before a program can be said to be thoroughly tested, it is necessary that each statement within the program has at some time been executed. The COUNT table will list out those statements which have not been executed. If it is found that certain statements cannot be executed, whatever input test data is used, the question to be asked is 'why are they in the program?'

Another use of the COUNT option is as an aid for increasing program efficiency. If efforts are being made to speed up program execution, then it will be far more useful to concentrate one's effort on statements which are frequently executed rather than on ones which are only executed once or twice.

GOSTMT | NOGOSTMT

If a program 'blows up' at execution time then normally a message is produced stating the cause of the error. The GOSTMT option specifies that within the message, statement numbers will be used to identify the location of the error. Later in the topic a more efficient method of locating errors will be introduced. The message will be discussed in full in Topic 20, 'TESTING AND DEBUGGING AIDS'.

FLOW((n,m)) | NOFLOW

FLOW requests that a table be maintained, at execution time, of program branches. 'n' specifies the number of entries to be maintained while 'm' specifies the number of procedures to be maintained. The defaults for 'n' and 'm' are 25 and 10 respectively.

The table can be printed out on request by means of the SNAP option (see Topic 19, 'HANDLING EXCEPTIONAL CONDITIONS'). It is usually required on program failure and is an aid for the programmer to trace the flow of the program. An example of the flow output is given below. This is the flow produced by the program shown under the COUNT option. It will be discussed at a later stage (Topic 20, 'TESTING AND DEBUGGING AIDS').

```
14    TO 12
14    TO 12
14    TO 12
14    TO 12
18    TO 23
24    TO  9
14    TO 12
18    TO 23
```

Listing Options

The options below refer to various listings produced by the compiler. While reading the explanation you should refer to the sample listings in the appendix named 'PROGRAMMING EXAMPLE' in the Programmers Guide.

OPTIONS | NOOPTIONS

The first listing is produced by the OPTIONS option. It gives two sets of information, first of all a list of options that have been specifically requested, and secondly details of all the options for this execution of the program. The options are listed in three columns, the first giving options which are 'on', the second those which are 'off' and the third those which have a value associated with them.

INSOURCE | NOINSOURCE

The second listing is the input to the preprocessor and produced by the INSOURCE option. On this segment we will not be considering it in any detail. Note, in the OS/VS example, that a number of statements commence with a '%'. These are statements which give instructions to the preprocessor. In the DOS/VS example there is a % INCLUDE statement. Note that it is replaced by the included text in the SOURCE statement listing.

SOURCE | NOSOURCE

After the preprocessor diagnostic message there appears the source statement listing, produced by the SOURCE option. This is the input to the syntax checking and compilation. If the STMT option is in force, each statement is numbered so it can be referenced by diagnostic messages. The numbers on the righthand side are references to INSOURCE linenumbers.

NEST | NONEST

This option produces the columns headed LEV and NT on the source statement listing. LEV is an indication of depth of block level and NT is an indication of DO group level. The columns can be useful in checking how the compiler has matched up END statements with DO, PROCEDURE and BEGIN statements.

ATTRIBUTES[(FULL | SHORT)] | NOATTRIBUTE

This option produces the attributes table: a list of all identifiers used within the program. If an identifier has been explicitly declared, the statement in which this occurred is stated. Otherwise, a row of asterisks is printed. The attributes of the identifier are printed in the right-hand column. If the ATTRIBUTE option SHORT is in force then any unreferenced identifiers are omitted.

XREF[(FULL | SHORT)] | NOXREF

This option produces a table of all identifiers and the source statements in which they have been used. If the ATTRIBUTE option is in force then both printouts appear on the same listing. The SHORT option of the XREF attribute has the same effect as the the ATTRIBUTES attribute.

These two listings are useful when queries regarding identifiers arise.

Three examples of their usage are as follows:

1. Checking the attributes of an identifier.
2. If the name of an identifier is to be altered, then all references to that identifier will also have to be altered. The XREF listing will indicate where these references are.
3. If a punching error has created a wrongly spelled identifier, this will appear as an implicitly declared identifier and can easily be detected.

AGGREGATE | NOAGGREGATE

The AGGREGATE LENGTH TABLE is the next listing. Here is displayed information regarding structures and arrays used within the program. The size of each element and also the number of elements in the array is presented. For structures, the size of each element, the offset of the elements from the beginning of the structure and also the total size of the structure is given.

The new few listings produced by the STORAGE, ESD and MAP options will not be discussed on this segment.

OFFSET | NOFFSET

This option produces the 'TABLES OF OFFSETS AND STATEMENT NUMBERS'. These tables can be used to find the statement in which a program blew up, rather than use the GOSTMT option. The tables consist of the OFFSET (the displacement) of each statement from the beginning of the procedure in which it appears. The execution time error message will contain the offset as follows:

.......AT OFFSET xx in PROCEDURE WITH ENTRY POINT name.

The statement, where the error occurred, can be found by searching the offset table of procedure 'name' for the largest offset printed which is smaller than 'xx'. This will be the offset of the desired statement. More detail on this subject will be found within Topic 20, 'TESTING AND DEBUGGING AIDS'.

LIST | NOLIST

The listing called OBJECT LISTING is produced by the LIST option. This is a breakdown of the PL/I statements into pseudo-assembler coding. This will be of use to programmers with knowledge of assembler language, for in-depth debugging.

FLAG(I | W | E | S)

This option informs the compiler as to what level of error diagnostic messages are required to be printed e.g. FLAG(E) means print only messages of severity level E and above. As can be seen the format of the messages is

IELnnnnI severity level [Statement] Text

Full information about the messages can be found by looking up the appropriate message number in the DOS/VS or OS/VS PL/I Optimizer Compiler: Messages Manual. The statement number refers to the statement in error. In some cases it may be omitted if the compiler cannot pin-point the statement e.g. if an END statement is missing the compiler will now know where-abouts the omission occurred and thus would be unable to give a statement number.

Summary

This topic has acquainted you with options which may be specified for the compiler, how to specify them, and what their effects are. You should be in a position to decide what options you need for any given purpose, so that these can be specified precisely, others being rejected. Check first however for any installation standards which are in effect. **It is important that you do not specify options which are not required as these can be wasteful in time and paper.**

Compilation errors can normally be eliminated easily be reference to the diagnostic messages. It should be noted that frequently a single error in the program can produce several messages. Even if the compiler has made the correct assumptions in the diagnostic messages, you are advised to correct the error yourself otherwise compiler optimization is inhibited in future compilations.

Exercises

1. Assume that the IBM default compiler options apply. Use your manual to answer the following questions:

   (a) What is the minimum severity of compiler messages which will be printed?

   (b) Which other listings will be produced?

   (c) How many lines per page will be printed for these listings?

   (d) Between which columns can PL/I statements be coded?

2. What is indicated by a string of asterisks next to an identifier name is the Attribute and Cross-reference Table?

3. What is the main advantage of 'batched compilation'?

4. When the compiler was installed, the following changes were made to the defaults:

   MARGINS(2,72) became MARGINS(10,80)

   CHARSET(60,EBCDIC) became CHARSET(48,EBCDIC)

   NODECK became DECK

   For a particular compilation, you require '+' printed down each side of the source listing.

   Your source deck was coded between the usual columns of 2 through 72, and you have written in the 60 character set. You wish also to produce an object deck.

   OS/VS **Complete the EXEC card**

```
//STEP EXEC PLIXCG . . . .
```

   DOS/VS **Code the PROCESS card(s) required**

Answers

1. (a) ALL messages will be printed, because FLAG(I) is the default.

   (b) Source statement listing (SOURCE). Note the INSOURCE listing will not appear because the MACRO option is not in effect.

   (c) 55 (LINECOUNT (55))

   (d) Columns 2 and 72 (MARGINS (2,72,0))

2. This indicates that the identifier has not been explicitly declared by the programmer.

3. Batched compilations save the overheads of repeated initialization of the compiler.

4. **OS/VS**

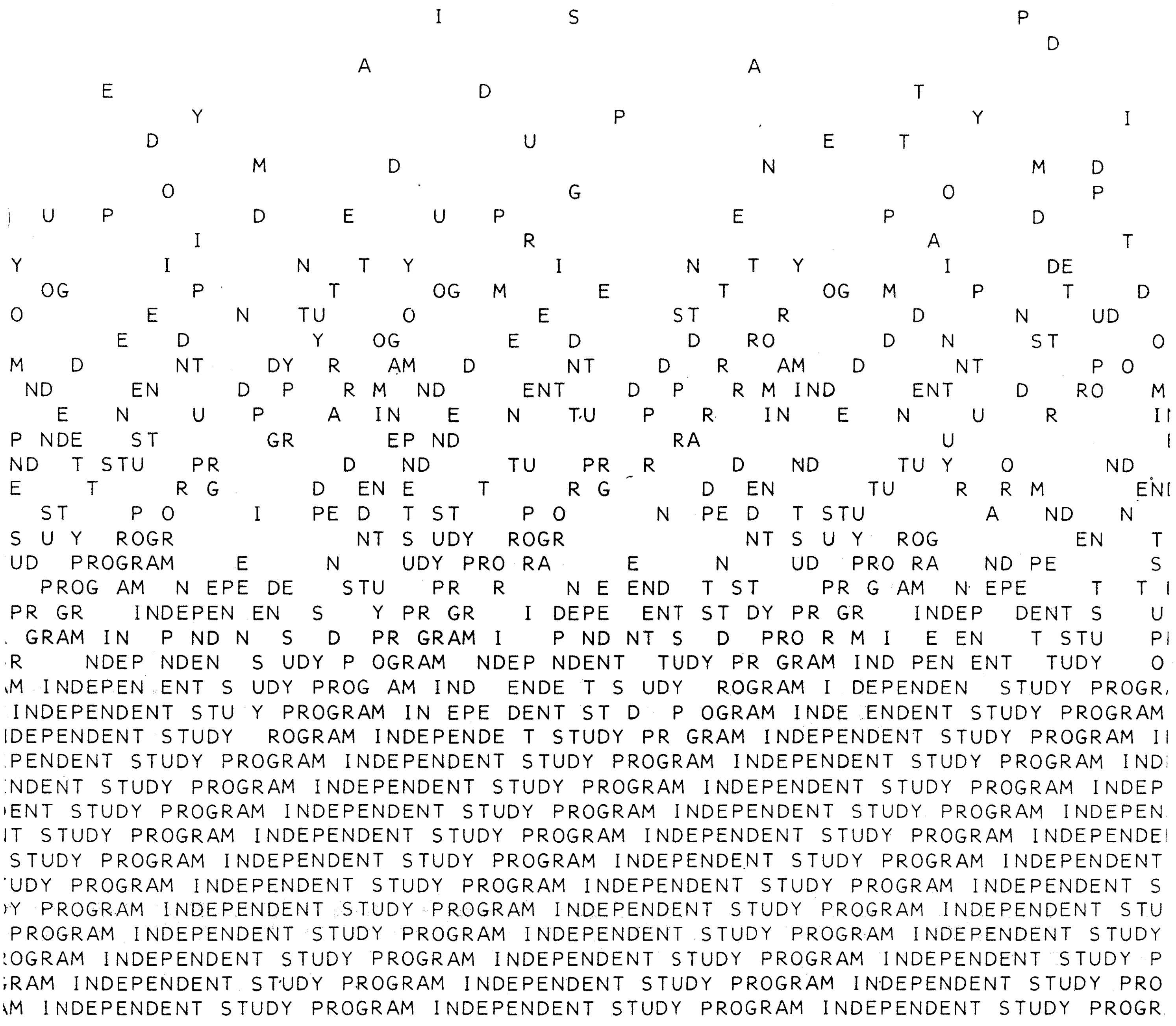
```
//STEP EXEC PLIXCG,PARM.PLI='MARGINS(2,72),MARGINI(''+''),CHARSET(6Ø)'
```

DOS/VS

```
* PROCESS MARGINS(2,72),MARGINI('+'),CHARSET(6Ø);
```

Note that within CHARSET, only the required change needs to be specified; EBCDIC can be omitted.

Topic 17



Topic 17

The PL/I Block Structure

In this topic you will learn about the two types of 'blocks' in PL/I: BEGIN and PROCEDURE. You will learn why and when they are used and what effect the block structure has on the scope of names and on variables with automatic storage class.

Objectives

At the end of this topic you should be able to:

- explain why there are 'blocks' in PL/I
- explain the difference between BEGIN blocks and PROCEDURE blocks
- describe the relationship between the block structure and the scope of names
- describe the relationship between the block structure and the allocation of storage for variables with AUTOMATIC storage class
- explain the effect of multiple closure of blocks and DO-groups.

Introduction

It is generally accepted that any reasonably sized program is best divided into separate parts or modules. This 'modular' approach has advantages not only during coding but also during subsequent testing and debugging. Because of this, it is important that a programming language should lend itself to the modular approach. This modular approach will be discussed in detail in Topic 18.

The essential feature of PL/I which provides modularity is its flexible **block structure.** A block constitutes one module or section of a PL/I program; it consists of a delimited sequence of statements. In this topic we will discuss the nature and use of PL/I blocks.

Note that a block is not the same as a group or DO-group. A **group** in PL/I is a set of statements beginning with a DO statement and ending with an END statement. Groups are used to group together a collection of statements so that they may all be used as the THEN or ELSE clause of an IF statement, or may be executed more than once (iterative DO-group), and that is all.

A block is a collection of statements beginning with either a PROCEDURE statement or a BEGIN statement and ending with an END statement. Blocks may be used to isolate identifiers so that an identifier in one block refers to a different area of storage to an identifier **with the same name** in another block (see later in this topic). Blocks may be used to reduce the amount of storage required when executing the program (see later in this topic). Most commonly, blocks are used to speed program development and to make sections of code usable in other programs (see Topic 18).

Let's look at procedure blocks first. These can be either **internal** or **external.** We will explain what they are, how they are used and the flow or control between procedure blocks. Towards the end of this topic we will talk about BEGIN blocks - these are less important.

EXTERNAL PROCEDURE BLOCKS

Procedures may be complete in themselves and entirely separate from other procedures. Consider the following example:

```
    PMAIN:PROCEDURE OPTIONS (MAIN);
          .
          .
    END PMAIN;




       P1:PROCEDURE;
          .
          .
       END P1;
```

The two procedures shown here, PMAIN and P1, are separate from one another; they are **external** procedures. Procedure blocks (both internal and external) must be labelled - the label is called the **entry name** of the procedure.

The OPTIONS attribute can only be specified for an external procedure; OPTIONS (MAIN) must be specified for the main or master procedure. At execution time, this procedure is given control by the operating system. Note that it is possible to specify OPTIONS (MAIN) for more than one external procedure: the operating system will then invoke the one that it first encounters.

Entry names of external procedures cannot exceed seven alphameric characters (this restriction is imposed by the operating system). If an entry name (or any external name) is written with more than seven characters, the compiler will shorten it by concatenating the first four with the last three characters.

INTERNAL PROCEDURE BLOCKS

A procedure may be nested within another procedure. Consider the following example:

```
     PEXT:PROCEDURE;
          .
          .
          PINT:PROCEDURE;
               .
               .
          END PINT;
          .
          .
     END PEXT;
```

In this example, the procedure PINT is nested within - or contained in - the external procedure PEXT. PINT is therefore an internal procedure. Note that all the statements which constitute PINT (including the PROCEDURE and END statements) occur between the PROCEDURE and END statements of PEXT (the **containing** procedure).

Blocks may be nested to depths greater than one; thus an internal procedure may itself contain another internal procedure. In addition, a procedure may contain two or more internal procedures at the same level of nesting. The use of one or more internal procedures at one (and only one) level of nesting is not uncommon; complicated nesting of internal procedures, however, is not generally used. We shall confine our attention in this topic to the typical case of an external procedure containing one internal procedure.

The entry name of an internal procedure may contain up to 31 characters, like most PL/I identifiers. Variables declared in an external procedure may be referred to within a contained internal procedure. This is one of the reasons for the provision of internal procedures and is concerned with the scope of names (see later).

Flow of Control of Procedure Blocks

We have already mentioned that a PL/I program becomes active when the initial procedure (the one with the OPTIONS (MAIN) specification) is invoked by the operating system. Thereafter the flow of control depends on the logic of the program. Normally control is passed from each statement to its successor, in sequence, unless a branch or a loop causes some modification.

External procedures (other than the initial one) and internal procedures can be **activated** (i.e. given control) by means of a CALL statement. This is called a **procedure reference** and the procedure would be a **subroutine. (We will discuss subroutines in detail in Topic 18).**

```
  CONTROL:  PROC OPTIONS (MAIN);
            .
            CALL SUB1;
            .
            CALL SUB2;
            .
            statement X;
            SUB2: PROC;
                 .
                 .
            END SUB2;
            statement Y;
            .
  END CONTROL;



     SUB1: PROC;
          .
          .
     END SUB1;
```

The statement CALL SUB1; transfers control to the external procedure SUB1. When SUB1 has completed execution it returns control to the statement in CONTROL immediately following the CALL statement which invoked it.

Similarly, the statement CALL SUB2; invokes the internal procedure SUB2. Control is returned to CONTROL at the statement following the point of invocation.

Note that an internal procedure cannot be executed as part of normal sequential flow; it must be specifically **invoked.** Thus, in the example, statement Y would be executed immediately after statement X as part of the normal sequential flow within the external procedure CONTROL.

Procedures may also be invoked by means of a **function reference** and in this case the procedure would be a function. (We will discuss subroutines and functions in the next topic).

Any procedure (whether external or internal) can invoke an external procedure which contains an internal procedure. There are restrictions, however, on directly invoking an internal procedure which is contained in some other procedure, and on the conditions under which one internal procedure can refer to another one. Certain complications can arise when procedures are nested to depths greater than one (the interested topic-reader should refer to the Language Reference Manual - but not now please!).

It is important to note that an invoking procedure **remains active** during execution of the procedure which it invokes. The MAIN procedure remains active for the duration of the entire program.

Blocks are only said to be **terminated** (i.e. become inactive) under certain conditions. The following section discusses these conditions.

Termination of Procedure Blocks

A procedure block is normally terminated when it returns control to the invoking procedure. This happens, for example, if control reaches a RETURN statement in the invoked procedure:

```
          RETURN;
```

This statement returns control to the invoking procedure at the point immediately after the point of invocation.

Control is also returned to the invoking procedure when the END statement of the invoked procedure is reached. The effect is equivalent to that of the RETURN statement. (Note that the RETURN statement can be used to return control from different points in the invoked procedure).

A GOTO statement which transfers control from an internal procedure to the external procedure which invoked it will terminate the internal procedure. This use of the GOTO statement is not recommended. (In general, a GOTO statement may pass control to any **active** block).

There are other, but much less common, causes of block termination. Details of these are in the Language Reference Manual.

The Scope of Names

The scope of a name is the area of a program throughout which it can be **directly** referred to. The following questions illustrate why the scope of names is important.

If you use a variable J declared in one external procedure, will it be referring to the same area of storage as another variable called J declared in a separate external procedure? Would the answer be any different if one procedure were internal to the other?

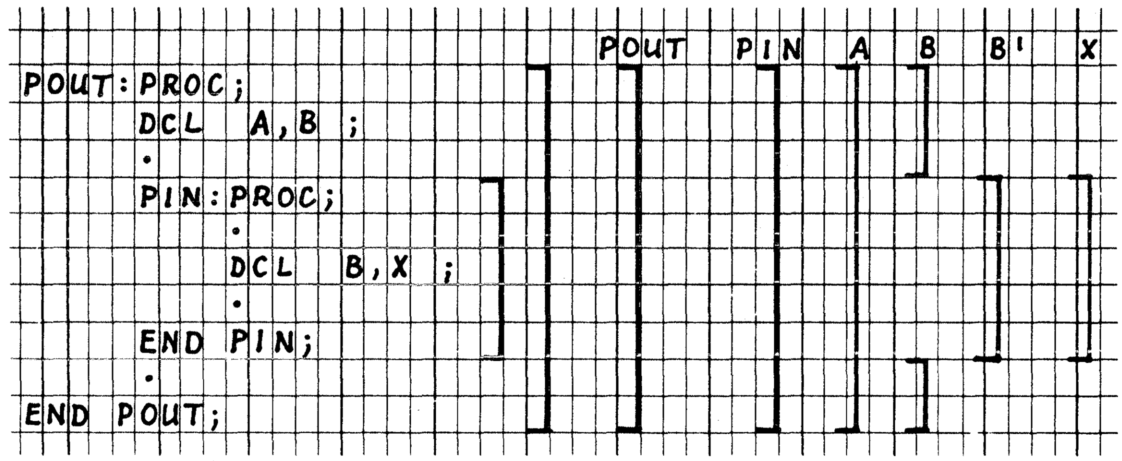
If you are writing an internal procedure can you refer to variables declared in the external procedure?

The answers to these questions depend on the scope of the name (or identifier). The scope of a name depends on how the variable is declared: explicitly, contextually or implicitly.

The Scope of an Explicit Declaration

The scope of an explicitly declared identifier is the block in which the declaration is made and any contained blocks **which do not contain an explicit declaration of the same identifier.**

For example:



```
POUT:PROC;
     DCL  A,B  ;
     .
     PIN:PROC;
         .
         DCL  B,X ;
         .
     END PIN;
     .
END POUT;
```

The lines to the right indicate the scope of the names. Points to note are:

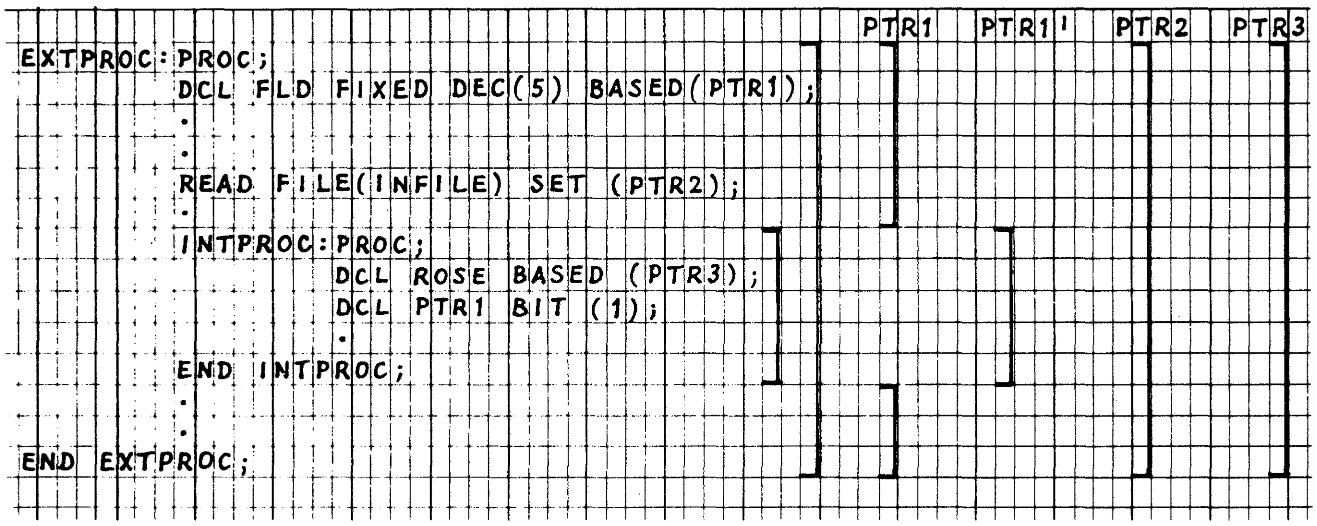
1. A may be referred to in both POUT and PIN.
2. The B declared in POUT and the B declared in PIN have the same default attributes but they refer to two different area of storage. These are represented as B and B' in the headings to the right, to illustrate the corresponding scopes.
3. X may only be addressed in PIN; it is now known in POUT.

The Scope of a Contextual Declaration

The scope of a contextual declaration is the external procedure in which the name appears (or which contains an internal procedure in which the name appears) and all contained blocks in which the same name is not explicitly declared.

Note that the scope extends to the external procedure **even when the contextual declaration is made in a block which is internal to this procedure.**

Example:



```
EXTPROC: PROC;
         DCL FLD FIXED DEC(5) BASED(PTR1);
         .
         .
         READ FILE(INFILE) SET (PTR2);
         .
         INTPROC: PROC;
                 DCL ROSE BASED (PTR3);
                 DCL PTR1 BIT (1);
                 .
         END INTPROC;
         .
         .
END EXTPROC;
```

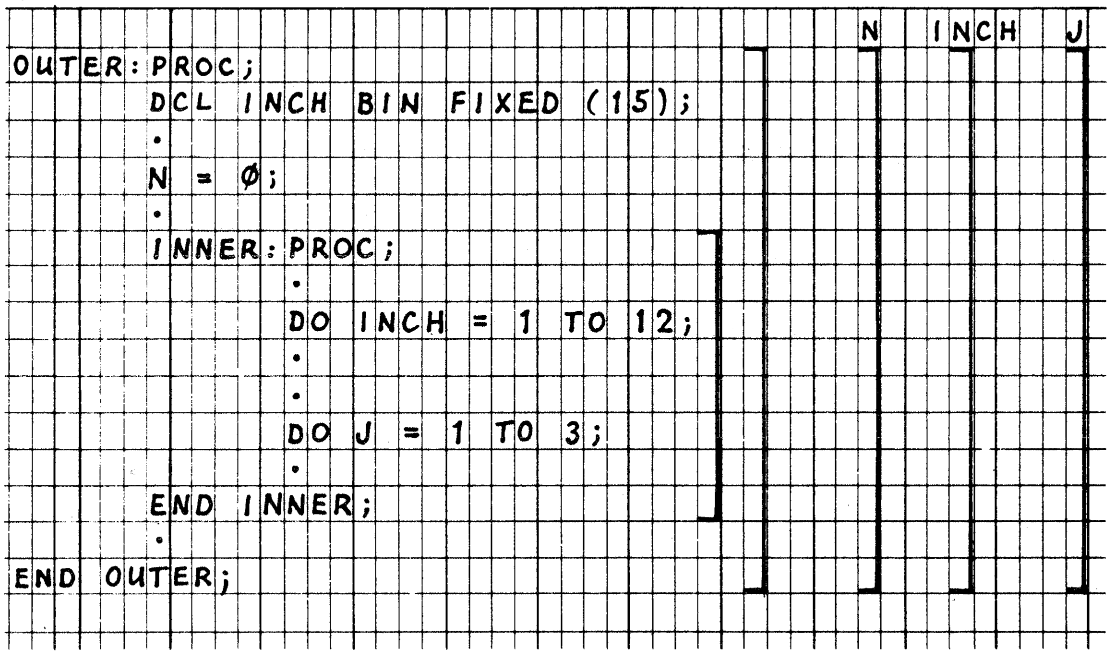
In this example, three identifiers are contextually declared as POINTER variables; these are PTR1 and PTR2 in the external procedure EXTPROC, and PTR3 in the internal procedure INTPROC. The scope of the various names is indicated by the lines to the right; note that PTR1 and PTR1' denote two separate uses of the name PTR1.

The diagram is self-explanatory. The important point to notice is that PTR3, which is contextually declared in INTPROC, may be referred to in the **external procedure EXTPROC.** The compiler treats a contextual declaration as if the declaration were made in the external procedure. This is significantly different from the scope rules for explicitly declared identifiers in internal procedures.

The Scope of an Implicit Declaration

The rules are the same as those which govern the scope of a contextual declaration. An identifier which is declared implicitly in an internal procedure is considered to be declared in the external procedure. The following example illustrates the scope of two implicitly declared identifiers, J and N (J is implicitly declared in the internal procedure; N is implicitly declared in the containing external procedure):

Example:



```
OUTER: PROC;
       DCL INCH BIN FIXED (15);
       .
       N = Ø;
       .
       INNER: PROC;
              .
              DO INCH = 1 TO 12;
              .
              .
              DO J = 1 TO 3;
              .
       END INNER;
       .
END OUTER;
```

The Scope Attributes

At this point we can introduce the scope attributes - INTERNAL and EXTERNAL (you may already have noticed these on program listings).

The scope of an INTERNAL identifier is simply the scope of its declaration (explicit, contextual or implicit).

The scope of an EXTERNAL identifier is the sum of the scopes of its declarations. In other words, an EXTERNAL variable is known in all blocks in which it is declared EXTERNAL.

As you would expect, INTERNAL is the default scope attribute for most data types. Exceptions to this rule include file names and the entry names of external procedures, which are EXTERNAL by default.

Use of the EXTERNAL Attribute

The EXTERNAL attribute is used to provide a simple way of sharing the same data items between two or more procedures. Consider the following example. GETIT, CHEKIT, and PUTIT are three external procedures: GETIT assembles data in a field called SHARE, CHEKIT examines it for validity, and PUTIT prints it. The data in SHARE could be made addressable in all three procedures by coding:

```
    GETIT:PROC;
          DCL SHARE CHAR(99) EXT;
          DCL MY_VAR BIN FIXED (15);
          .
          .
          CALL CHEKIT;
          .
    END GETIT;
```

```
   CHEKIT:PROC;
          DCL SHARE CHAR(99)EXT;
          DCL MY_VAR BIN FIXED (15);
          .
          CALL PUTIT;
          .
   END CHEKIT;
```

```
    PUTIT:PROC;
          DCL SHARE CHAR(99) EXT;
          DCL MY_VAR BIN FIXED(15);
          .
          .
    END PUTIT;
```

SHARE, which is required to be **known** in all three procedures, is declared with the EXTERNAL attribute in each of them. A reference to SHARE in any of the three procedures is then a reference to the **same area of main storage.**

In our example SHARE is declared with the same set of attributes in each of the three procedures. This is essential (since all the declarations refer to the same area) but **impossible for the compiler to check.** Remember that external procedures are separately compiled: GETIT, for example, could be coded and compiled on a Monday while CHEKIT may be written by some other programmer and compiled the following Friday. The compiler deals with EXTERNAL variables by generating an external reference which is subsequently resolved by the Linkage Editor. Care should therefore be taken to ensure that different declarations of the same name with the EXTERNAL attribute **do** have matching attributes. Failure to do this will create a misunderstanding between programmer and compiler that may have spectacular (but unprofitable) repercussions during program execution.

We have seen that the EXTERNAL attribute provides a method for sharing data between procedures which are external to each other.

If you refer again to the previous examples, you will notice that each procedure contains a declaration of a data item called MY__VAR. Each declaration is INTERNAL by default, and therefore each one refers to a different variable which can only be addressed from within the block in which it is declared and any contained blocks. When a procedure alters its own MY__VAR, the other two MY__VARs remain unchanged. Thus each MY__VAR might, for example, contain a count of how many times the associated procedure is invoked (CALLed).

A Comparison of the Uses of External and Internal Procedures

Internal and External procedures have similar characteristics and are used in much the same way.

The distinguishing point about an internal procedure is that it is nested within another procedure. The reason why PL/I allows one procedure to be nested within another is that the two procedures can then conveniently share the use of certain variables. Variables in the containing procedure are **known** within the contained procedure. In addition, variables which are declared explicitly in the internal procedure cannot be referred to in the outer procedure; an identifier may be explicitly declared in an internal procedure with the same name as one used in the containing procedure without causing confusion.

One disadvantage of an internal procedure is that a programmer might use an implicitly declared identifier (e.g. I) in an internal procedure believing it to be inaccessible from outside this procedure - **but it is accessible** (see Exercise 12 later).

External procedures have much to recommend them. They can be coded, compiled and tested separately, even by different programmers working apart from each other. Program maintenance is facilitated because procedures can be updated individually. Internal procedures must be amended with the containing procedure(s) and it is usually impracticable (or at best unwieldy) to have one programmer coding an external procedure and another writing the contained procedure. In addition, an external procedure used in one program can be link-edited into different programs - thus saving coding.

At the same time, internal procedures **are** easier to use than external procedures although they can cause problems - see Exercise 12 (later). Of course, where a procedure is used by more than one program or more than one external procedure in the same program, it should be written as an external procedure.

Storage Classes

It is often the case that main storage is not required for certain variables **throughout** a program, but only for part of the program. If different variables require storage at different times during execution, storage economy can be improved by allocating an area of storage to a particular variable when that variable is needed, and releasing it again when the variable is no longer required. This is known as 'dynamic' storage allocation. AUTOMATIC storage class is 'dynamic' in this manner. AUTOMATIC is the default for INTERNAL variables.

Variables which are required throughout the duration of a program must be allocated storage at the start of the program's execution or at compile time. This storage must not be released at any time, or the current values of the variable will be lost. One way to preserve storage throughout a program is to allocate it at compile time; this is called STATIC storage. STATIC is the default of EXTERNAL variables. (In fact EXTERNAL variables **must** be STATIC - think about it, but not for too long!).

STATIC Storage

STATIC storage is assigned space at compilation time and this space is allocated in computer storage during the loading of the program; the contents are also initialized at compilation time if the INITIAL attribute is specified. The storage for STATIC variables is not released until the program terminates.

Example:

```
     PROG: PROC;
           DCL WORD CHAR(3) INIT('OLD') STATIC;
           •
           PUT LIST (WORD);
           •
           WORD='NEW';
           PUT LIST(WORD);
     END;
```

The first executable statement is a stream output statement which will print out 'OLD', and the second output statement will print out 'NEW'.

Note that if the procedure PROG is executed again later, during the same program, the DCL statement will **not** cause WORD to be re-initialized, since initialization of static variables is determined at compile time. Provided no new assignments have been made in the interim period, the first PUT statement will now print out 'NEW'.

AUTOMATIC Storage

The AUTOMATIC attribute is provided to save total storage requirements for the variables used in multi-block programs. Its usefulness relies on the fact that variables often require storage only for the duration of the block in which they are declared.

AUTOMATIC variables are allocated storage when the block which contains their declarations is activated; initialization, if any, also takes place at this time. When the block is terminated, the storage for its AUTOMATIC variables is released, and made available for dynamic storage for other blocks. (When AUTOMATIC storage is released, the current values of the variables are lost).

Every time a block is activated, its AUTOMATIC variables are initialized by the item in the INITIAL attribute (if any). If WORD in the example above had been allowed to default to AUTOMATIC storage class, it would have been reinitialized to 'OLD' every time the procedure PROG had been activated.

AUTOMATIC vs STATIC

Use STATIC whenever a variable needs to hold its value throughout a multi-block program or to improve execution time (no storage needs to be allocated at execution time as with AUTOMATIC). Use AUTOMATIC to save total computer storage requirements for multi-block programs.

BEGIN Blocks

Up till now we have restricted our discussion to procedure blocks. It is appropriate now to consider the other type of block which PL/I provides - the begin block. The begin block must appear within a procedure block.

A begin block is a set of statements delimited by BEGIN and END statements:

```
   [label] : BEGIN;
           •
           •
    END [label];
```

Unlike a procedure block, a label is optional for a begin block. This is because control passes into a begin block **sequentially,** following execution of the preceding statement. A begin block cannot be CALLed; it is activated and executed as part of normal sequential program flow or by a GOTO statement.

This is the property that distinguishes begin blocks from internal procedures. Apart from this the two types of block have much in common: the rules that govern the allocation of storage and the scope of names for internal procedures also apply to begin blocks. (Note that the use of begin blocks as on-units is a special case - see Topic 19).

Example:

```
 CUSPROC:PROC OPTIONS (MAIN);
         FIRST:BEGIN;
               DCL 1 STRUC,
                      2 AMOUNT DEC FIXED (5),
                      2 NAME CHAR(77);

               DCL TOTAL      DEC FIXED (7);
               •
               •
               •
         END FIRST;
         SECOND:BEGIN;
                DCL   ARRAY(1Ø) BIN FIXED (15);
                DCL   BIT1  BIT(1);
                •
                •
         END SECOND;
         •
         CALL EXTPROC;
         •
 END CUSPROC;
```

When the begin block FIRST is activated the storage is allocated for STRUC, TOTAL and any other AUTOMATIC variables declared within FIRST. This storage is released when FIRST is deactivated (by the END FIRST; statement), and so can be reused by any AUTOMATIC variables within SECOND or within the external procedure EXTPROC when this procedure is invoked.

Situations where it is worthwhile to use begin blocks in this manner do not often occur in programs. The main use of begin blocks is as on-units for handling PL/I on-conditions (see Topic 19).

Prologues and Epilogues

The use of blocks in a PL/I program involves a slight overhead of both time and space. When a block is activated, certain activities must be performed before control can be given to the first statement in the block. These activities are performed by a compiler-written routine called the **block prologue.** The prologue is appended to the beginning of the block. It performs essential housekeeping functions such as allocating storage for AUTOMATIC variables (including initialization if this is specified).

Similarly when a block terminates certain actions must be carried out before control is transferred out of the block. A **block epilogue** is provided by the compiler for this purpose. The epilogue is appended to the end of the block and one of its functions is to release the storage occupied by AUTOMATIC variables.

Further information about the functions of prologues and epilogues can be found in the Language Reference Manual.

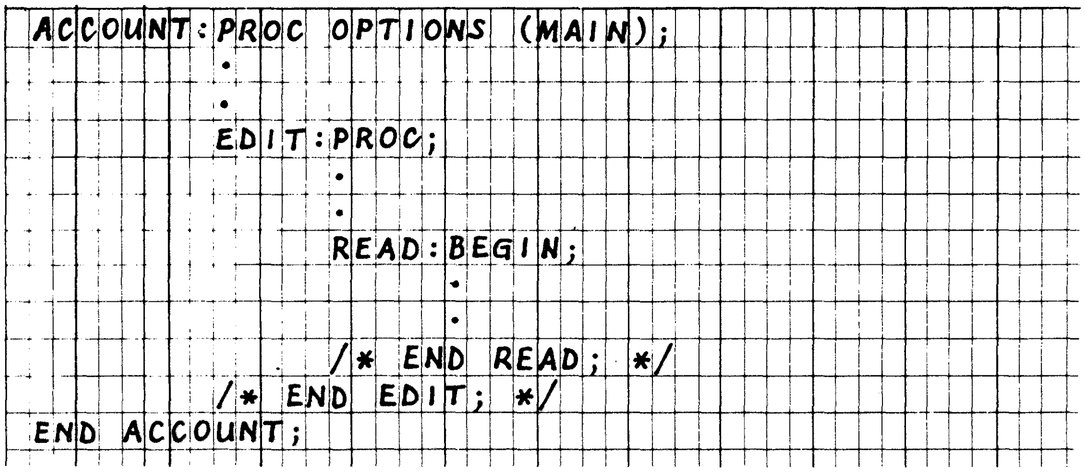
In most cases the **advantages** of 'building' a program from blocks (see next topic on subroutines and functions) far outweight the slight overheads mentioned above.

Multiple Closure of Nested Blocks and Do-Groups

Normally an END statement is matched up with the nearest preceding BEGIN, PROCEDURE or DO statement which does not already have an END statement; its effect is to 'close' (i.e. end) a block or a DO-group.

When the END statement is followed by the optional label, its power is considerably increased. It closes the block or DO-group which has a matching label, **and all intermediate (nested) unclosed blocks or DO-groups.** This is known as **multiple closure.**

Example:



```
ACCOUNT:PROC OPTIONS (MAIN);
        .
        .
        EDIT:PROC;
             .
             .
             READ:BEGIN;
                  .
                  .
             /* END READ; */
        /* END EDIT; */
END ACCOUNT;
```

In the example above, the statement END ACCOUNT; closes the external procedure ACCOUNT, the internal procedure EDIT and the begin block READ. (The END statements, written as comments, would have produced the same effect if written as actual statements).

Multiple closure is **not** recommended because it impairs program documentation. Blocks and do-groups should be explicitly closed. Multiple closure can also affect program logic in situations where an 'inner' END statement has been omitted. Beware of this.

The use of labels on END statements is quite useful as a documentation aid.

Exercises

1. Why does PL/I provide the facility to code a program in **blocks?**
2. What is the essential difference between an external procedure and an internal procedure?
3. Describe one way in which data variables may be referred to by more than one external procedure.
4.

```
  WATER: PROC;
        DCL A FIXED DEC(5);
        DCL B CHAR(2);
        DCL X CHAR(8Ø) BASED (PTR);
        I, J=Ø;
        .
        .
        MELON: PROC;
              DCL A FIXED DEC(5);
              DCL I DEC FLOAT (6);
              DCL PIP BASED (Q);
              J, N=1ØØ;
              .
        END MELON;
  END WATER;
```

Write down the scope of the following variables:

A, B, X, I, PTR, J, PIP, Q, N

(Note: Distinguish between different uses of the same name where applicable).

5. How is a procedure block normally activated?
6. How is a procedure block terminated?
7. Give one advantage of using
   a) external procedures

   and

   b) internal procedures
8. What is the feature that distinguishes a begin block from an internal procedure?
9. a) How are begin blocks terminated?

   b) What is the most common use of begin blocks?

10. Name one function performed by

    a) the block prologue.

    b) the block epilogue.

11.

```
       X:PROC;
         •
         Y:PROC;
           •
           Z:BEGIN;
             •
             •
       END X;
```

What is the effect of the END X; statement?

(It is the only END statement in the coding).

12.

```
TRICK:PROC;
      DCL (I,N) BIN FIXED (15);
      N=Ø;
      DO I = 1 TO 5;
                   N=N+1;
                   CALL INTPROC;
                END;
      INTPROC:PROC;
              DCL ARRAY (5) CHAR(2);
              •
              DO I = 1 TO 5;
                           ARRAY(I)=ARRAY(I)+2;
                        END;
      END INTPROC;
END TRICK;
```

What is the value of N after executing TRICK?

Answers

1. To facilitate modular programming (see Topic 18).
2. An internal procedure is **nested** (contained) within a containing procedure.
3. By use of the EXTERNAL attribute.
4.

Variable	'Known' In
A (declared in WATER)	WATER
A (declared in MELON)	MELON
B (explicit)	WATER and MELON
X (explicit)	WATER and MELON
I (implicitly declared in WATER)	WATER
I (explicitly declared in MELON)	MELON
PTR (contextual)	WATER and MELON
J (implicit)	WATER and MELON
PIP (explicit)	MELON
Q (contextual)	WATER and MELON
N (implicit)	WATER and MELON

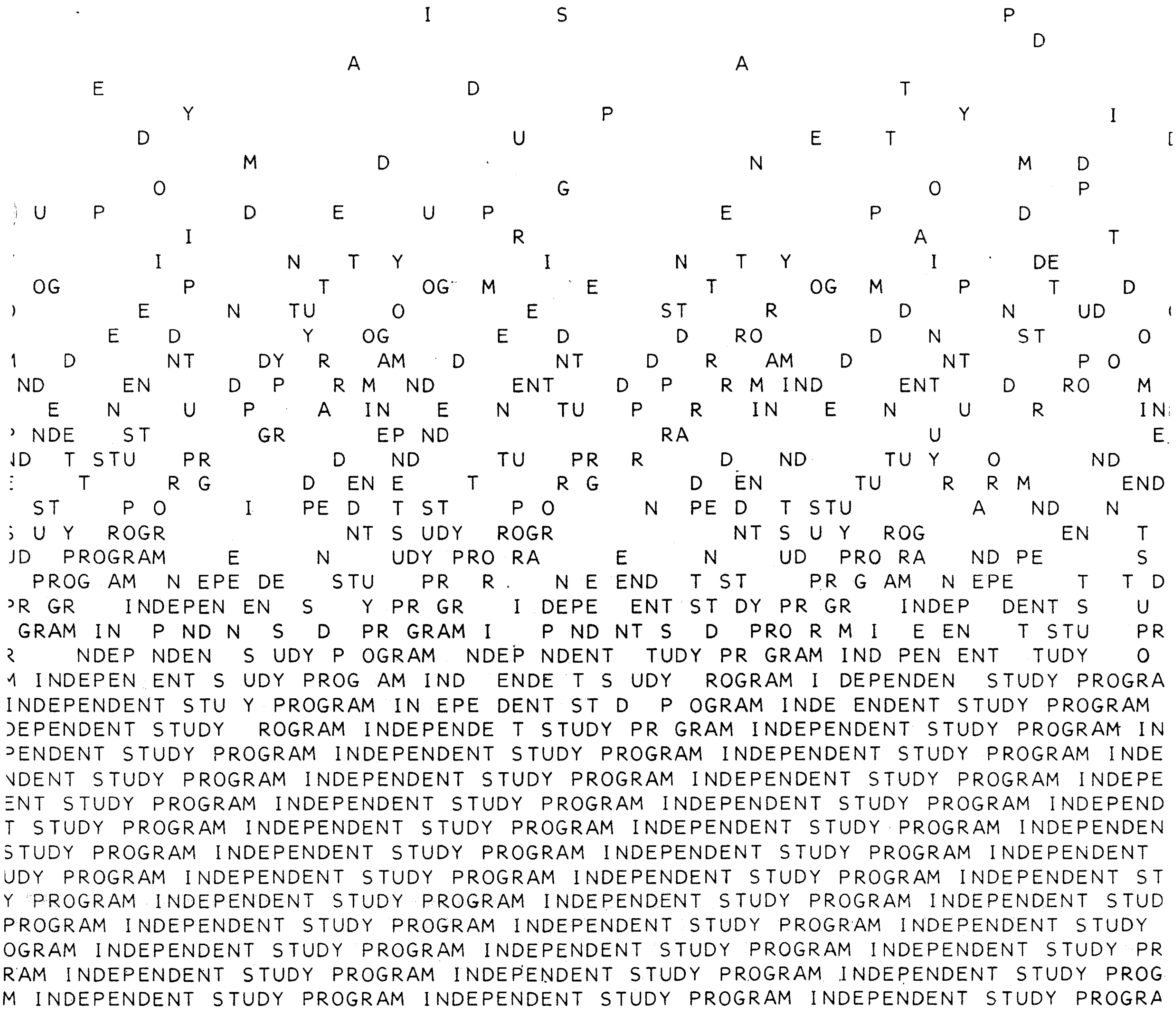
5. By **procedure invocation,** for example, by using a CALL statement.
6. By use of the RETURN or the END statement.
7. a) External procedures can be compiled and tested independently.

   b) An internal procedure can share variables with the containing procedure.
8. Control passes into a begin block during **normal sequential flow** or by a GOTO statement; an internal procedure must be **invoked.**
9. a) when the END; statement for the block, or a GOTO statement which specifies a label outside the block, is reached.

   b) as on-units (we will come to these in Topic 19).
10. a) Allocation of AUTOMATIC storage.

    b) Releasing storage occupied by AUTOMATIC variables.
11. It closes the external procedure X, the internal procedure Y, and the begin block Z.
12. The final value of N is 1. The first DO loop is only executed once. There is only one variable called I in the whole of procedure TRICK (see scope of implicitly declared variables). The value of I will be 6 at the end of the DO loop within INTPROC (see the DO statement expansion in section J of the Language Reference Manual). Thus, on return from INTPROC, the first DO loop will be terminated.

Topic 18



Topic 18

Subroutines and Functions

In this topic you will learn about the advantages of modular programming and how this is implemented in PL/I using procedure blocks; these have been described in Topic 17. In this topic you will learn about the two types of procedure block - subroutines and functions.

Objectives

At the end of this topic you should be able to:

- state the advantages of modular programming
- state how modular programming is implemented in PL/I
- state when you would use a function procedure and when you would use a subroutine procedure
- code subroutines and functions, sharing data as required
- code a subroutine or function with secondary entry points
- use PL/I builtin functions.

Introduction

Modular programming requires programs to be written in units or modules. In PL/I these modules are procedure blocks. We will look at the two kinds of procedure block - subroutine and function. We will discuss the reasons for using one or other type and explain how control is transferred to a subroutine or function procedure (and how control returns to the 'calling' procedure). The previous topic on block structure explained the flow of control for subroutine procedure blocks and so this will merely be summarized in the present topic. There will be more information on the flow of control for function procedure blocks because these will be new to you.

Later in the topic we will look at how data can be shared between procedure blocks - subroutine or function - and discuss a few problems which might arise.

The final part of the topic is about various functions which are commonly required during programming and which PL/I provides - these are the PL/I Builtin Functions.

The Advantages of Modular Programming

The usefulness of dividing a program into separate modules (or procedures) which can be written and tested separately is well recognized by experienced programmers. There are so many advantages to be gained by doing this that it is difficult to imagine any reasonably-sized program **not** being written in modules. The most important benefits of modular programming are:

1. It gives the program a sound logical structure.
2. It provides a disciplined method of working.
3. It ensures the provision of comprehensive documentation. Moreover, this is produced as part of the programming process - not as a separate task.
4. Writing and testing are easier because:
   a) Each module is a logical entity which may be written with very little reference to other modules; at any one time the programmer can concentrate on **part** of the program without having to consider all of it.
   b) Different modules can be tested independently; once a module is functioning satisfactorily it can be placed in a program library as an object module and subsequently linked into a program.
   c) Parts of a program can be tested while the rest is being coded, thereby making better use of machine time.
5. Program maintenance is less arduous and more efficient because:
   a) up-to-date documentation is available.
   b) amendments can be made at the module level; since a module is self-contained, alterations to it will not normally have unexpected repercussions elsewhere.
6. Large programs may be brought quickly to the production stage by dividing the modules between several programmers.
7. Trainees may be usefully employed in coding the small, straightforward modules of a large program.
8. Good control can be exercised over a programming project when it is broken down into several separate tasks; this allows progress to be effectively monitored.
9. Standard routines may be written once and then incorporated in several programs.
10. Different modules can be written in different languages. (For instance, it might be **necessary** to write a particular module in Assembler Language; the other modules in the program could be written in PL/I).

Modular programming has some disadvantages too. It carries overheads of both time and space. This is because the compiler includes a prologue and an epilogue in each procedure (see Topic 17) which means that there is extra code to be executed.

It is impossible to generalize about the maximum or minimum size (in terms of executable statements) for a module. One very functional maximum would be the maximum number of executable statements which can be contained on one page of computer listing. This is convenient because you do not have to turn over any listing pages when reading through the program - but it is **not** a rule. There is no functional minimum size for a module - it is up to the programmer to decide.

Procedure Invocation

PL/I allows the programmer to write two distinct kinds of procedures (modules) - **subroutines** and **functions.** (The examples of procedure blocks in the previous topic on Block Structure have been subroutines). Both subroutines and functions are written as separate procedures, which may be external or internal and both are said to be **invoked** when receiving control from some other procedure (the **invoking procedure**). A subroutine is invoked by means of a CALL statement; a function is invoked by **use** of the function name in a statement.

ENTRY Attribute

The ENTRY attribute is used to declare a procedure as an external procedure. It obviously does not apply to internal procedures - they need no declaration at all.

If an external procedure name is undeclared but PL/I can detect that the name is being used as a procedure name (either subroutines or function) then PL/I will **assume** that the name is an **external** procedure name. However, PL/I will give a diagnostic message. (Remember that the compiler compiles each external procedure separately. So, while it is compiling one external procedure, it does not **know** of the existence of any other external procedures and therefore can only make the **assumption** that any undeclared procedure, which is not internal to the invoking procedure, is an external procedure which will be provided at linkage-edit-time).

It is **invalid** to write a declaration for a procedure which is **internal** to the invoking procedure. PL/I can **see** that the procedure is **internal** (declaring a procedure with the ENTRY attribute means that the procedure is **external!).**

CALLing a Subroutine

This has been covered in a previous topic. Basically a CALL statement transfers control to the invoked procedure. Control is returned to the invoking procedure (to the statement after the CALL statement) by the END statement in the invoked procedure. There are other ways of returning control to the invoking procedure (e.g. the RETURN statement) - see previous topic.

Function Reference

A procedure is invoked as a function by using the function name in a statement.

The significant point about a function reference is that it represents a **single value.** Consider the following example:

```
       PI:PROC;
       .
       .
       I=FUNC+J;
       .
       .
       END PI;
```

```
     FUNC:PROC;
          .
          .
          .
          .
          RETURN(3.14*N);
     END FUNC;
```

FUNC is an external function procedure (it could equally well be internal). It is invoked in P1 by reference to its name in the expression FUNC + J. This has the effect of passing control to the function procedure, which computes a single value and returns this value to P1. The expression FUNC + J is then evaluated using this returned value, and the result is assigned to I. In effect therefore, a function reference in an expression is **replaced** by a single value.

A special form of the RETURN statement is used to cause a return from a function procedure. The keyword RETURN is followed by a parenthesized element-expression which specifies the function value to be returned to the invoking procedure. In our example we return the value (3.14 * N) to P1. The effect is the same as if we had written:

```
       P1:PROC;
          .
          .
          I=3.14*N+J;
          .
          .
       END P1;
```

(This assumes that N is known in P1). This form of the RETURN statement is **always** used in a function procedure and **never** used in a subroutine; it is the one thing that always serves to differentiate between the two.

Like an ordinary element variable, a function has attributes. These are the attributes of the single returned value, and if they are not explicitly declared then default attributes will be applied in the usual way, **according to the first letter of the function name.**

Even if the default attributes are those which are required it is still sensible to declare them explicitly (for program documentation). They are specified in two places:

1. with the **RETURNS attribute** in a DECLARE statement in the invoking procedure.
2. with the **RETURNS option** in the PROCEDURE statement or ENTRY statement (see later) in the function procedure.

Suppose that we wish FUNC to return a value with the attributes FIXED BINARY (15). We would specify this as follows:

In the Invoking Procedure

```
P1: PROC;
   DCL FUNC ENTRY RETURNS(BINARY FIXED(15));
   .
   .
   I=FUNC+J;
   .
   .
END P1;
```

In the Function Procedure

```
  FUNC: PROC RETURNS(BINARY FIXED(15));
        .
        .
        .
        RETURN(3.14*N);
  END FUNC;
```

In P1 the RETURNS **attribute** specifies that FUNC will return a value whose attributes are FIXED BINARY (15). So, when P1 is being compiled, a FIXED BINARY (15) field will be set up within P1 to receive the returned value when the program is executed. In FUNC itself the RETURNS **option** specifies that the attributes of the value returned by the RETURN statement will also be FIXED BINARY (15). Thus at execution time, the attributes of the value returned from FUNC will match the attributes of the field set up in P1 to receive the returned value.

If there had been no RETURNS attribute in P1 then the attributes of the **field** set up in P1 to receive the returned value would be DEC FLOAT (6) - according to the first letter of the function name. If there had been no RETURNS option in FUNC the returned **value** would also have had attributes DEC FLOAT (6) - according to the first letter of the function name.

For an external function procedure both the RETURNS option and the RETURNS attribute should be specified (or both allowed to default to the same attributes) to ensure correct execution. Remember that external procedures are compiled separately so, when PL/I is setting up the field to receive the returned value (in the invoking procedure), it cannot see the external function nor what attributes the returned value will have. The compiler therefore has to be told what attributes to give to the field in the invoking procedure by means of the RETURNS attribute or by default.

If the function is an internal procedure only the RETURNS option in the function procedure statement need be specified (or allowed to default):

```
       P1:PROC;
          .
          .
          I=FUNC+J;
          .
          .
          FUNC:PROC RETURNS(BINARY FIXED(15));
               .
               .
               RETURN (3·14*N);
          END FUNC;
       END P1;
```

This is because the compiler can 'see' the invoking procedure and the function procedure together, and it does not need to be given the same information twice. In fact, declaring FUNC in P1 with the RETURNS attribute by means of a DCL statement would in this case constitute an error.

Sharing Data Between Procedures

Usually when control is passed to a procedure block (subroutine or function) some information (i.e. data) is made available to the invoked procedure. There are several ways in which data can be shared in this manner.

If the invoked procedure is internal to the invoking procedure then any data declared in the invoking procedure can be referred to within the internal procedure (see scope of names in topic 17).

Example:

```
    FIRST:PROC;
          DCL NUMBER     FIXED DECIMAL(5);
          DCL UNITPRICE  FIXED DECIMAL(7,2);
          DCL TAX        FIXED DECIMAL(5,2);
          DCL TOTAL      FIXED DECIMAL(9,2);
          DCL GTOTAL     FIXED DECIMAL(11,2);
          .
          .
          CALL COST;
          GTOTAL=GTOTAL+TOTAL;
          .
          COST:PROC;
               .
               TOTAL=NUMBER*(UNITPRICE+TAX);
               .
          END COST;
    END FIRST;
```

If the invoked procedure is external to the invoking procedure then the data can be declared EXTERNAL (see previous topic).

Example:

```
FIRST:PROC;
      DCL NUMBER       FIXED DECIMAL (5)    EXTERNAL;
      DCL UNITPRICE    FIXED DECIMAL (7,2)  EXTERNAL;
      DCL TAX          FIXED DECIMAL (5,2)  EXTERNAL;
      DCL TOTAL        FIXED DECIMAL (9,2)  EXTERNAL;
      DCL GTOTAL       FIXED DECIMAL (11,2);
      .
      .
      CALL COST;
      GTOTAL=GTOTAL+TOTAL;
      .
END FIRST;
```

```
 COST:PROC;
      DCL NUMBER       FIXED DECIMAL (5)    EXTERNAL;
      DCL UNITPRICE    FIXED DECIMAL (7,2)  EXTERNAL;
      DCL TAX          FIXED DECIMAL (5,2)  EXTERNAL;
      DCL TOTAL        FIXED DECIMAL (11,2) EXTERNAL;
      .
      TOTAL=NUMBER*(UNITPRICE+TAX);
      .
 END COST;
```

In the above example NUMBER, UNITPRICE, TAX and TOTAL in FIRST, will refer to the same areas of storage, at execution time, as NUMBER, UNITPRICE, TAX and TOTAL in COST. This is a perfectly valid method of sharing data. However the external variables must be declared with the same names in both procedures. This might not be a problem when there are few data items involved, but in a modular programming environment, where different people might write each external procedure, then this limitation might be a significant restriction.

Also external variables are STATIC (see previous topic), i.e. they occupy storage from compilation time onwards. So if the external procedures FIRST and COST were tested separately, they would be stored separately on disk and within each procedure there would be storage for NUMBER, UNITPRICE, TAX and TOTAL. Thus the amount of disk space required during testing is increased. (You may be wondering when these separate areas for each external variable become 'combined' into one area. This takes place at linkage-edit-time. The linkage editor takes just one area for each external variable and incorporates that area into the final program. So, at **execution time,** any procedure which has a declaration for an external variable will be referring to the same area of storage as any other procedure with a similar external declaration for that variable).

Both the limitations of the previous method of sharing data can be overcome by specifying, in the invoking procedure, a list of data items to be shared (called an **argument list)** and associating these data items with corresponding names in a list (called a **parameter list)** in the invoked procedure.

Both a CALL statement and a function reference can specify an argument list in order to make data available to the invoked procedure. The data items in question are written in a parenthesized list following the entry name in a CALL statement or the function name in a function reference. For example:

```
          CALL SUB (A,B);
```

This CALL statement invokes the procedure SUB and specifies that two arguments, A and B, are to be passed to it.

```
          X=FUNCX(P,Q,R);
```

This statement invokes the function FUNCX and passes three arguments to it.

Arguments which are passed to a procedure are 'received' in that procedure by specifying a parameter list with the corresponding PROCEDURE or ENTRY statement. Thus the parameters corresponding to A and B might be specified with the PROCEDURE statement as follows:

```
      SUB:PROC (C,D);
```

The parameters corresponding to P, Q and R might be specified as:

```
    FUNCX:PROC (X,Y,Z);
```

The number of arguments should be the same as the number of parameters; they are matched one for one, from left to right in the lists. The parameter should be declared in the invoked procedure with the same attributes as the corresponding argument in the invoking procedure. However storage is only allocated for arguments in the invoking procedure; **parameters occupy no storage.** In fact the address of the argument is passed to the invoked procedure and this address is used in the invoked procedure whenever the corresponding parameter is referred to. This overcomes one limitation of the previous method of sharing data using external variables. Also the names of arguments and their associated parameters need not be the same - this overcomes the second restriction of external variables.

By reference to its parameters, a subroutine or function may obtain values from the procedure that invoked it. In addition, arguments may be provided for the purpose of receiving values returned from a CALLed subroutine (this technique would not normally be applied to a function reference since a function is commonly used to return only a **single** value, by means of the RETURN statement).

Example:

```
   FIRST:PROC;
         DCL (ARG1,ARG2)     FIXED DEC (5);
          .
         CALL SUB(ARG1,ARG2);
          .
   END FIRST;
```

```
     SUB:PROC (I,J);
         DCL  (I,J)          FIXED DEC (5);
          .
         I=J**2;
          .
     END SUB;
```

The above subroutine calculates the square of ARG2 and places its value in ARG1. I and J do not occupy any storage within SUB.

Non-matching Attributes for Arguments/Parameters

Suppose that the previous example had been coded as follows:

```
    FIRST:PROC;
          DCL (ARG1,ARG2)     FIXED DEC (5);
          .
          CALL SUB (ARG1,ARG2);
          .
    END FIRST;
```

```
      SUB:PROC(I,J);
          DCL I               FIXED DEC (5);
          .
          I=J**2;
          .
      END SUB;
```

J has not been explicitly declared and so takes the default attributes of BINARY FIXED (15). Now FIXED DECIMAL (5) occupies three bytes and BINARY FIXED (15) occupies two bytes. So, when we refer to J in SUB above we are actually referring to the first two bytes of ARG2 and interpreting them as if they were BINARY FIXED (15)! This will obviously not give the right result at execution time and may even cause a program interrupt.

In the previous example the mismatch of attributes was a mistake but there are circumstances in which an argument has certain attributes for particular reasons (e.g. efficiency) and the parameter in the invoked procedure has different attributes. This might arise if a previously written procedure were being incorporated into a new program; the 'old' procedure might contain parameters which had different attributes to those required. Of course, the 'old' procedure could be altered and recompiled so that the parameters had the correct attributes.

Another solution is to use a **parameter descriptor list** which describes the attributes of the parameters. This parameter descriptor list is attached to the declaration of the external procedure, e.g.

```
FIRST:PROC;
     DCL (ARG1,ARG2)          FIXED DEC (5);
     DCL SUB  ENTRY (FIXED DEC(5),BIN FIXED(15));
     .
     CALL SUB (ARG1,ARG2);
     .
END FIRST;
```

```
  SUB:PROC(I,J);
     DCL I                    FIXED DECIMAL (5);
     .
     I=J**2;
     .
  END SUB;
```

In the above coding SUB is declared as an external procedure whose parameters have attributes FIXED DECIMAL (5) and BINARY FIXED (15) respectively.

During the compilation of FIRST, PL/I would then realize that the attributes of ARG2 (DECIMAL FIXED (5)) did not match those of the associated parameter (BINARY FIXED (15)). So PL/I would set up a 'dummy' argument field which did have the correct attributes and it would arrange to place the value in the original field into the dummy field, making the necessary conversions. The address of this 'dummy' field would be passed across to the invoked procedure at execution time. You can see from this that any operations on the parameter associated with this dummy field would only affect the dummy field and not the original field in the invoking procedure. For instance, if we had said:

```
J=J**2;
```

in SUB in the previous example, the value of ARG2 would remain unaltered - the 'dummy' field set up for ARG2 would contain the square of ARG2. Beware of this!

The parameter descriptor list could equally well have been written as:

```
DCL SUB ENTRY (,BINARY FIXED(15));
```

The first item in the parameter descriptor list is a **null specification;** indicating that the attributes of the first argument match those of the first parameter. Null specifications are indicated by a comma.

It is important to realize that the problems caused by non-matching arguments and parameters do **not** arise when the invoked subroutine or function is an internal procedure. Thus, if we make SUB internal to FIRST, there is no need to declare the attributes of the parameters in a parameter descriptor list:

```
FIRST:PROC;
      DCL (ARG1,ARG2)     FIXED DEC (5);
      .
      CALL SUB (ARG1,ARG2);
      .
      .
      SUB:PROC (I,J);
          DCL I           FIXED DEC (5);
          .
          I=J**2;
      END SUB;
END FIRST;
```

In this case the compiler can 'see' that the attributes of the argument ARG2 do not match those of J, the corresponding parameter. It will therefore **automatically** create a dummy argument with the correct attributes to replace ARG2.

The previous examples have used subroutines to illustrate the relationship of arguments and parameters.

The same rules apply also to functions. It should be noted that a function PROCEDURE may need to specify both a parameter list **and** the attributes of the returned value. For example, suppose that FUNCX will be passed three arguments and is required to return a value with the attributes FIXED DECIMAL (5,1). This could be specified as follows:

```
FUNCX:PROC (X,Y,Z) RETURNS (FIXED DEC (5,1));
```

Dummy Arguments

We saw previously that a dummy argument was created when the compiler was able to detect (or was told) that the attributes of the argument did not match those of the corresponding parameter. A dummy argument is also created whenever a constant or an expression is used as an argument. For example:

```
AMOUNT=FUNC(TOTAL,12.5,'A',NUMBER*UNITP);
```

A dummy field is set up for the decimal constant 12.5 and the character string 'A' and also for the result of the expression NUMBER*UNITP. The addresses of these dummy fields are passed across to the invoked procedure (in this case a function).

You can 'force' the creation of a dummy argument be placing parentheses around an argument field. For example:

```
    CALL INCREASE ((MY_SALARY));
```

A dummy argument would be set up for MY__SALARY and code would be inserted to place the value of MY__SALARY into its dummy field. The address of this dummy field would be passed across to INCREASE at execution time. So it would be impossible to alter the original value of MY__SALARY from within the subroutine INCREASE. This technique is useful for fields which you want to share with other procedures for read-only purposes but which you do not want to be altered, accidentally or otherwise!

Primary and Secondary Entry Points

The label attached to a PROCEDURE statement constitutes the primary entry point of that procedure. We have seen that a procedure can be invoked at its primary entry point by either a CALL statement or a function reference which specifies the procedure name. In addition it is possible to specify secondary entry points in the invoked procedure by means of **ENTRY statements.** Do not confuse this with the ENTRY attribute (see example below). Secondary entry points apply equally to subroutines and functions.

Example:

```
 MAIN: PROC OPTIONS (MAIN);
       DCL (CIRCLE,TRAP,RECT) ENTRY;
       DCL (RADIUS,X,Y,Z,L,M,AREAC,AREAT,AREAR) FIXED DEC(3);
       .
       CALL CIRCLE (RADIUS,AREAC);
       .
       CALL TRAP (X,Y,Z,AREAT);
       .
       CALL RECT (L,M,AREAR);
       .
 END MAIN;
```

```
CIRCLE:PROC (R,AREA);
       DCL (R,A,B,C,AREA)        FIXED DEC (3);
       AREA=3.14*R*R;
       RETURN;
  TRAP:ENTRY (A,B,C,AREA);
       A=(A+B)/2;
  RECT:ENTRY (A,C,AREA);
       AREA=A*C;
END;
```

CIRCLE, TRAP and RECT are explicitly declared with the ENTRY **attribute** in MAIN. CIRCLE is the primary entry point. TRAP and RECT are secondary entry points and are explicitly declared as such by appearing as the labels on ENTRY **statements** within CIRCLE. Basically the procedure labelled CIRCLE calculates the area of a circle, trapezium (quadrilateral with two sides parallel) or rectangle, depending on which entry point is used. If we enter the procedure at the primary entry point (CIRCLE) we work out the area of the circle and then return to MAIN before the TRAP entry point. However, if we enter at the TRAP entry point, we work out the average length of the two parallel sides and then 'drop through' the RECT entry point to multiply this average length by the perpendicular distance between the parallel sides (C). When you 'drop through' an ENTRY statement during normal sequential execution of instructions, the ENTRY statement is treated as a comment. (Do not worry about the geometry, just the PL/I).

Use of Functions and Procedures

You should have some idea now about when you would use a function and when you would use a subroutine. Subroutines are used to divide the total program logic into smaller units (the subroutines). Functions are used to return a single value to the point of invocation.

PL/I Built-in Functions

There are many functions which are often required by many users of PL/I; it would be wasteful if they had to be coded by every programmer who required them. For this reason, with every PL/I compiler a useful set of precompiled functions is provided. These are called the PL/I **built-in functions.** They are described in section G of the Language Reference Manual. In this topic we will look at the use of a few built-in functions. When you have time (i.e. not now) look through section G of the Language Reference Manual to see what other built-in functions are available.

The built-in functions (let's call them b.i.f.s from now on) are divided into different categories:

string handling b.i.f.s	-	example later
arithmetic b.i.f.s	-	example later
mathematical b.i.f.s	-	no examples
array handling b.i.f.s	-	see Topic 22
condition handling. b.i.f.s	-	see Topic 19
stream I/O b.i.f.s	-	see Topic 15
storage control b.i.f.s	-	example later

Declaring Built-in Functions - the BUILTIN Attribute

In general it is not necessary to tell the compiler that a particular identifier is a built-in function; the name and its context will normally imply it.

In the statement

```
          Y = SQRT(2);
```

SQRT will be taken to mean the built-in function, **unless** it has been declared as something else. This is because the fact that it is followed by a parenthesized item (2) means that it could only possibly be a function or an array element, and if an array with the name SQRT has not been declared, the latter possibility is eliminated.

Unless the programmer has declared SQRT as a function which he has coded **himself** the compiler assumes that the built-in function is being referred to. This discussion is equally valid for all the other built-in functions which make use of arguments.

Certain built-in function names are not followed by parenthesized arguments, however, and so the compiler will not recognize them as built-in unless they are declared with the BUILTIN attribute. An example of these is DATE, which 'returns' a six character string. (This means that at execution time the name DATA is replaced, wherever it occurs, by the value which the function supplies).

Example:

```
         DCL FIELD CHAR(6);
         DCL DATE  BUILTIN;
         .
         FIELD=DATE;
```

Since DATA has been given the BUILTIN attribute, the built-in function will be invoked and the value it returns will be assigned to FIELD.

Alternatively a null argument list (an argument list with no arguments) could have been inserted after the DATE b.i.f. in the assignment statement. Then there would have been no need to declare DATA as BUILTIN.

Example:

```
         FIELD=DATE();
```

String-handling Built-in Functions

These b.i.f.s are used for manipulating character-strings or bit-strings. Look up the INDEX and SUBSTR b.i.f.s in section G of the Language Reference Manual. These b.i.f.s will be used in the following example.

```
DCL ADDRESS CHAR (5Ø);
DCL ( FIRST_LINE,SECOND_LINE) CHAR (25);
.
ADDRESS= '76 ABERCROMBIE ROAD*MIDVILLE MA';
I = INDEX (ADDRESS,'*');
FIRST_LINE= SUBSTR (ADDRESS,1,I-1);
SECOND_LINE = SUBSTR (ADDRESS,I+1);
```

The value of I will be 20 after execution of the INDEX b.i.f. The rest of the coding assigns the first and second lines of the address to corresponding fields. Notice that if the third argument for the SUBSTR b.i.f. is omitted, then the length is assumed to be to the end of the string.

Arithmetic Built-In Functions

Some of the b.i.f.s perform the simple operations which may otherwise be done with PL/I operators. Instead of *, / and +, the b.i.f.s MULTIPLY, DIVIDE and ADD may be used. The purpose is to avoid the possibility of loss of significant digits during complicated assignments, e.g. X=Y*Z+Z**P/Q/R; (in such cases intermediate target areas of **default** precision are used - by using the ADD, MULTIPLY and DIVIDE functions, however, any desired precision may be retained throughout the calculation).

As well as these three arithmetic b.i.f.s there are many others, some of which also permit the programmer to control precision. Arguments should be passed to arithmetic b.i.f.s in coded arithmetic form (i.e. binary or decimal), and values will also be returned in coded arithmetic form. If an argument is not in the correct form it will automatically be converted, where possible.

As an example of this type of b.i.f., look up the ROUND b.i.f. which rounds a value at a specified digit.

Example:

```
DCL (FLD1,FLD2) FIXED DEC (8,4);

FLD1 = 1234.5678;
FLD2 = ROUND (FLD1,2); /* FLD2 CONTAINS 1234.57ØØ */
FLD2 = ROUND (FLD1,Ø); /* FLD2 CONTAINS 1235.ØØØØ */
FLD2 = ROUND (FLD1,-2);/* FLD2 CONTAINS 12ØØ.ØØØØ */
```

Storage control b.i.f.s

We will only cover the ADDR b.i.f. under this heading. Look it up in the Language Reference Manual.

Example:

```
 DCL 1 STRUCA,
       2 CODEA    CHAR (1),
       2 QUANTITY    FIXED DEC (5),
       2 UNITPRICE   FIXED DEC (5),
       2 NAME     CHAR (73);

 DCL 1 STRUCB BASED (PTR),
       2 CODEB    CHAR (1),
       2 DESCRIPTION CHAR(79);

 PTR = ADDR(STRUCA);
```

The above coding 'overlays' the area of storage occupied by STRUCA with an alternative structure STRUCB. Thus two different 'record types', corresponding to STRUCA and STRUCB, could exist on the same data set and be read into the one area STRUCA; the record could then be interpreted with the appropriate structure depending on its code value in the first byte. There are other methods of achieving this overlaying; they will be described in Topic 21.

Pseudo-variables

Some built-in functions can also be used as pseudo-variables.

A pseudo-variable is a built-in function which is used to **insert** information into a variable, rather than to obtain information from it. **It is a built-in function in reverse. Those built-in functions which can be used as pseudo-variables are described in section G of the Language Reference Manual.**

Example:

```
 DCL VAR CHAR(6) INIT ('CHARGE');
 SUBSTR(VAR,4,1)= 'N';
```

The above illustrates the pseudo-variable use of SUBSTR. In effect it says 'Consider the one-character string beginning from the fourth character of VAR, and **assign 'N' to that position'**. (VAR will now contain 'CHANGE'). Contrast this with the **built-in function** use of SUBSTR.

```
 DCL LETTER CHAR (1);
 DCL VAR    CHAR (6) INIT ('CHARGE');
 LETTER = SUBSTR (VAR,4,1);
```

This says 'consider the one-character string beginning from the fourth character of VAR, and **assign it to the variable LETTER'**. (The variable LETTER will now contain the character 'R').

Although pseudo-variables may be used anywhere as a receiving field (e.g. in the data list of a GET statement in stream I/O) the typical usage is on the left of an assignment sign, as in the above example.

Exercises

1.

```
P:PROC OPTIONS (MAIN);
  DCL ARR (10) DEC FIXED (5);
  DCL TOTAL    DEC FIXED (9);
  CALL SUM (ARR,TOTAL);
  .
END;
```

Code the external subroutine SUM which returns in TOTAL the sum of the values stored in the elements of ARR. (Try not to make use of the SUM b.i.f.).

You may assume that at the time of the CALL these values have been correctly set up.

2.

```
Q:PROC OPTIONS (MAIN);
  DCL MARK FIXED DEC (3);
  DCL COMMENT   CHAR(4);
```

Code an external function ASSESS invoked from Q such that 'GOOD', 'FAIR', 'POOR', or '*****' is moved into COMMENT depending on whether MARK is greater than 75, 50, 25, or less than or equal to 25.

Also code the invoking statement and any others required in Q.

3.

```
  DCL QUOTE CHARACTER(155);
  DCL PHRASE CHARACTER(13);
```

Find the position of the string 'YORICK' in QUOTE.

Place the 13 characters immediately following 'YORICK' into PHRASE.

4. Look up the TRANSLATE b.i.f. in section G of the Language Reference Manual. What will the following coding place in FLD?

```
DCL FLD   CHAR(6) INIT('12310S');
FLD = TRANSLATE (FLD,'5Ø1','SOI');
```

5. Look up the MOD b.i.f. in section G. Write PL/I coding which uses the MOD b.i.f. to work out whether a particular year, contained in YEAR (declared below) is a leap year. Move the character strings 'YES' or 'NO' into RESULT (declared below) depending on the result. (Remember that a leap year is divisible by 4 exactly but century years, e.g. 1900, must be divisible by 400).

```
DCL YEAR   PIC'9999';
DCL RESULT CHAR(4);
```

Answers

1.

```
SUM:PROC(A,T);
    DCL A(1Ø) DEC FIXED (5);
    DCL T     DEC FIXED (9);
    T=Ø;  /* NOT NECESSARY IF INITIALISED IN P */
    DO I= 1 TO 1Ø;
                 T=T+A(I);
              END;
END;
```

2.

```
     Q:PROC OPTIONS (MAIN);
       DCL ASSESS ENTRY RETURNS (CHAR(4));
       .
       .
       COMMENT = ASSESS (MARK);
       .
     END;
```

```
ASSESS:PROC (M) RETURNS (CHAR(4));
       DCL M  FIXED DEC (3);
       IF M > 75 THEN RETURN ('GOOD');
       IF M > 50 THEN RETURN ('FAIR');
       IF M > 25 THEN RETURN ('POOR');
       RETURN ('****');
END;
```

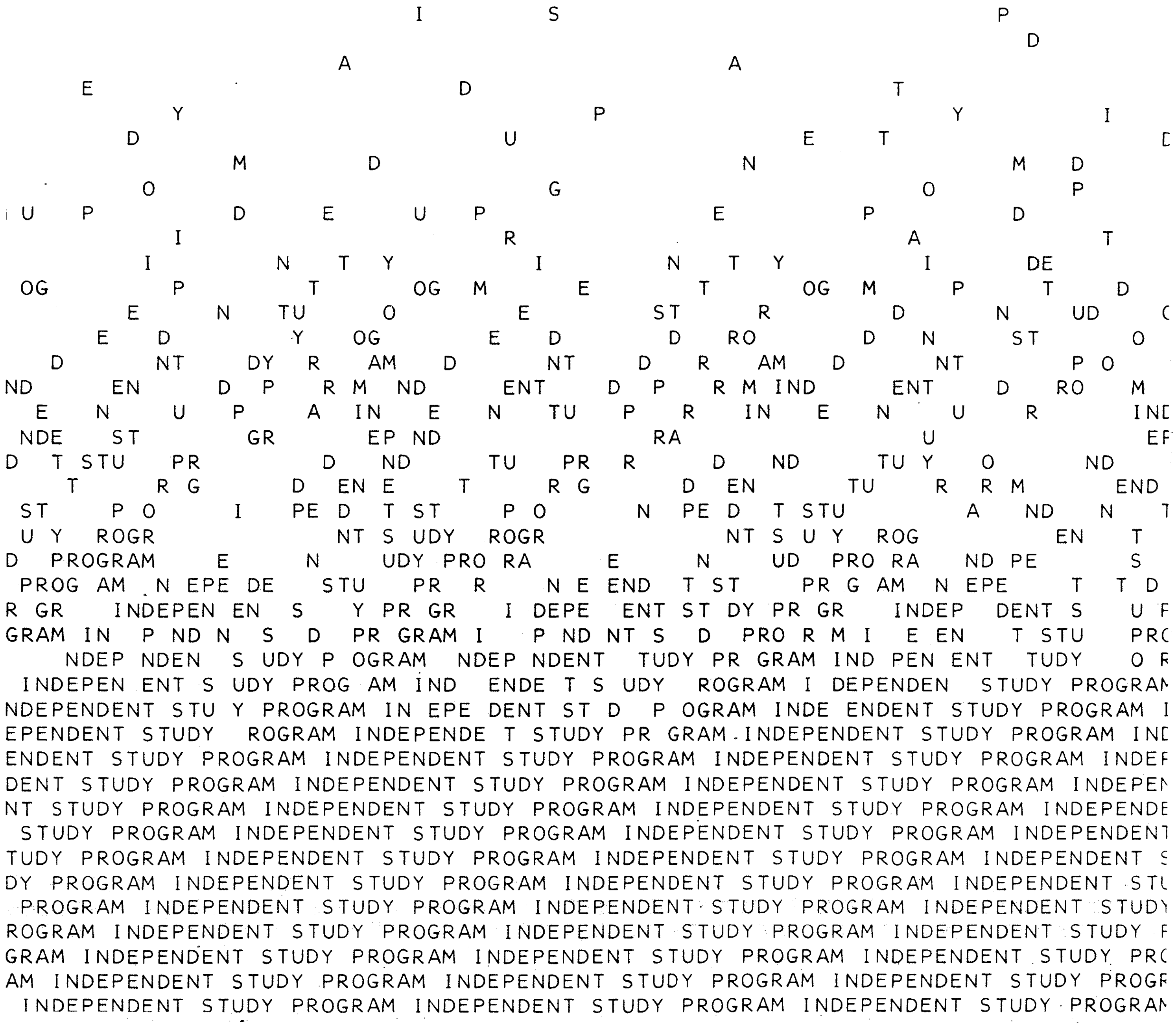
3.

```
I = INDEX (QUOTE,'YORICK');
IF(I > Ø) & (I < 138) THEN PHRASE = SUBSTR(QUOTE,I+6,13);
/* IF I=Ø THEN  'YORICK' IS NOT WITHIN QUOTE */
```

4. FLD will contain 123105 (all numeric).
5. The following solution uses the SELECT statement. The solution could have been written using IF statements but this would have made the logic less easy to follow.

```
SELECT(MOD(YEAR,1ØØ));
  WHEN(Ø) SELECT(MOD(YEAR,4ØØ));    /* CENTURY YEAR */
            WHEN(Ø) RESULT = 'YES';
            OTHERWISE RESULT = 'NO';
          END;
  OTHERWISE SELECT(MOD(YEAR,4));
              WHEN(Ø) RESULT = 'YES';
              OTHERWISE RESULT = 'NO';
            END;
END;
```

Topic 19



Topic 19

Handling Exceptional Conditions

When there is an interrupt to the normal flow of a program the system has a standard default course of action. This topic describes how the programmer can code suitable statements to override this action and to take other action more appropriate to the situation.

Objectives

At the end of this topic you should be able to:

- use the PL/I exceptional condition facilities to
  - (a) ENABLE/DISABLE conditions
  - (b) control the action to be taken when an ENABLED condition arises
- use the PL/I Built-in-Functions that are specifically designed to help in exceptional condition handling
- control the point at which execution is resumed after handling an exceptional condition
- state the effects of, and code, SIGNAL and REVERT statements
- state the function of the SNAP option of the ON statement.

Introduction

An EXCEPTIONAL CONDITION is a statement which causes interruption of the normal program flow. Some have already been met in earlier topics:

ENDFILE

KEY

RECORD

There are many more. A full list is given in section H of the Language Reference Manual. Although the meaning of some will be mentioned in passing, you are advised to look others up in the manual, to leave you thoroughly acquainted with them and to know in which circumstances they apply.

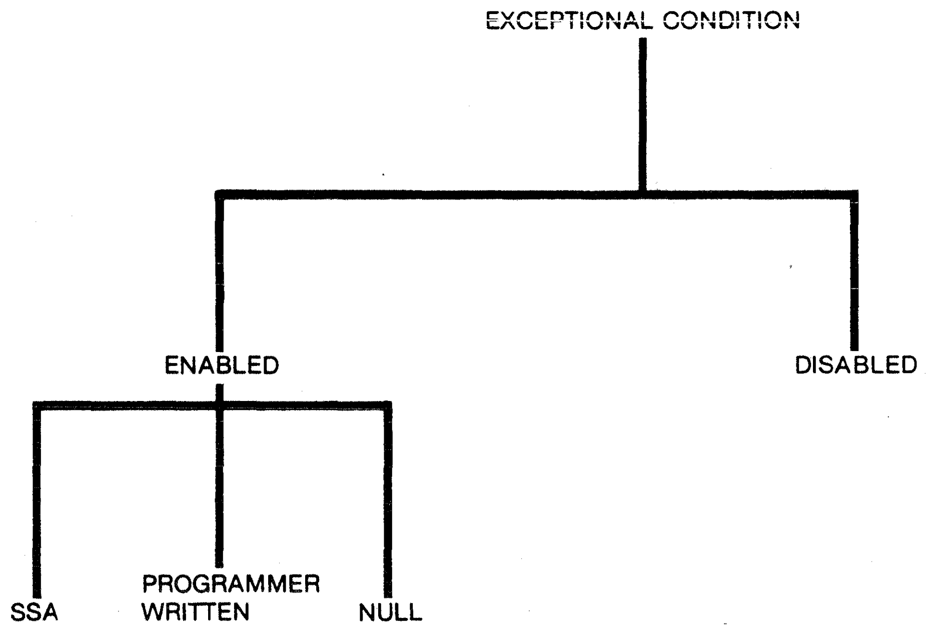
PL/I has a default course of action associated with each of these conditions. This is called STANDARD SYSTEM ACTION (SSA for short). This varies from condition to condition. Each SSA is listed in the manual under the appropriate condition heading.

However, it is possible, in our program, to code what action we wish to be taken should particular situations arise, e.g.

```
 ON ENDFILE(INFILE) SW='Ø'B;
```

This statement indicates that if the end of file marker for file INFILE is read then we are to set the bit field SW to zero rather than take the SSA which is to print a message and raise the ERROR condition (see later).

Enablement/Disablement



There are two possible states for an exceptional condition: ENABLED or DISABLED.

Enabled

An ENABLED condition is one which is being monitored throughout program execution, and when it occurs an interrupt will be caused and the program will have to take one of three actions:

1. STANDARD SYSTEM ACTION
2. a programmer written action
3. NULL action (see later)

Disabled

A DISABLED condition is one which is not being monitored and thus there will be no program interrupt even if the situation associated with the condition were to occur. For example, if ZERODIVIDE condition (raised by dividing by zero) is disabled, then even if within the program a division by zero occurs, no interrupt will be caused and the program will continue executing normally (even though the result of dividing by zero is unpredictable).

Categories

Exceptional conditions fall into one of three categories:

1. those which are disabled by default and can be enabled
2. those which are enabled by default and can be disabled
3. those which are enabled by default and can not be disabled.

Those in category 1 are known as RANGE CHECKING conditions and will be met in more detail in the topic named 'TESTING AND DEBUGGING AIDS'. Included in category 3 are all the conditions which are associated with file handling, e.g. ENDFILE, RECORD and KEY.

Condition Prefix

A condition is enabled/disabled by means of condition prefixes. A condition prefix is the name of the condition, preceded by the letters NO if it is to be disabled, enclosed in parentheses and preceding the statement to which it applies and any labels which that statement has. It is separated from the statement or its labels, if any, by means of a colon.

Example:

```
(NOCONVERSION): NAME: PROC OPTIONS(MAIN);
                .
                .
                .
(NOOVERFLOW):   X = Y * Z;
                .
                .
                .
                END;
```

Here condition prefixes have been used to disable CONVERS~~ATION~~ION throughout the main procedure NAME and to disable OVERFLOW just for the duration of one assignment. Notice that there is no intervening space between NO and the condition-name.

The RANGE CHECKING conditions, as mentioned, are disabled by default and so a condition prefix is only required if they are to be enabled.

Example:

```
(SIZE): P = Q * R;
```

Note that more than one condition prefix may appear within the parenthesis.

Example:

```
(SIZE,STRINGRANGE,NOZERODIVIDE): . . .
```

Scope of Conditions

A condition prefix to a PROCEDURE or BEGIN statement applies to the complete procedure except where it has been respecified. A condition prefix to any other statement applies to that statement only.

External procedures take default disablement/enablement unless specifically overridden by a condition prefix. Internal procedures inherit disablement/enablement from their immediate containing procedure unless overridden by a condition prefix.

Example:

```
 (NOCONV): A: PROCEDURE OPTIONS(MAIN);
              CALL B;
              CALL C;
              CALL D;
                  (CONV): B: PROC;
                             .
                   (NOCONV): STMT1: I = B + C;
                             .
                            END;
                         C: PROC;
                             .
                            END;
              END;


           D: PROC;
               .
              END;
```

In the above coding the CONVERSION condition is enabled in procedure D and in procedure B, excluding the statement labelled STMT1. It is disabled elsewhere.

Note

A condition prefix to a DO group only applies to the DO statement itself, not to the complete DO group. A condition prefix to an IF statement only applies to the IF clause, not to the THEN or ELSE units. (Although these units may be given their own condition prefixes).

Example:

```
 (NOCONV): IF A = B THEN (SIZE): DO;
                                  .
                                  .
                                 END;
```

Enabled Conditions

As mentioned previously, if an exceptional condition is raised then one of three actions must be taken; NULL, Standard System Action (SSA) or programmer written.

Programmer Written Routine

The programmer controls the action to be taken on the occurrence of an interrupt by coding a statement or group of statements known as an ON-UNIT. A simple example is

```
ON ENDFILE(INFILE) SW = 'Ø'B;
```

which means that **from now on** if the end of file marker for file INFILE is read then set the bit field SW to zero. The phrase 'from now on' is important because it implies that the ON-UNIT must establish the action to be taken, before it is possible for the interrupt to occur. **Thus in most programs, ON-UNITs are coded at the beginning. The established action will exist until either:**

a) the termination of the block in which the action was set

or

b) it is overridden by another action - i.e. another ON-UNIT.

Example:

```
PROCA: PROC OPTIONS(MAIN);
         DCL PROCB ENTRY;
                             /* S.S.A. */
         ON CONVERSION CALL AERR;
                             /* AERR */
         ON CONVERSION CALL BERR;
                             /* BERR */
         CALL PROCB;
                             /* BERR */
         CALL PROCC;
                             /* BERR */
          (NOCONV): PROCC: PROC;
                             /* DISABLED */
                           END;
       END;

PROCB: PROC;
                             /* BERR */
         ON CONVERSION CALL CERR;
                             /* CERR */
       END;
```

The action 'CALL AERR' overwrites the 'SSAs' and, in turn, is overwritten by 'CALL BERR'. In PROCB, this later action is overwritten by 'CALL CERR'. However, on returning to PROCA, PROCB is terminated and the action reverts back to that established on entry to PROCB. Note that within PROCC the action is still 'CALL BERR' although there is no possibility of an interrupt causing a CONVERSION condition.

Revert Statement

The REVERT statement is used within a PROCEDURE or BEGIN block to re-establish the action which was in effect at the time of entry to that block.

Example:

```
PROCA: PROC OPTIONS(MAIN);
                      /* S.S.A.  */
         ON FOFL CALL AERR;
                      /* AERR    */
         CALL PROCB;
                      /* AERR    */
         REVERT FOFL;
                      /* S.S.A.  */
       END;

PROCB: PROC;
                      /* AERR    */
         ON FOFL CALL BERR;
                      /* BERR    */
         ON FOFL CALL CERR;
                      /* CERR    */
         REVERT FOFL;
                      /* AERR    */
       END;
```

In PROCB the REVERT statement has the same effect as the statement

```
ON FOFL CALL AERR;
```

However if PROCB is a procedure which is called from various other procedures, each having different actions established for fixed overflow, then the REVERT statement is the only way of returning to the action in force on entry to PROCB for each case. This is, in particular, appropriate in modular programming, the program being composed of several procedures some of which are likely to be called repeatedly from the other procedures.

ON Statements

The general format of the ON-STATEMENTs which a programmer may code is

ON condition [(qualifier)] [SNAP] on-unit;

Condition

The condition is the name of the exceptional condition, e.g. SIZE, ZERODIVIDE etc.

Qualifier

For all those conditions in the input/output group (e.g. ENDFILE, RECORD, KEY) the condition must be further qualified by the name of the associated file.

SNAP

Coding SNAP is optional. If present then the flow table, maintained if the compiler option FLOW has been specified (see Topic 16, 'CONTROLLING THE COMPILER') will be written onto SYSPRINT before the on-unit is executed. This is a useful debugging aid, enabling the programmer to trace the flow of the program immediately previous to the point of interrupt. An example of the output is given in Topic 20, 'TESTING AND DEBUGGING AIDS'.

On-unit

This is the action to be associated with the condition (system, programmer, null).

The on-unit must be either a single unlabelled simple statement or else, if a **group** of statements is required to be executed as an on-unit, they must be coded as a 'begin-block'.

Examples:

```
ON ERROR BEGIN;
           PUT LIST ('PROGRAM BOMBOUT');
           ERROR_CODE = '1'B;
           GOTO RECOVERY_POINT;
         END;


ON ENDFILE(INFILE) SNAP SW = 'Ø'B;

ON CONVERSION X = Ø;

ON CONVERSION CALL ERR;

ON CONVERSION;   /* NULL ACTION */
```

System on Units

These are only needed to 'reset' the action back to STANDARD SYSTEM ACTION from a programmer written action. The general format is

ON condition SYSTEM;

Dynamically Descendent On-Units

Suppose that during execution of a CONVERSION on-unit the CONVERSION condition were raised. The flow of control would then be back to the beginning of that on-unit and to execute it again, and hence produce a loop within the program. This can be prevented as follows:

```
 ON CONVERSION BEGIN;
                 ON CONVERSION SYSTEM;
                 .
                 .
               END;
```

If the CONVERSION condition were raised now within the on-unit, the resulting action would be STANDARD SYSTEM ACTION. On exit from the on-unit, the block is terminated and the resulting action, due to CONVERSION condition, reverts back to execution of the on-unit.

NULL ON-UNITS

This is almost the same as disabling a condition. With a disabled condition, processing continues as if nothing had happened. With a null on-unit however, we still get Normal Return, i.e. with

```
 ON SUBSCRIPTRANGE;
```

where the normal return is to raise the ERROR condition, the program will stop executing (see later).

Disabling a condition is more efficient than a null on-unit because the former actually prevents the occurrence of the interrupt altogether while the latter says 'let the interrupt occur, but do nothing when it does'. However, some conditions cannot be disabled and in these cases one needs to resort to null on-units.

Flow of Control

After execution of an on-unit, one problem remains: which is the next instruction to be executed? Consider the following:

```
 A: PROC;
      ON ZERODIVIDE BEGIN;
                    .
                    .
                    END;
 DIVØ: A = T/C;            /* C = Ø */
    END;
```

On execution of the instruction labelled DIV0, the ZERODIVIDE condition will be raised and the BEGIN block will be executed. To discover the flow of control after the termination of the BEGIN block (the on-unit) it is necessary to refer to section H of the Language Reference Manual again: the ZERODIVIDE condition section and the paragraph headed NORMAL RETURN. The answer is found here, in this case the flow of control passes to the point immediately following the point of interruption i.e. the next statement.

It should be noted that the NORMAL RETURN varies for different exceptional conditions and reference should always be made to the manual. The programmer may override normal return by means of a GOTO statement within the on-unit and then the latter will not have a 'normal termination'.

The Signal Statement

When a program contains ON-units, it is a good idea to test them before the interrupts start to occur in production runs. There is no need to invent ingenious routines for generating such things as CONVERSION interrupts to test CONVERSION ON-units. An easier and more economical method is provided, namely the SIGNAL statement.

The format is:

SIGNAL Condition-name;

and the effect is precisely the same as if the named interrupt had occurred 'naturally'. The NORMAL RETURN from an ON-UNIT raised by the SIGNAL statement is always the statement following the SIGNAL statement.

Obviously, once the program is working any SIGNAL statements should be removed.

Error Condition

The ERROR condition is often raised as the STANDARD SYSTEM ACTION for other conditions, e.g. look up the STANDARD SYSTEM ACTION for KEY condition in the Language Reference Manual. It is also raised for interrupts which have no ON conditions (see ONCODES later). The STANDARD SYSTEM ACTION for the ERROR condition is to halt the program. Its major use is that instead of having to write several ON statements, one for each condition that could arise, the interrupts can be 'trapped' at a later stage by an ON ERROR statement which will account for all the possible conditions, e.g.

```
 ON ERROR BEGIN;
            ON ERROR SYSTEM;
            .
            .
          END;
```

This is the most common ON-unit appearing within programs. Note the SYSTEM on-unit within the ERROR on-unit to prevent any looping.

Condition Built-in Functions

These are special purpose built-in functions which can **only be used within ON-units.**

A single type of interrupt may have a multitude of possible causes. The CONVERSION condition, for example, might be raised anywhere in the program for any number of reasons. (One common cause of the CONVERSION condition is attempting arithmetic on character fields containing non-numeric characters (e.g. '1234S'); this can be caused by mispunching). The purpose of the condition built-in functions is to determine the precise reason for a given interrupt, either for debugging or so that the program might attempt error-recovery. The various codes and their meanings are listed at the beginning of Section H of the Language Reference Manual.

If they are used it is necessary to declare them as builtin, e.g.

```
 DCL ONCHAR BUILTIN;
```

Full details of each function can be found in Section G of the Language Reference Manual.

Datafield

This function identifies the contents of the field in error when the NAME condition is raised during a GET statement (see STREAM INPUT/OUTPUT topic).

ONCHAR

This function identifies the character in error when the CONVERSION condition is raised.

ONSOURCE

This function identifies the whole field in error when the CONVERSION condition is raised.

The latter two functions can be used as pseudovariables, i.e. on the lefthand side of assignment statements. Thus values can be assigned to them in an attempt to correct the conversion error. If either are used within a CONVERSION on-unit, then the NORMAL RETURN from that on-unit is to retry the statement causing the CONVERSION condition to be raised, otherwise the NORMAL RETURN is to cancel the program. Care needs to be taken that any attempt at correction will not reproduce the CONVERSION condition, otherwise the program will loop.

Example:

```
DCL(ONSOURCE,ONCHAR) BUILTIN;
ON CONVERSION BEGIN;
                PUT LIST(ONSOURCE);
                PUT LIST(ONCHAR);
                ONCHAR = 'Ø';
              END;
```

This on-unit, using STREAM output, prints the field and character causing the error and then replaces the character by a zero.

ONCODE

This function identifies the cause of error more precisely. The function is replaced by a number which uniquely defines the cause. On the initial pages of Section H of the Language Reference Manual, the codes are listed alongside the error conditions. Note that any code greater than 1000 raises the ERROR condition.

Example:

```
DCL ONCODE BUILTIN;
ON KEY(ISFIL) BEGIN;
                SELECT(ONCODE);
                  WHEN(51) CALL NO_REC;
                  WHEN(52) CALL DUPL_KEY;
                  OTHERWISE CALL ERR_RTE;
                END;
              END;
```

ONFILE

This function identifies the file for which an I/O or CONVERSION condition was raised.

ONKEY

This function returns the value of the key of the record when KEY condition is raised.

ONLOC

This function returns the entry point to a procedure in which any condition is raised.

Summary

The PL/I ON-unit has been shown to provide a simple and convenient way of dealing with interrupts and of diagnosing programming errors. Certain conditions, such as ENDFILE and ENDPAGE are frequently used during production programs which are functioning normally, in order to handle interrupts which are not errors. Others may also be used in this way: a program might, for example, be designed to raise ZERODIVIDE condition in certain cases, causing the activation of a suitable on-unit. Also, ON conditions are used to aid debugging. Especially helpful in this respect are the condition built-in functions which will pinpoint the causes of interrupts and possibly attempt to correct the errors.

Exercises

1. Write an ON statement and the associated coding which will have the arithmetic value of a field called BIG and double that of a field called SMALL whenever a FIXED OVERFLOW occurs.

2. What does the SNAP option do?

3. A program statement is coded as follows

```
STMT: X = Y**2;
```

and the following facts should be noted:

(a) An ON OVERFLOW statement was in force when the program block containing STMT was entered. This has since been overridden. It is required to restore the original ON-unit for statement STMT and onwards.

(b) It is required to override a previously specified ON CONVERSION statement and to restore standard system action for this condition for statement STMT and onwards.

(c) It is required to enable the SIZE condition for STMT and STMT alone.

Rewrite the statement STMT together with the coding which will fulfil these requirements.

4. Write the coding which will cause a branch to the procedure labelled BIG if RECORD condition is raised, for the file called INFILE, due to the record variable being larger than the record size. If it is raised for any other reason, a branch to the procedure labelled SMALL is to take place.

5. Consider the following coding:

```
A:  PROCEDURE OPTIONS(MAIN);
     P: X = ONE;
     CALL B;
     CALL C;
          B:PROCEDURE;
            •
            Q:Y=THREE**1Ø;
            •
            END;
   END;


C:PROCEDURE;
    •
    R:Z=FOUR+FIVE;
    •
  END;
```

If SIZE is raised in Procedure A, then X is to be set to 0, and processing continued. If it is raised in Procedure B no action is to be taken. If it is raised in Procedure C system action is to be taken. Complete the above coding as necessary.

Answers

1.

```
 ON FOFL BEGIN;
           BIG = BIG/2;
           SMALL = SMALL*2;
         END;
```

2. Prints out the table maintained by the FLOW option.

3.

```
         REVERT OFL;
         ON CONVERSION SYSTEM;
 (SIZE): STMT: X = Y**2;
```

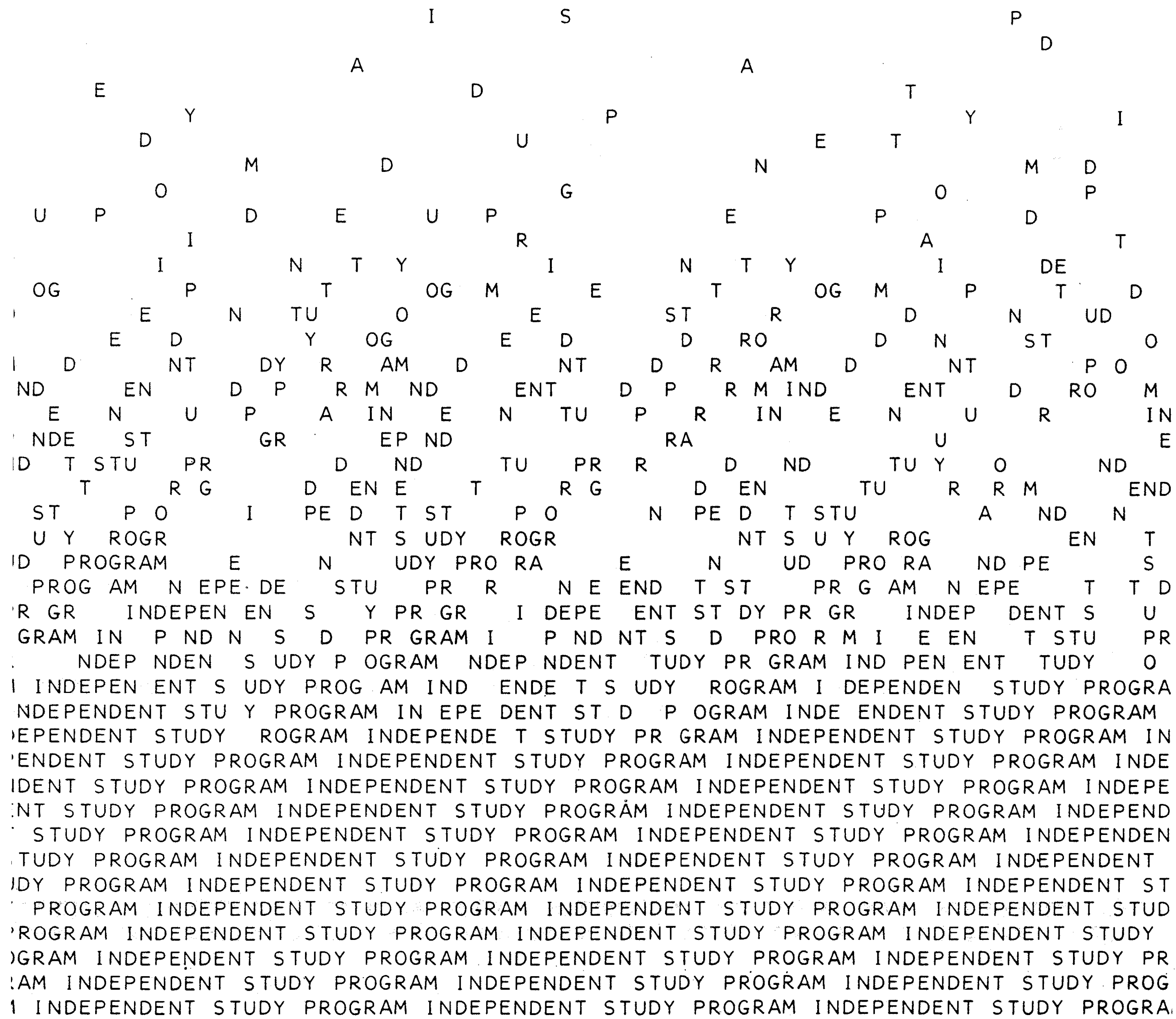
4.

```
 ON RECORD(INFILE) BEGIN;
                     IF ONCODE = 22 THEN CALL BIG;
                                    ELSE CALL SMALL;
                   END;
```

5.

```
(SIZE):A: PROCEDURE OPTIONS(MAIN);
            ON SIZE X = Ø;
            P:X=ONE;
            .
            CALL B;
            .
            CALL C;
             B: PROCEDURE;
                ON SIZE;
                .
                Q:Y=THREE**1Ø;
                .
                END;
          END;



(SIZE):C: PROCEDURE;
            ON SIZE SYSTEM;
            .
            R:Z=FOUR+FIVE;
            .
          END;
```

Topic 20



Topic 20

Testing and Debugging Aids

The purpose of this topic is to discuss the PL/I facilities which are useful for execution time debugging and testing.

Objectives

On completion of this topic you should be able to:

- Locate statements in error using object time messages
- List the on-conditions used to evaluate range values
- Use standard facilities to provide flow control tables during execution
- Describe the use of SNAP, CHECK and programmer defined conditions in program checkout.

Introduction

A program which compiles and link edits successfully can produce one of three results when executing:

1. It can terminate normally with the correct output.
2. It can terminate normally with incorrect output.
3. It can terminate abnormally.

This topic deals with the latter two results. Result 2 is normally caused by logic errors; the only clue as to what went wrong is the output. This may not be sufficient. Aids will be introduced which will allow the programmer to follow the flow of his program and to determine the value of variables at any point within the program. Result 3 can have various causes:

invalid data

uninitialized fields

unset pointers, etc.

When a program terminates abnormally, the message produced contains sufficient information for the programmer to be able to locate the point of termination and the immediate cause.

Object Time Messages

In Topic 19, 'HANDLING EXCEPTIONAL CONDITIONS', it was stated that the standard system action for the majority of ON-conditions is to raise the ERROR condition and to print a message. The general format of the message is:

IBMnnn **ONCODE=nnn 'conditionname' CONDITION RAISED message text - including error location**

For example:

```
IBM037 ONCODE=0612 CONVERSION CONDITION
RAISED CONVERSION FROM CHARACTER TO BIT IN
STATEMENT 207
AT OFFSET A8C IN PROCEDURE WITH ENTRY PROG1
```

The message text will of course vary with the condition that has been raised.

The phrase 'IN STATEMENT n' will only be printed if the compiler option GOSTMT is in effect, as mentioned in the topic named 'CONTROLLING THE COMPILER'. This phrase informs you immediately as to which statement is in error. If GOSTMT is not in effect then the offset must be looked up in the OFFSET table (produced by the OFFSET compiler option).

Example:

The following program, when compiled, produces the offset listing shown below.

```
OFFSET:  PROC OPTIONS(MAIN);
         DCL B(1ØØØØ) PIC'99999';
         DCL CARDIN FILE RECORD ENV(MEDIUM(SYSIPT,254Ø) F RECSIZE(8Ø));
                                             /* MEDIUM OPTION SINCE PROGRAM
                                                WAS RUN UNDER DOS/VS    */
         DCL 1 STR,
               2 NO PIC'9999999',
               2 REST CHAR(73);
         READ FILE(CARDIN) INTO(STR);
         K = NO;
         IF K > 1ØØØØ THEN CALL LARGE;
                       ELSE CALL SMALL;
LARGE:   PROC;
         DCL AR(K) CHAR(8Ø);
         DO L = 1 TO K;
           READ FILE(CARDIN) INTO AR(L);
         END;
         END;
SMALL:   PROC;
         DO L = 1 TO K;
           B(L) = L;
         END;
         END;
         END;
```

TABLES OF OFFSETS AND STATEMENT NUMBERS

```
                    WITHIN PROCEDURE OFFSET

OFFSET (HEX)      0      7B      B0      00      E6
STATEMENT NO.     1       5       6       7       8

                    WITHIN PROCEDURE LARGE

OFFSET (HEX)      0      C6      DE     128     144
STATEMENT NO.     9      11      12      13      14

                    WITHIN PROCEDURE SMALL

OFFSET (HEX)      0      5E      76      9E      B6
STATEMENT NO.    15      16      17      18      19
```

The offsets refer to the displacement of the numbered statement from the beginning of the appropriate procedure. To find the required statement from the offset given, you need to find the largest offset in the table less than the one in the message. The statement number below this offset is the required one. Thus in the above example if the message text is:

AT OFFSET CA IN PROCEDURE WITH ENTRY LARGE

then the required offset is C6 and hence the program terminated when executing statement number 11. Note that even if the offset in the message was DE the program still terminated at statement number 11.

A full explanation of each object time message will be found in the following manuals:

DOS PL/I TRANSIENT LIBRARY MESSAGES SC33-0035

OS PL/I OPTIMIZER COMPILER MESSAGES SC33-0027

In OS/VS these messages will be directed to the printer and so will appear with any other printed output. In DOS/VS, they will be directed to the operator's console unless the STREAM file SYSPRINT is open within the program. Obviously while testing a program, it is useful to have the messages with the printed output, so it is best to ensure that SYSPRINT is open. If the SNAP option has been used then this will be the case, otherwise the insertion of the statement

```
 PUT SKIP(Ø);
```

will open SYSPRINT. (This is a STREAM output statement which means 'go to the beginning of the current line' and so its inclusion will not affect the program logic in any way).

Thus when a program terminates abnormally you should be able to find the point of failure and the reason for it by using the execution-time message. Normally there will be no need to use further checkout facilities.

ON-CONDITIONS For Range Checking

These are the ON-CONDITIONS which are disabled by default. They should only be enabled when the program is being tested and not when it is a production program. This is because they have high overheads in terms of time and storage. They are used to check that the RANGES of various variables within the program are large enough to handle all possible valid data input. Thus when they are enabled, the program should be tested with a complete range of test data inputs and any resulting errors corrected. After this there should be no possibility of the program abending during a production run due to the reasons mentioned below. Full details of these conditions are found in section H of the Language Reference Manual.

SIZE Condition

The SIZE condition is raised when high order binary or decimal digits are lost either in an assignment to a variable or in a STREAM input/output operation.

Example:

```
         DCL SUM FIXED DECIMAL(4,2);
 (SIZE):     SUM = X + Y;
```

If the value of X is 99.9 and the value of Y is 0.1 (assume that X and Y are suitably declared) then the result of this calculation is 100.0. This would be assigned to SUM as 00.00. The SIZE condition would be raised, because the high-order digit (1) is lost.

Note that the SIZE condition would not be raised if only low-order digits were lost as the result of an assignment statement.

Example:

```
 (SIZE): SUM = 3.141593;
```

would cause the value 03.14 to be assigned to SUM with loss of the low-order fractional digits 1593. This would not raise the SIZE condition.

STRINGSIZE Condition

The STRINGSIZE condition occurs when a string is assigned to a shorter string and truncation occurs.

Example:

```
               DCL Y CHAR(6) VARYING;
               DCL X CHAR(4);
 (STRINGSIZE):     X = Y;
```

If the current length of Y is greater than four then STRINGSIZE condition will be raised.

STRINGRANGE Condition

This should be used during debugging if the program uses the SUBSTR built-in function, particularly if expressions are used in the argument list. The STRINGRANGE condition is raised when the substring defined by the SUBSTR built-in function (or pseudo-variable) does not lie within the given string (the precise rules which are used to determine whether or not a substring is valid can be found in the Language Reference Manual).

Example:

```
        DCL STRING CHAR(6);
 (STRG):    SUB = SUBSTR(STRING,N+1, 2);
```

If N had the value of 5, for instance, then an attempt is being made to obtain a substring of STRING consisting of positions 6 and 7. STRING, however, only has 6 positions. This would result in STRINGRANGE condition being raised.

SUBSCRIPTRANGE Condition

The SUBSCRIPTRANGE condition is raised when a subscript is used whose value is outside the bounds specified in the declaration of the associated array.

Example:

```
        DCL A(1Ø) FIXED(3);
        .
        .
        .
        DO I = 1 TO N;
 (SUBRG):  A(I) = A(I) + 3;
        END;
```

If N is greater than 10, then SUBSCRIPTRANGE condition would be raised on the 11th execution of the DO loop.

There is a standard system action associated with all the range checking conditions - see Language Reference Manual (Section H). If required the programmer can write an ON statement to override the standard system action with a programmer-defined on-unit. Note that the normal return from an on-unit for the SUBSCRIPTRANGE condition is to terminate the program.

FLOWTRACE

FLOW Option

This compiler option has already been discussed in Topic 16. It is specified as FLOW (n,m) and causes the compiler to maintain a table of the last n program branches and the last m blocks that were entered.

The information in this table is printed as part of the SNAP output whenever an on-unit with the SNAP option is entered. It appears as a list of the most recent transfers of control, indicating the relevant statement numbers. In addition, where a transfer of control is from one block to another block, the names of the two blocks (either procedures or on-units) are also shown.

An example of the output is given below. This is the flow table produced when the program mentioned in Topic 16, under the subheading COUNT option, is executed with the associated data.

```
'ENDFILE' CONDITION RAISED IN STATEMENT 23 AT OFFSET +000250 IN
PROCEDURE WITH ENTRY FLOW
    14        TO 12
    14        TO 12
    14        TO 12
    14        TO 12
    18        TO 23
    24        TO 9
    14        TO 12
    18        TO 23
```

The information printed here informs us that the program executed instructions sequentially until statement 14. It then returned to statement 12 and executed statements 12 to 14 four more times before continuing executing sequentially to statement 18 where the next branch occurred, and so on. By printing out this table within an ON-UNIT, the program flow immediately prior to program failure can be examined, possibly giving useful information as to the cause of failure.

PL/I Dump

The flow table can also be produced by calling a PLIDUMP. The general format of the required statement:

CALL PLIDUMP ('options list');

A full list of options is given in the PL/I PROGRAMMERS GUIDE in the chapter named 'PROGRAM CHECKOUT'. It should be noted that it is normally not necessary to obtain a full dump of the program partition.

Example:

```
 CALL PLIDUMP('TFC');
```

The option 'T' states a flow trace is required. This is similar to the output from the SNAP option. 'F' states that a list of the full file attributes and the contents of their buffers are required, and 'C' states that execution is to continue after the dump is taken.

The purpose of the flowtrace facilities is for the programmer to be able to follow the flow of control in the program in order to enable him to discover the 'route' the program took towards the error, and hence the reason for it.

CHECK Condition

Another method of tracing the flow of a program is with the CHECK condition. This is normally disabled and is enabled in the usual way i.e. with a condition prefix

```
 (CHECK): A: PROCEDURE;
```

The condition in the above example would be raised when:

1. a labelled statement is executed, and
2. an assignment is made to a variable.

The standard system action in case 1 is to print out the label and in case 2 to print out the variable and its value.

Hence a printout of every label branched to (including sequential execution) and also of each variable every time its value changes, can be obtained. This information is very useful for checking how the program has flowed and also for checking the values of variables to discover whereabouts in the program your logic has erred.

In a large program you can imagine that this would involve a great deal of printout and sorting out the useful information from the rest can be quite difficult. Provision is therefore given, whereby a subset of the labels and of the variables can be specified by enclosing the required ones in parentheses after the prefix as follows:

```
(CHECK(L1,L2,L3,V1)): A: PROC;
                       L1: V1  = 1;
                           DO J = 1 TO 3;
                              V1 = V1 + 1;
                       L2: END;
                       L3: ;
                         END;
```

The output as it appears on SYSPRINT is as follows:

```
L1;
V1= 1.00000E+00;
V1= 2.00000E+00;
L2;
V1= 3.00000E+00;
L2;
V1= 4.00000E+00;
L2;
L3;
```

The variables and labels being monitored can be varied throughout the program by disabling the CHECK for selective variables e.g.

```
(NOCHECK(L1)): BEGIN;
               .
               .
               END;
```

would switch off the monitoring of L1 for the BEGIN block. Also the state of the CHECK at entry to a procedure can be reobtained by use of the REVERT statement i.e.

```
REVERT CHECK(L2,V1);
```

The PUT Statement

The PUT statement is not specifically a debugging feature but it can conveniently be used to obtain debugging output, particularly within ON-units. It is, in fact, a STREAM output statement. The format is:

PUT DATA (data list);

The data list can be omitted. If it is, then an execution of the statement will result in the printing of every variable within the program and its value in the form of assignment statements. Every element of an array will be printed and so will all elements of a structure, fully qualified.

Example:

The output from the following program is shown below.

```
DATA:   PROC OPTIONS(MAIN);
        DCL ARR(6) CHAR(1) INIT('A','B','C',*,*,'F');
        DCL 1 STR,
              2 MINOR1 CHAR(1) INIT('G'),
              2 MINOR2,
                3 FB FIXED BIN(3) INIT(1),
                3 FD FIXED DEC(6,2) INIT(3.2);
        DCL PICT PIC'999V.9' INIT(65.4);
        PUT DATA;
        END;
```

```
PICT=065.4  STR.MINOR1='G'  STR.MINOR2.FB= 1    STR.MINOR2.FD= 3.20 ARR(1)='A'
ARR(2)='B'  ARR(3)='C'      ARR(4)=' '          ARR(5)=' '          ARR(6)='F';
```

Thus the values of variables at any point within a program can be found. In large programs this would result in large amounts of printout. Normally, you are only interested in specific variables, and you can limit the printout to the values of these variables. This is achieved by inserting the variables into a data list within the PUT DATA statement. Thus in the above program when the PUT DATA statement is replaced by

```
 PUT DATA(STR);
```

then the output will be

```
STR.MINOR1='G'          STR.MINOR2.FB=       1      STR.MINOR2.FD=        3.20;
```

A word of warning concerning the use of the PUT statement is necessary. Any attempt to PUT a fixed-decimal data item which does not contain a valid representation of a fixed decimal number will result in a data interrupt. This is particularly liable to happen when PUT DATA is used without a data list (when you print every variable the chances of encountering uninitialized or unassigned fixed-decimal data are fairly high).

Programmer-Defined Condition

The programmer is free to make up his own on-conditions and give them any name he desires. He can then, later on within the program, SIGNAL his own conditions. An example of this use is as follows:

```
ON CONDITION(MINE) SNAP BEGIN;
                          PUT DATA(V1,V2,V3);
                          PUT LIST('TEST POINT',I);
                          I = I + 1;
                        END;
DCL I FIXED DEC(2) INIT(1) EXTERNAL;
.
.
.
SIGNAL CONDITION(MINE);
.
.
.
SIGNAL CONDITION(MINE);
```

The on-condition has been called MINE, and each time it is executed, the flow table is printed, the values of V1, V2 and V3 are printed out together with a message stating which 'test point' it is. (The latter is a STREAM output statement). Then, wherever in the program this information needs to be extracted, all that needs to be coded is the SIGNAL statement as shown.

Exercises

1. Give two ways in which the point of interrupt in a program can be determined from a system action message.
2. Write down two PL/I features which are suitable for displaying the contents of data variables during program execution, for debugging purposes.
3. In the following coding, assuming that all exceptional conditions are enabled, which ones could possibly be raised.

```
DCL A(1Ø) CHAR(6);
DCL B CHAR(1Ø) VARYING;
    DO I = 1 TO N;
       /* ASSIGN VALID FIELD TO B; */
       A(I) = B;
    END;
```

4. Add code to the following 'skeleton' procedure such that, if an assignment to a binary or decimal field causes loss of high-order digits, then the value of all variables within the program will be printed on the printer; execution should resume at the statement following the assignment.

```
         TEST:PROC;
              •
              •
              •
              •
              END;
```

Answers

1. a. Use of the GOSTMT option

   b. Table of offsets and statement numbers

2. a. CHECK condition

   b. PUT statement

3. a. STRINGSIZE condition

   b. SUBSCRIPTRANGE condition

4.

```
  (SIZE):TEST:PROC;
              ON SIZE PUT DATA;
              •
              •
              •
              END;
```

Note that NORMAL RETURN resumes execution at the following statement.

Topic 21

Topic 21

Overlay Defining

This topic looks at some ways of sharing storage - i.e. referring to one area of storage by more than one name (often called overlaying). The majority of the topic will be concerned with the three types of defining using the DEFINED attribute - this gives overlaying at compilation time. Another method of sharing storage uses based storage and the ADDR built-in function - this gives overlaying at execution time.

Objectives

At the end of this topic you should be able to:

- understand the rules for simple and string overlay defining and be aware of the consequences of breaking these rules
- understand iSUB defining for arrays
- understand how based variables and the ADDR builtin function can be used to share storage
- use the appropriate storage sharing technique for a particular situation.

Introduction

There are certain situations in which it is convenient to interpret one area of storage in different ways. For instance, a program might read a file which contains two types of record - let's say one type contains the name and address of a customer and the other type contains details of purchases made by him/her. A 'code' could be included in each record to indicate the 'type' of record (e.g. an 'N' in the first byte means a name and address record, a 'P' means a purchases record). Once the record has been read into a [illegible]ge area it can be 'interpreted' using an appropriate structure, which corresponds to the storage area itself or is 'overlayed' on that area.

The three types of defining using the DEFINED attribute are **simple, string overlay** and **iSUB.** Based variables and the ADDR builtin function can be used to give overlaying of storage areas at execution time - this is useful in situations where the use of the DEFINED attribute would break the rules of the language (see later).

The Defined Attribute

The DEFINED attribute is used to specify that the variable being declared refers to all or part of some other area of storage. This takes place at compilation time.

Example:

```
 DCL NUMBER DEC FIXED(3) INIT(Ø);
 DCL QUANTITY DEC FIXED(3) DEF NUMBER;
```

Note that DEF is the standard abbreviation of DEFINED.
Here QUANTITY is the 'defined variable' and NUMBER is called the 'base variable' (which is nothing to do with based storage). The two declarations need not be next to one another. The base variable must be known in the block in which the defined variable is declared. If we refer to QUANTITY we are referring to the same area of storage as when we refer to NUMBER. QUANTITY is, therefore, effectively initialized by the initialization of NUMBER. It is not possible to initialize a defined variable directly (think about it).

Defined variables do not 'inherit' the data type or precision of the base variable; they must be declared with suitable attributes and precision of their own. The suitability of attributes depends on the type of defining being used - simple or string overlay (see later).

The preceding example was used to illustrate several points about the DEFINED attribute. The following is a more realistic use of the DEFINED attribute.

```
 DCL LETTERS CHAR(3);
 DCL NUMBERS PIC'999' DEF LETTERS;
```

The area of storage, in this example, can be referred to by the name LETTERS when it contains character data. Alternatively the same area can be referred to by the name NUMBERS when we wish to perform arithmetic on the contents of the field - in which case the contents must be numbers or else an execution error will result. (Arithmetic can be performed on a numeric character field but this is less efficient than using a numeric picture field).

The following is a summary of the more important points about the DEFINED attribute. **Do not try to learn these points.** Merely try to understand the reasons behind them. More information is contained in Section I of the Language Reference Manual under the DEFINED heading.

1. Base variables are declared in the ordinary way, which may be explicitly, implicitly or contextually.
2. Base variables cannot themselves be defined on another variable, nor may they be BASED on a pointer.
3. Defined variables **cannot** have the EXTERNAL attribute, although base variables may be INTERNAL or EXTERNAL.
4. Defined variables have the storage class of their base variables; they cannot be **declared** with their own storage class (i.e. they cannot be given the AUTOMATIC, BASED, CONTROLLED or STATIC attribute).
5. Only the base variable may be initialized.
6. Any number of variables may be defined on a single base variable.
7. The base variable is in no way changed or affected merely by having another variable defined on it, but any **assignments** to a defined variable will change the contents of the base variable.

Simple Defining

The basic rule about simple defining is that the defined variable must 'match' the base variable (see below).

Elements

Two **element** variables are said to match when they are of the same data type and precision. However a defined string (character or bit) can be equal to or less than the size of the base string.

Example:

```
DCL CHAR6 CHAR(6);
DCL CHAR4 CHAR(4) DEF CHAR6;
```

Arrays

The simple defining of **arrays** involves a one-to-one correspondence between elements **with the same subscripts** in the base array and defined array. These elements must match in the sense described above. The bounds of the base array and defined array can be different, however the defined array should be completely contained in the base array.

Examples:

```
DCL ARR(3)      CHAR(2) INIT('OB','LA','DI');
DCL ARRAY1(3)   CHAR(1) DEF ARR;
DCL ARRAY2(2:3) CHAR(2) DEF ARR;
DCL ELEM        CHAR(1) DEF ARR(1);
```

In the above example ARRAY1 only refers to the first character in each element of the base array ARR. Hency ARRAY1 has been effectively initialized so that ARRAY(1) is '0', ARRAY1(2) is 'L' and ARRAY(3) is 'D'; ARRAY2 has been defined on top of the second and third elements of ARR and so has effectively been initialized so that ARRAY2(2) is 'LA' and ARRAY2(3) is 'DI'. ELEM is defined on top of a particular element so ELEM is effectively initialized to 'O'. Notice that ARRAY 1 consists of 'non-connected' areas of storage.

Structures

The simple defining of **structures** involves a correspondence between elements in the same 'logical position' within the structures. Structures are said to match when their logical organization is identical **and** corresponding elements match. In addition, the defined structure must be a major (level-one) structure. However the base structure can be a minor structure.

Examples:

```
  DCL  1  INRECORD1,
          2  CODE    CHAR(1),
          2  NAME,
               3  FIRST  CHAR(2Ø)  INIT('JULIE'),
               3  SECOND CHAR(2Ø   INIT('APSON'),
          2  NUMBER  DEC  FIXED  (3);

  DCL  1  INRECORD2  DEF  INRECORD1,
          2  TYPE    CHAR(1),
          2  ADDRESS,
               3  FIRST    CHAR(2Ø),
               3  SECOND   CHAR(2Ø),
          2  QUANTITY  DEC  FIXED  (3);

  DCL  1  INITIALS  DEF  NAME,
          2  FIRST      CHAR(1),
          2  SECOND     CHAR(1);
```

The above examples illustrate most of the points. In particular INITIALS. FIRST is effectively initialized to 'J' and INITIALS effectively initialized to 'A'. INITIALS consists of 'non-contiguous' areas of storage.

Whenever the base variable does not match the defined variable (different structuring or different element attributes for 'corresponding' structure elements) then simple defining is impossible and string overlay defining (see next section) is attempted. The compiler normally issues a message to indicate this. However, if the defined and base structure consist entirely of character string elements or entirely of bit string elements and the sizes of the two structures are the same, then string overlay defining will take place automatically and the compiler will give no message even if the structuring is different.

String overlay defining is also used if the POSITION attribute is specified (see next section).

String Overlay Defining

In the case of a simply defined data aggregate, it is defined element by element upon the base aggregate. We have seen that, when the elements of the defined aggregate are strings which are shorter than those of the base aggregate, the defined aggregate will contain 'gaps' in main storage: it will be 'non-contiguous'.

The distinguishing feature of string overlay defining is that aggregates are not defined element by element, but each aggregate is treated as a continuous string. It follows that the defined aggregate always occupies a **contiguous** portion of the storage occupied by the base aggregate. In string overlay defining, neither the base variable nor the defined variable may be non-contiguous. Only string variables may be defined in this way. Base and defined variables may be either elements or aggregates, but they must be both character string data, or both bit string data. (Here picture data is included as character string data).

The chief application of string overlay defining is the defining of more than one structure upon a single input or output area, so that records of different types may be accommodated. Remember that string overlay defining automatically occurs whenever the defined and base items do not 'match', and also whenever the defined variable has the POSITION attribute.

Example:

```
      DCL  1 REC_1,
             2 RECTYPE  CHAR(1),
             2 NAME     CHAR(2Ø),
             2 ADDRESS  CHAR(3Ø);

      DCL  1 REC_2 DEF REC_1,
             2 TYPE CHAR(1),
             2 PURCHASE_ITEMS(1Ø) CHAR(5);

      READ FILE(INFILE) INTO(REC_1);
      IF RECTYPE='N' THEN DO;
                           .
                         END;
      IF RECTYPE='P' THEN DO;
                           .
                         END;
```

Here the file INFILE contains both name/address and purchase item records and structures have been declared to correspond with these two record types. The two structures consist entirely of character string elements and the sizes of the same, therefore the compiler will produce string overlay defining automatically.

The POSITION Attribute

There are occasions when the defined variable 'matches' the base variable, and yet simple defining is **not** required. In this case we may use the POSITION attribute to produce string overlay defining. (The POSITION attribute may be abbreviated to POS).

Example:

```
   DCL 1 INREC,
         2 WORD CHAR(1Ø),
         2 LINE CHAR(3Ø);

   DCL 1 TWINREC DEF INREC POS(1),
         2 HALF CHAR(5),
         2 REST CHAR(5);
```

In these declarations, TWINREC is string overlay defined upon INREC. Thus HALF refers to the storage occupied by the first five characters of WORD, and REST refers to that occupied by the second five characters of WORD. If string overlay defining had not been forced by POS(1), however, **simple defining** would have been in effect (because the structures match): HALF would then have referred to the same storage as before, but REST would have referred to the storage occupied by the first five characters of LINE.

As you may have guessed, the POSITION attribute is not always specified as POS(1). The parentheses may contain any expression, which is evaluated as an integer. This specifies the bit or character of the base variable where the defined variable is to start.

Example:

```
 DCL ARR(3) CHAR(3) INIT('BLA','CKP','OOL');
 DCL MID(3) CHAR(1) DEF ARR POS(4);
```

Here, although MID matches ARR, the defining is not **simple. String overlay** defining is used because of the POS attribute and the three elements of MID are effectively initialized to 'C', 'K' and 'P'.

Non-Matching Base and Defined Variables

Although it is possible to 'define' structures, arrays or elements on areas of storage whose attributes are different, this is not to be recommended. The effect is to 'force' string overlay defining - the rules of this say that both the defined and the base variable must be bit or both must be character. In many cases (e.g. fixed decimal field overlaying a character field) the rules for string overlay defining will be broken. The compiler merely issues a message and it is possible for the program to execute. The programmer should be aware that he is breaking the 'rules', however, because future releases of the compiler might not be so lenient. One way of 'legally' string overlaying completely different structures on the same storage area is to use the based storage class and the ADDR builtin function.

Based Variables

Example:

```
  DCL 1 STRUCA,
        2 CODE    CHAR(1),
        2 NAME    CHAR(24),
        2 ADDRESS CHAR(2Ø);

  DCL 1 STRUCB BASED(P),
        2 ITEM CHAR(15),
        2 DESCRIPTION CHAR(3Ø);

  P = ADDR(STRUCA);
```

The above coding associates the pointer P with the address of STRUCA at **execution time** by means of the assignment statement (look up the ADDR builtin function if you have forgotten). This has the same effect as the following coding:

```
  DCL 1 STRUCB DEF STRUCA POS(1),
        2 ITEM     CHAR(15),
        2 DESCRIPTION CHAR(3Ø);
```

However, string overlay defining takes place at **compilation time.**

The use of based variables differs from string overlay defining in another important respect. Data of any type or organization may be given the same main storage address as data of any other type or organization: the programmer will only become aware of absurdities when they generate execution time errors. This sharing of storage between completely different data types enables storage to be used more economically.

Example:

```
DCL 1 DESCRIPTION BASED(P),
      2 TYPE CHAR(1),
      2 DESC CHAR(39);
DCL 1 DETAIL,
      2 LETTER CHAR(1),
      2 CODE_NO CHAR(5),
      2 QTY          FIXED DEC(5),
      2 REORD_QTY    FIXED DEC(5),
      2 WAREHOUSE CHAR(28);

P = ADDR(DETAIL);
READ FILE(INFILE) INTO (DETAIL);
IF TYPE ='A' THEN DO;
                  /* PROCESS DETAIL RECORD */
                   END;
IF TYPE ='B' THEN DO;
                  /* PROCESS DESCRIPTION RECORD */
                   END;
```

The above coding could be used to handle a file which contains two completely different record types corresponding to DESCRIPTION and DETAIL. (Note that even more storage could be saved by declaring DETAIL and DESCRIPTION based on pointer P. Then locate mode statements (READ....SET) could be used to process the records in the buffer.

iSUB Defining

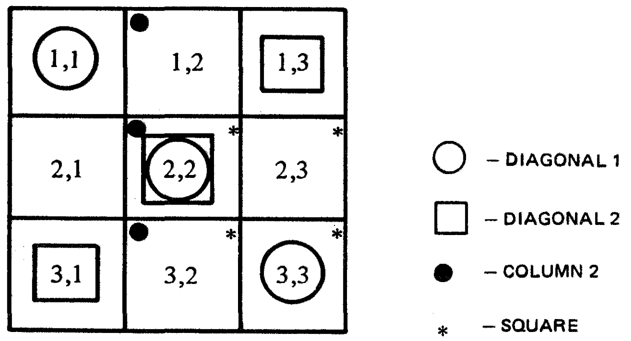
This is similar to the **simple defining of arrays,** except that it is used when the programmer does not wish the subscripts of each element of the defined array to be the same as the subscripts of the corresponding element in the base array; instead he is able to specify an algebraic relationship between the subscripts of corresponding elements.

A list of dummy variables called an iSUB list is used to define this relationship. There are as many items in this list as there are dimensions in the base array; they specify the algebraic operations to be performed on the subscripts of a defined element to obtain the subscripts of the corresponding base element. The items in the iSUB list can be constants or expressions involving iSUB where i refers to a dimension of the defined array (see examples).

```
DCL BASE(3,3) DEC FIXED(5);
DCL DIAGONAL1(3) DEC FIXED(5) DEF BASE(1SUB,1SUB);
DCL DIAGONAL2(3) DEC FIXED(5) DEF BASE(1SUB,4-1SUB);
DCL COLUMN2(3)   DEC FIXED(5) DEF BASE(1SUB,2);
DCL SQUARE(2,2)  DEC FIXED(5) DEF BASE(1SUB+1,2SUB+1);
```

DIAGONAL1 is a one dimensional, 3 element array defined on BASE such that

DIAGONAL1(1) corresponds to BASE(1,1)
DIAGONAL1(2) corresponds to BASE(2,2)
DIAGONAL1(3) corresponds to BASE(3,3)



DIAGONAL1 thus corresponds to the 'forward' diagonal and DIAGONAL2 corresponds to the 'reverse' diagonal. COLUMN2 corresponds to the second column of BASE. SQUARE corresponds to the four elements shown in the diagram as follows:

```
SQUARE(1,1) corresponds to BASE(1+1,1+1) i.e. BASE(2,2)
SQUARE(1,2) corresponds to BASE(1+1,2+1) i.e. BASE(2,3)
SQUARE(2,1) corresponds to BASE(2+1,1+1) i.e. BASE(3,2)
SQUARE(2,2) corresponds to BASE(2+1,2+1) i.e. BASE(3,3)
```

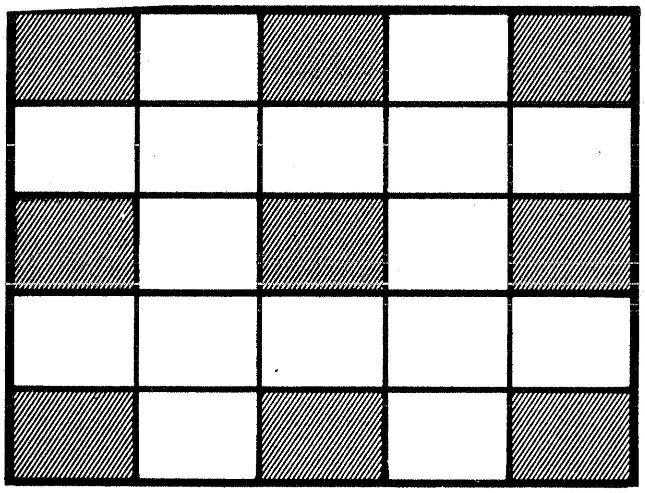
You should now be reasonably familiar with **how** to use iSUB defining but why is it used? Well, once an array has been iSUB-defined on selected elements of the base array, then operations can easily be performed on those selected portions using the defined array within a DO loop. For instance, an insurance company might use an array of insurance premiums for men and women of different ages, weights and occupations. An iSUB defined array could be used to alter the premiums for a particular category of person (e.g. male computer programmers over 27 years old and less than 180 lbs. in weight).

Exercises

1. Twenty bytes of storage are to be treated as a twenty character field called LONG, or as an array of four FIXED DECIMAL(9) fields, called ARRAY. Write the necessary declarations and any other statements required.
2. Declare a four element array, called FOUREL, such that each element has the CHAR(2) attribute, and which occupies bytes 7-14 inclusive of the storage occupied by the array LINEAR, which has been declared as:

```
       DCL LINEAR(5) CHAR(1Ø);
```

3. Declare a 20 byte character field whose **sole** purpose is to contain data which can be accessed two characters at a time. You might call the field TRICK (hint!).
4. Declare a '3-by-3' array, ARR2, whose elements correspond to the shaded elements of ARR1 as shown in the diagram.



ARR1 is declared as:

```
       DCL ARR1(5,5) CHAR(1);
```

Answers

1.

```
DCL LONG   CHAR(2Ø);
DCL ARRAY(4) FIXED DEC(9) BASED(P);
P=ADDR(LONG);
```

Based variables and the ADDR builtin function are used because ARRAY has different attributes to LONG. It is possible, at present, to use string overlay defining here but it is breaking the rules (see section on non-matching base and defined variables).

2.

```
DCL  FOUREL(4) CHAR(2) DEF LINEAR POS(7);
```

3.

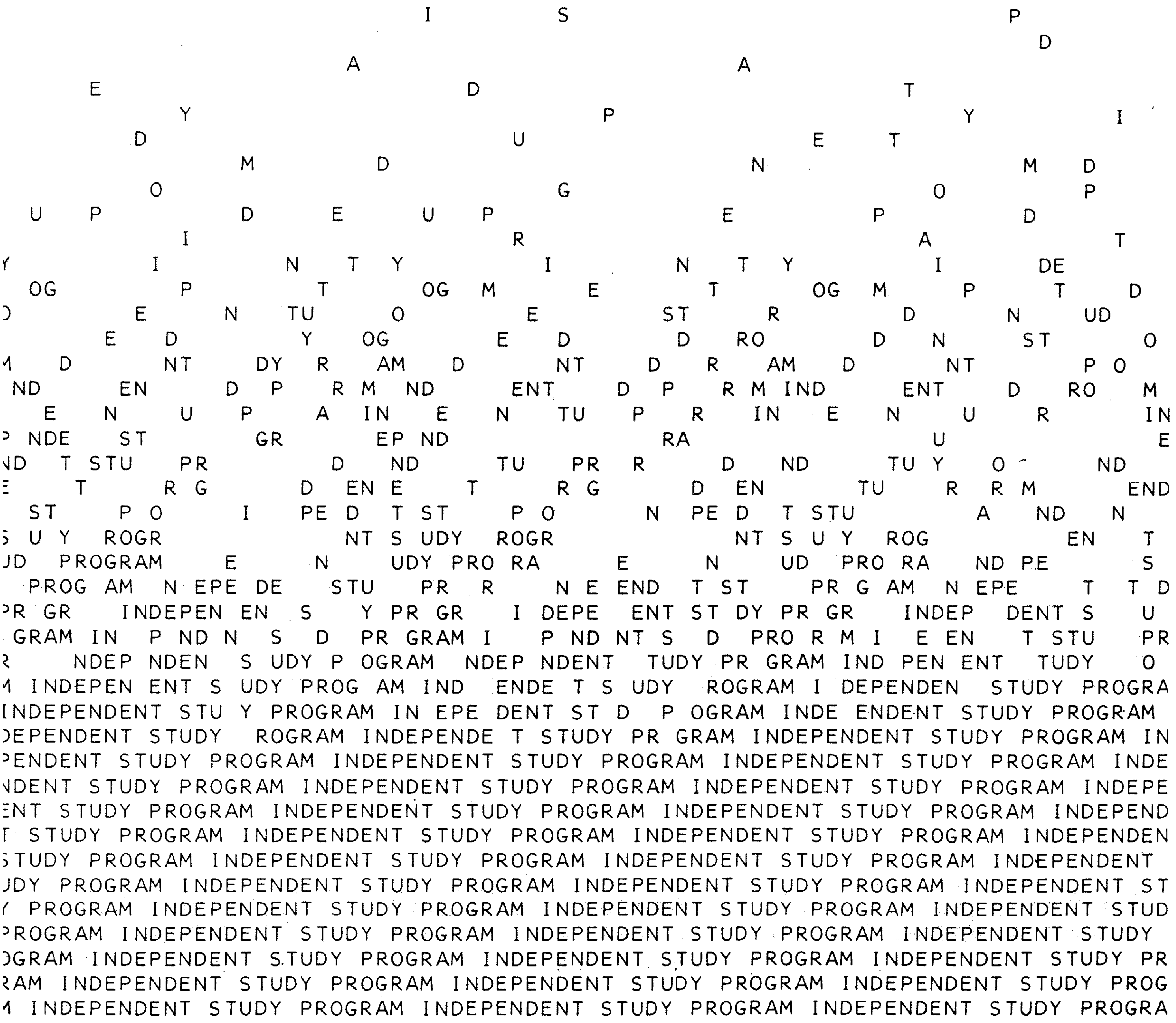
```
DCL TRICK(1Ø) CHAR(2);
```

No DEF attribute is required! Arrays are intended to be used exactly as described in the question.

4.

```
DCL ARR2(3,3) DEF ARR1(2*1SUB-1,2*2SUB-1);
```

Topic **22**



Topic 22

Structures and Arrays

This topic deals with various uses of structures and arrays which have not been covered so far. The first part of the topic is concerned with structures, the second part with arrays.

Objectives

At the end of this topic you should be able to:

- use the LIKE attribute for declaring structures, where applicable
- use BY NAME assignment for structures, where applicable
- understand and be able to use cross section of an array
- use array handling built-in functions
- understand the CONNECTED attribute and when to use it.

Introduction

PL/I uses **elements** of data which can have attributes of decimal fixed, binary float, character etc. PL/I also allows the programmer to refer to **aggregates** of these data elements. **Structures** are aggregates whose elements may have different attributes. **Arrays** are aggregates whose elements have the same attributes.

Structures

LIKE Attribute

The LIKE attribute is used to declare a major or minor structure with the same structuring as that of another major or minor structure. **Minor structure names, elementary names and attributes are copied exactly.**

Example:

```
   DCL 1 MASTREC,
         2 CODE CHAR(1),
         2 NAME,
           3 FIRST CHAR(10),
           3 SECOND CHAR(10),
         2 ADDRESS CHAR(40);

   DCL 1 UPDATE LIKE MASTREC;
/*    EFFECTIVELY THE SAME AS     */
/* DCL 1 UPDATE,                  */
/*       2 CODE CHAR(1),          */
/*       2 NAME,                  */
/*         3 FIRST CHAR(10),      */
/*         3 SECOND CHAR(10),     */
/*       2 ADDRESS CHAR(40);      */

   READ FILE(MASTFIL) INTO (MASTREC);
   READ FILE(UPFIL) INTO (UPDATE);

   MASTREC.ADDRESS = UPDATE.ADDRESS;
```

You can see from the example that the LIKE attribute saves coding at declaration time. (Remember, however, that references to elements within the two structures would have to be qualified by the major structure name, e.g. MASTREC.ADDRESS, in order to avoid ambiguity). See section I in the Language Reference Manual for further information.

BY NAME Assignment

The BY NAME assignment is a special case of the assignment of one structure to another structure. A BY NAME assignment only takes place for those elements whose element names are the same in the structures concerned and whose qualifying names, except the highest-level qualifier appearing in the assignment statement, are also the same.

In the example below the programmer has arranged that the relevant parts of the printline structure have the same names (and qualifying names) as the input record structure. Having taken this small amount of care, the necessary assignments can then be carried out in one statement.

Example:

```
DCL 1 MASTREC,
      2 CODE CHAR(1),
      2 NAME,
        3 FIRST CHAR(1Ø),
        3 SECOND CHAR(1Ø),
      2 ADDRESS CHAR(4Ø);

DCL 1 PRINTLINE,
      2 ASA       CHAR(1)  INIT(' '),
      2 BLANKS1   CHAR(1Ø) INIT(' '),
      2 NAME,
        3 FIRST   CHAR(1Ø),
        3 SECOND  CHAR(1Ø),
      2 BLANKS2   CHAR(2Ø)  INIT(' '),
      2 ADDRESS   CHAR(4Ø);

READ FILE(MASTFIL) INTO (MASTREC);
PRINTLINE = MASTREC, BY NAME;
WRITE FILE(PRTFIL) FROM (PRINTLINE);
```

ARRAYS

Introduction

You should already be familiar with the terms, dimensions, bounds, extents and elements as applied to arrays.

Cross-Sections of Arrays

It is possible to specify a particular row or column of an array using asterisk notation.

Example:

```
      DCL ARRAY(3,4)  CHAR(1) INIT((12)(1)' ');

      ARRAY(*,3)='C'; /* THIRD COLUMN */
      ARRAY(3,*)='R'; /* THIRD ROW    */
```

ARRAY (*,3) refers to all the elements in the third column of ARRAY; ARRAY (3,*) refers to all the elements of the third row. ARRAY will be stored row by row (right subscript varying faster). So ARRAY (*,3) will be in **contiguous storage;** ARRAY (3,*) will be in **non-contiguous storage.** After the above assignments ARRAY will look like:

		C	
		C	
R	R	R	R

Incidentally, if you do not understand the initialization expression ((12)(1)' ') then re-read the appropriate section in Topic 7.

Array Handling Built-in Functions

By now, you will be familiar with the use of PL/I built-in functions. In this section we will be looking at some of the more useful built-in functions concerned with arrays.

The following example illustrates the use of the DIM, LBOUND and HBOUND built-in functions. Look them up in section G of the Language Reference Manual before reading the example. The subroutine SUB has been written to process any three-dimensional array with BINARY FIXED (15) attributes, regardless of the bounds in each dimension. This is indicated by declaring the parameter ARR with an asterisk for each of the three dimensions. Within SUB the extent of the associated array can be referred to by the DIM built-in function, and the upper and lower bounds in a particular dimension can be referred to by the HBOUND and LBOUND respectively.

```
DCL A(5:1Ø,8,-3:7)  BIN FIXED(15);
CALL SUB(A);
.
.
SUB:PROC(ARR);
    DCL ARR(*,*,*)  BIN  FIXED(15);
    I=LBOUND(ARR,1);
    K=LBOUND(ARR,3);
    IF DIM(ARR,2)=8
    THEN DO J = LBOUND(ARR,2) TO HBOUND(ARR,2);
                  ARRAY(I,J,K)=ARRAY(I,J,K)*1Ø;
           END;

END;
```

For the example above, the assignment statement in the DO loop would result in:

```
  ARRAY(5,J,-3)=ARRAY(5,J,-3)*1Ø;
```

The SUM and PROD built-in functions can be used to return the sum or product of all the elements of an array or of a particular crosssection.

Example:

```
 DCL ARIA(2,2)  BIN FIXED(15)  INIT(1,2,3,4);

 I = SUM (ARIA);        /* I=1Ø  */
 J = PROD(ARIA(*,1));   /* J=3   */
```

CONNECTED attribute

The CONNECTED attribute applies to array parameters.

When an array is passed as an argument, the compiler cannot assume that the elements of this array will be contiguous in storage. This is because the array could be a cross-section of another array, or it could be simple-defined on another array. Unless the compiler is explicitly told otherwise, it will cater for the possibility that an array parameter refers to non-contiguous storage. This involves the use of less efficient code than if the array parameter referred to contiguous storage. Therefore if you know that the array parameter refers to contiguous storage, declare the parameter as CONNECTED for efficiency.

Furthermore, it is **necessary** to declare the array parameter as CONNECTED if you wish to use it as:

a) a target or source field in a record I/O statement

b) a base variable for **string** overlay defining (but not **simple** defining).

It is possible to pass a non-contiguous array argument (e.g. a column cross-section) to an array parameter declared as CONNECTED. If the CONNECTED parameter is declared in an internal procedure then the compiler can detect that the non-contiguous argument does not match the CONNECTED parameter and so the compiler will set up a 'dummy' argument in contiguous storage. If the CONNECTED parameter is declared in an external procedure then you **must** indicate this in a parameter descriptor list within the calling procedure, so that the compiler can set up a 'dummy' argument in contiguous storage within the calling procedure. In either case remember that if you alter the CONNECTED parameter, this is actually altering the dummy argument set up in the calling procedure **but not the original non-contiguous array argument.**

Example:

```
   CORFU: PROC;
         DCL CRETE ENTRY((2) CHAR(1) CONNECTED);
         DCL ARRAY(2,4) CHAR(1);
         .
         .
         CALL CRETE(ARRAY(*,2));
         .
   END;
```

```
   CRETE: PROC(ISLAND);
         DCL ISLAND(2) CONNECTED CHAR(1);
         .
         WRITE FILE(OUT) FROM (ISLAND);
         .
   END;
```

In the preceding example a dummy argument will be created for ARRAY (*,2) in contiguous storage within procedure CORFU and the address of this dummy argument will be associated with ISLAND in procedure CRETE.

Note that ISLAND **must** be declared CONNECTED within CRETE because it is used as a source variable in the WRITE statement.

Exercises

1.

```
DCL ROSSETTI(2,4)  CHAR(1);
DCL MANZANERA(2,2) CHAR(1) DEF ROSSETTI(1SUB,2SUB+1);

RESULT = SUM(MANZANERA);
```

What is wrong with the above coding? (Hint: Section G of the Language Reference Manual might be useful here).

2.

```
 MAN:PROC;
     DCL ARRAY(3,3) CHAR(1) INIT('A','B','C','D','E',
                                'F','G','H','I' );
     CALL TAXI(ARRAY(*,3));

 END;



TAXI:PROC(ARR);
     DCL ARR(3) CHAR(1) CONNECTED;
     .
     .
END;
```

Write the statement required in procedure MAN which will enable ARR (in TAXI) to refer to the required non-contiguous cross-section of ARRAY (in MAN).

Are there any restrictions on the way in which ARR can be used within procedure TAXI?

Answers

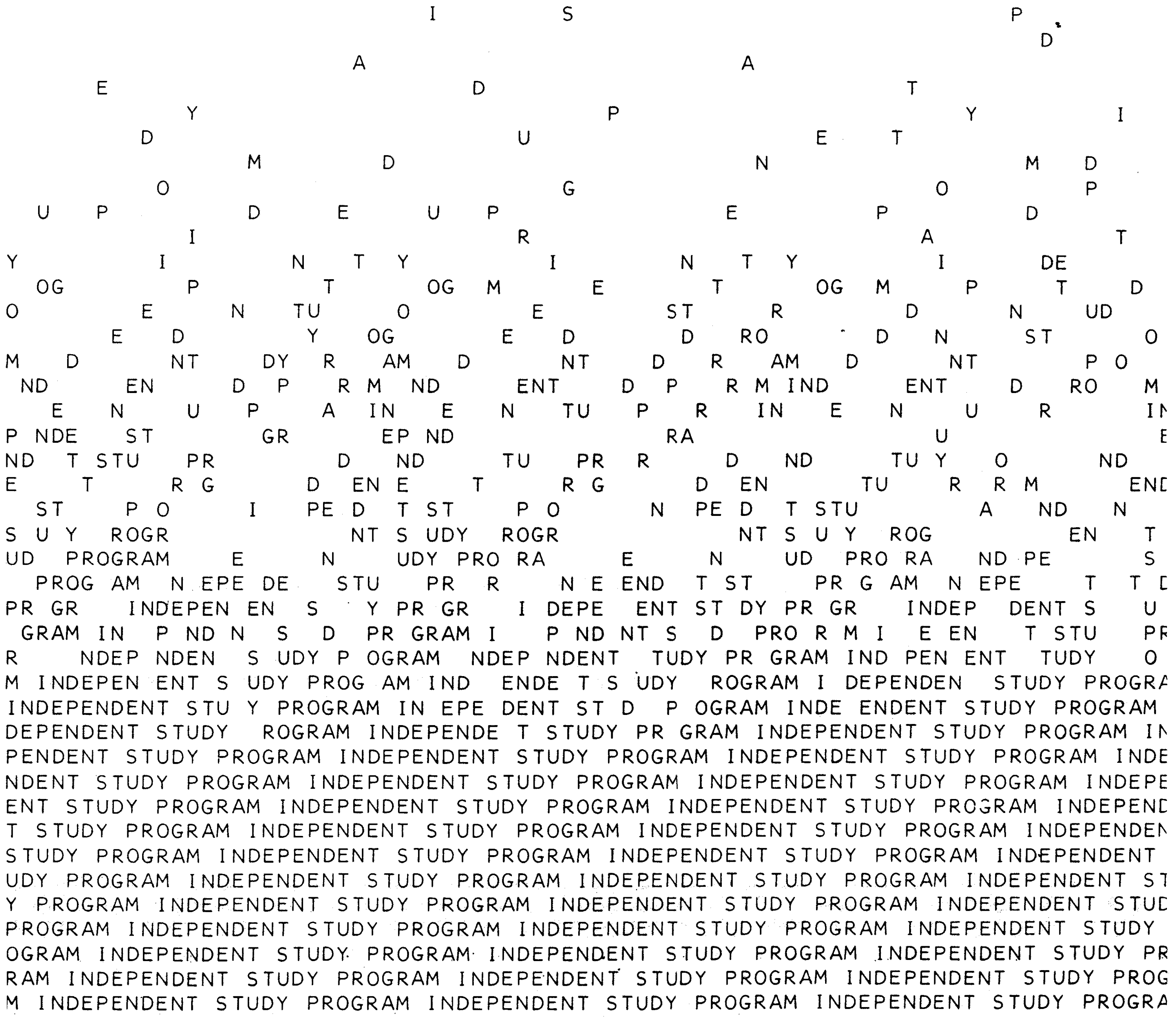
1. The argument of the SUM built-in function cannot be an iSUB-defined array.

2.

```
  DCL TAXI ENTRY((3) CHAR(1) CONNECTED);
```

Reading or assigning data into ARR will affect the dummy argument within MAN but not the original argument ARRAY (*,3).

Topic 23



Topic 23

Data Storage Allocation

This topic describes the way in which various forms of data aggregates and elements are arranged in storage and considers the problems which can occur with the mapping of structures.

Objectives

By the end of the topic the student should be able to:

- state the number of storage bytes required to hold variables of each type
- describe the internal mapping of structure aggregates
- state the effect of the ALIGNED/UNALIGNED attribute
- describe the impact of alignment on I/O operations
- state the function of the COBOL option of the ENVIRONMENT attribute.

System/370 Storage Alignment Requirements

One of the functions of the compiler is to decide how the data items used by a program should be arranged or 'mapped' in main storage. It does this for both element variables and data aggregates.

As far as element variables are concerned it is not essential to know the rules but in the case of structures it is most important that the programmer is familiar with those aspects of the mapping process which affect record I/O. We shall be discussing this in detail in this topic but first let's take a look at the simple rules which govern the mapping of **element** variables.

System/370 machines are capable of manipulating all forms of data, regardless of where that data is located in our partition. However, certain types of data can be dealt with more efficiently if they start at a particular 'sort' of address.

For instance a field declared in our PL/I program as

BINARY FIXED (15)

could be dealt with more efficiently if it began at an address in the machine's storage, divisible by 2.

When the system is generated it is possible to indicate that we wish these alignment requirements to be maintained.

The PL/I compiler **assumes** that this is the case, and therefore, to our declaration of BINARY FIXED (15), it adds the default attribute ALIGNED. When allocating storage to this variable, PL/I ensures that it starts at an address divisible by 2 - i.e. a HALFWORD boundary. Let's look at the other boundaries that exist.

Divisible by—	Boundary type
1	Byte
2	Halfword
4	Fullword
8	Doubleword

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	etc.
B	B	B	B	B	B	B	B	B	B	B	B	B	B	B	B	B	
H		H		H		H		H		H		H		H		H	
F				F				F				F				F	
D								D								D	

PL/I Element Variables

Bit

These fields require CEIL (x/8) bytes - where CEIL is the built-in function which returns the smallest integer greater than or equal to the expression in parentheses and x is the number of bits in the field. Bit fields are aligned on byte boundaries.

Character

These fields require one byte for every character and can start on any byte boundary.

Picture

These fields require one byte for every picture character other than V, and can start on any byte boundary.

Decimal Fixed

These require one half byte for a sign, plus a half-byte for each digit, rounded up to a complete number of bytes, e.g. DECIMAL FIXED (5) and DECIMAL FIXED (4) both require three bytes. The number of digits beyond the decimal point has no effect on the field size requirements. Decimal Fixed fields can start on any byte boundary.

Binary Fixed/Float and Decimal Float

The following fields require the following alignments and have lengths as indicated.

BINARY FIXED (1-15)	H	2 bytes
BINARY FIXED (16-31)	F	4 bytes
BINARY FLOAT (1-21)	F	4 bytes
BINARY FLOAT (22-53)	D	8 bytes
DECIMAL FLOAT (1-6)	F	4 bytes
DECIMAL FLOAT (7-16)	D	8 bytes

Example:

```
DCL A CHAR(14);                        /* 14 BYTES */
DCL B PIC'$$$,$$$,$$9V,99';            /* 14 BYTES */
DCL C DECIMAL FIXED(1Ø,6);             /*  6 BYTES */
DCL D FLOAT;                           /*  4 BYTES */
DCL E BINARY FIXED(2Ø);                /*  4 BYTES */
DCL F BIT(16);                         /*  2 BYTES */
DCL G BIT(17);                         /*  3 BYTES */
```

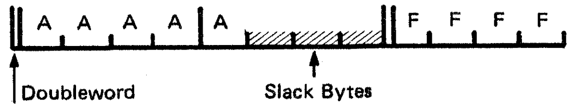
Storage Mapping of Element Variables

PL/I will shuffle individual fields around in order to make the most efficient use of storage. If the compiler allocated storage for data items in the order in which they were declared, it would have to insert 'padding' or 'slack' bytes in certain cases.

Example:

```
DCL A CHAR(5),
    F FIXED BIN(16);
```

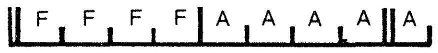
Assuming that the next available byte started on a doubleword boundary, the compiler would have to allocate storage for A and F as follows:



The compiler seeks to avoid or at least minimize the wasteful occurrence of slack bytes by mapping element variables in the following order:

1. Items requiring a doubleword boundary
2. Items requiring a fullword boundary
3. Items requiring a halfword boundary
4. Items requiring a byte boundary.

Thus in our example, the compiler would map F and A as follows:



Structure Mapping

The above mentioned 'shuffling' around of storage cannot take place with structures, where the relationship of one field to another is important. Otherwise it would not be possible to transmit data to and from structures using RECORD I/O nor would it be possible to DEFINE other variables over a structure.

The mapping of a structure in PL/I only needs careful consideration when the structure contains fields requiring half, full and/or double word alignment, i.e. BINARY FIXED/FLOAT and DECIMAL FLOAT fields.

This is of more interest to the scientific rather than the commercial programmer, the latter being more concerned with structure elements having byte alignment (character, picture and fixed decimal fields).

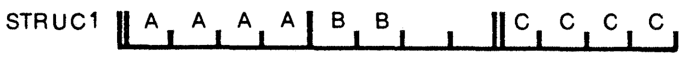
We are now going to consider the rules for structure mapping. Before we do, it should be noted that structure mapping does not involve actual **physical movement** of data items in storage. In fact it is the process by which the relative location of each element in the structure is **calculated.** This process is completed by compiler-generated code **before** storage is allocated for the structure. Thus, when the structure is allocated, the addressability of the elements within it is already established.

It is not our intention to give a **formal** list of the structure mapping rules in this section. What we shall do is to illustrate the process by describing the mapping of two different structures. The full rules are located in the Language Reference Manual - Section K, 'DATA MAPPING'.

Example 1:

```
 DCL 1 STRUC1,
       2 A,
       2 B CHAR(2),
       2 C BINARY FIXED(31);
```

Structures are mapped by the PL/I compiler, relative to a double word and thus STRUC1 would be mapped as follows:



The field C requires alignment on a full word boundary and hence cannot follow immediately on from the field B.

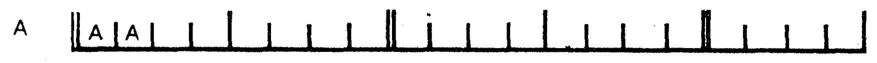
Thus records written from this structure would be 12 bytes long, incorporating two embedded blanks, and not, as might have been expected, 10 bytes long.

Example 2:

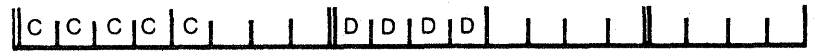
```
DCL 1 STRUC2,
        2 A PIC'99',
        2 B,
          3 C CHAR(5),
          3 D FLOAT(4),
        2 E,
          3 F CHAR(3),
          3 G BINARY FIXED(2Ø);
```

Mapping occurs from the deepest logical level outwards, i.e. A, B and E are mapped as three separate units, and then joined together top to bottom.

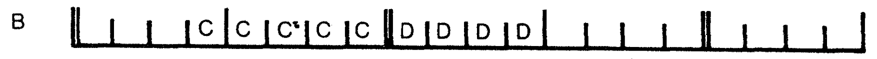
Consider unit A. It would be mapped as:



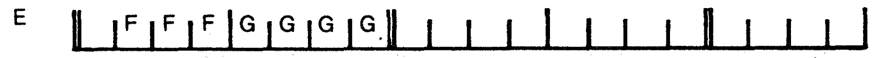
Unit B would initially be mapped as:



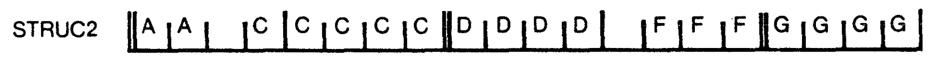
This implies three embedded bytes between C and D. However, C has only an individual byte alignment requirement and so can be moved 3 bytes to the right to give:



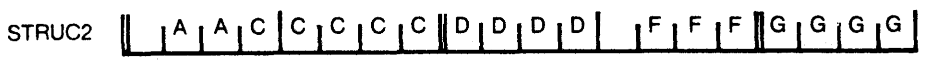
Similarly the final mapping of E will appear as



It is now necessary to join A, B and E together, keeping in mind the alignment requirements. Initially STRUC2 will appear as:



In this case there are 2 embedded bytes. The field A only has a byte alignment requirement and so can be moved to the right, one byte, to eliminate one embedded byte. The second one cannot be eliminated because of the full word boundary alignment requirements of both D and G. Hence STRUC2 will finally appear as:



being in fact 19 bytes long, incorporating one embedded blank.

One impact of this is that the RECSIZE of the relative file in an I/O operation must be of size 19 even though there are only 18 data bytes.

Effect of Alignment Requirements on I/O Operations

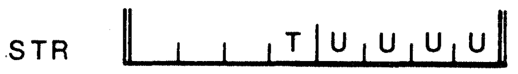
When using LOCATE mode processing it is essential that the programmer gives careful thought to record alignment within the buffer. First let's consider MOVE mode processing, which has no 'alignment problems' then we will look at LOCATE mode processing and the 'alignment problems' which can arise.

MOVE mode

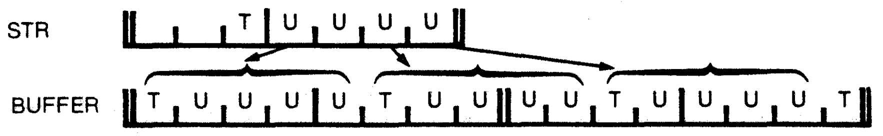
Consider the following structure:

```
 DCL 1 STR,
       2 T CHAR(1),
       2 U FIXED BINARY(31);
```

The map for this structure is:



Suppose that we wish to create a FB-format record with logical records defined by this structure. If we do this by coding WRITE...FROM(STR) statements, then the records would be transferred to the buffer (which always begins on a double word boundary) as follows:



Note that the alignment of the record in the buffer is different from its alignment in the structure STR. The record starts on a double word boundary in the buffer; moreover, the first U (which requires fullword alignment) is aligned on a byte boundary.

This causes no problem if the records are subsequently retrieved using READ...INTO(STR) statements, because the records would first be read into an input buffer and then transferred from this buffer to a correctly aligned structure (this is the output process in reverse).

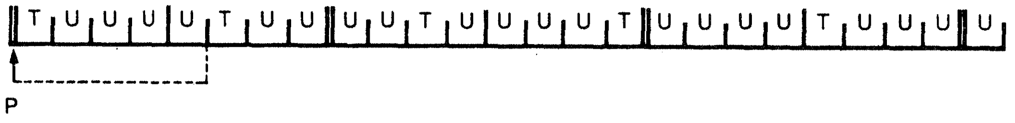
LOCATE mode

The first data byte of the first record in a block is aligned in the buffer on a double word boundary. If an attempt is made to process the records in the buffer itself, where they are not correctly aligned, then problems will occur.

Consider the following based structure:

```
 DCL 1 STR BASED(P),
       2 T CHAR(1),
       2 U FIXED BINARY(31);
```

Suppose we attempt to retrieve the records, created above by READ.. (P) statements, then the records would be moved into the input buffer (which starts on a double word boundary) and the pointer would be set, initially, to this boundary. Thus the structure STR would effectively be overlaid on the data in the buffer as follows:



An attempt to access the data in the first STR.U would be in error, because this name would refer to data aligned on a **byte** boundary, not a fullword as the program expects.

The ERROR condition (ONCODE=8096, specification exception) would be raised.

The same problem would be caused if we attempt to create the record in question using a LOCATE statement. The specified pointer on which the structure is based would be set to address a double word and thus a reference to STR.U would be invalid.

Thus two possible problems of alignment requirements have now been isolated: that of embedded blanks and that of LOCATE mode I/O. Two methods of overcoming these problems will not be discussed.

Structure Design

First, it must be emphasized that this will usually be the function of the systems analyst, not the programmer.

Embedded blanks can be eliminated by redesigning the structure so that the fields are in decreasing order of alignment requirements i.e.

Doublewords

Fullwords

Halfwords

Bytes

ALIGNED/UNALIGNED Attribute

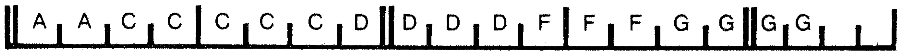
Two attributes which can be added to the declaration statements of variables are ALIGNED/UNALIGNED. ALIGNED is the default, and maintains the alignment requirements that we have been discussing up to this point in the topic.

If the UNALIGNED attribute is specified for a structure (major or minor) then data which normally requires halfword, fullword or doubleword alignment may be aligned on a byte boundary. The rules for structure mapping are the same as above but the reduced alignment requirements are used. This makes the process considerably easier; in fact the map can be written down directly.

Consider the structure

```
 DCL 1 STRUC2 UNALIGNED,
       2 A PIC'99',
       2 B,
         3 C CHAR(5),
         3 D FLOAT(4),
       2 E,
         3 F CHAR(3),
         3 G BINARY FIXED(2Ø);
```

previously considered under the heading 'STRUCTURE MAPPING' but now UNALIGNED. It would be mapped as:



and hence records with that mapping could be read into STR, using either LOCATE or MOVE mode, without any alignment problems (referred to as 'specification exceptions').

Conservation of External Storage

The UNALIGNED attribute can also conserve storage on external medium, in particular when writing out from data aggregates.

Example:

```
 DCL A(1ØØ) BIT(1)  ALIGNED;
```

This array occupies 100 bytes of storage, although only 1 bit of each byte has meaningful data.

```
 DCL B(1ØØ) BIT(1) UNALIGNED;
```

Element bit fields cannot be UNALIGNED but when they are grouped together in a data **aggregate** they **can** be UNALIGNED and so they can start on any bit boundary.

Thus the array B will occupy 13 bytes of storage (the last 4 bits being unused).

Efficiency Considerations

The conservation of external storage must be measured against the programming inefficiencies involved in handling UNALIGNED data. If arithmetic is to be done on an UNALIGNED field then the compiler will generate an extra workfield, correctly aligned, to which the field would be moved. This, of course, will increase the execution time of the program.

If individual bits in a bit array are to be tested or have values assigned to them, it is much more efficient in execution if the array of bits is declared with the ALIGNED attribute.

COBOL Option

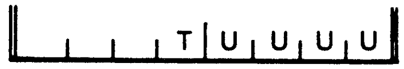
The way in which the COBOL compiler maps a structure is different from the method used by the PL/I compiler.

Example:

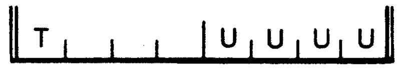
Consider the structure

```
 DCL 1 STR,
       2 T CHAR(1),
       2 U BINARY FIXED(31);
```

The PL/I mapping is:



while the COBOL one is:



Thus difficulties could arise if a COBOL program is attempting to read a data set created by a PL/I program, and vice versa. The problem is overcome by adding the COBOL option in the ENVIRONMENT attribute of the PL/I file declaration:

```
 DCL X FILE ... ENV(... COBOL...);
     WRITE FILE(X) FROM(STR);
```

MOVE Mode I/O must be used. The PL/I compiler would generate not only our STR but a COBOL-type structure. The fields would be moved from the PL/I structure to the COBOL structure, and the WRITE would actually be done from the latter structure.

The record length in the file declaration would have to be 8 and not 5 as before. A COBOL structure would only be produced for those structures which are different and these would be shown on the AGGREGATE listing. The above also applies to data sets created by COBOL programs and read by PL/I programs.

Exercises

1. How much storage is occupied by the following fields:

```
DCL A FIXED DEC(6,2);
DCL B FIXED DEC(7,2);
DCL C BIT(23);
DCL D PIC'$$$9,999V,9';
DCL E FIXED BINARY(17);
```

2. a) In what order does the compiler map element variables?

   b) Why does the compiler use this order?

3. 'The UNALIGNED attribute has no effect on the way arrays are mapped in storage'. Is this statement true or false (give reasons)?

4. Draw the map of the following structure stating the record length:

```
DCL 1 GRADE,
      2 UPPER,
        3 A CHAR(3),
        3 B FIXED BIN(15),
      2 MIDDLE,
        3 C FLOAT DEC(7),
        3 D BIT(8),
      2 E FIXED BIN(31);
```

Answers

1.

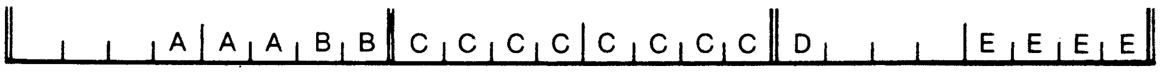
A	4 bytes
B	4 bytes
C	3 bytes
D	10 bytes
E	4 bytes

2. (a) Doublewords
Fullwords
Halfwords
Bytes

b) In order to eliminate any slack bytes of storage.

3. False: Bit arrays may be mapped differently if given the UNALIGNED attribute.

4.



The record length is 21 bytes.

Appendix A

```
                     I      S                 .    P
                                                    D
                 A                   A
     E                  D                     T
         Y                    P                    Y       I
       D                  U              E  T
            M      D                  N            M  D
        O                   G                  O      P
  U  P      D    E   U  P            E      P      D
         I                R                  A         T
Y       I      N  T Y      I       N  T  Y      I    DE
 OG       P      T    OG  M     E     T    OG  M   P   T   D
O      E   N  TU     O     E      ST    R      D    N  UD
     E  D      Y  OG      E  D     D  RO      D  N  .ST    O
M  D    NT   DY R  AM   D    NT   D  R  AM  D     NT     P O
 ND    EN   D  P  R M  ND     ENT   D  P  R M IND    ENT   D  RO  M
  E  N   U  P   A  IN  E  N  TU   P  R  IN   E  N  U   R    IN
P.NDE  ST      GR   EP ND             RA             U          E
ND T STU PR      D  ND   TU  PR  R   D  ND     TU Y   O    ND
E   T   R G    D  EN E   T   R G    D  EN    TU   R R M     END
 ST   P O    I  PE D T ST   P O    N PE D T STU      A  ND  N
S U Y ROGR        NT S UDY ROGR         NT S U Y ROG        EN  T
UD PROGRAM     E     N    UDY PRO.RA     E     N    UD  PRO RA   ND PE     S
  PROG AM  N EPE DE   STU   PR  R    N E END  T ST   PR G AM  N EPE     T  T D
PR GR   INDEPEN EN  S   Y PR GR   I DEPE  ENT ST DY PR GR  INDEP  DENT S   U
 GRAM IN  P ND N  S  D  PR GRAM I   P ND NT S  D  PRO R M I  E EN   T STU   PR
R   NDEP NDEN  S UDY P OGRAM  NDEP NDENT  TUDY PR GRAM IND PEN ENT  TUDY   O
M INDEPEN ENT S UDY PROG AM IND  ENDE T S UDY  ROGRAM I DEPENDEN  STUDY PROGRA
INDEPENDENT STU Y PROGRAM IN EPE DENT ST D  P OGRAM INDE ENDENT STUDY PROGRAM
DEPENDENT STUDY  ROGRAM INDEPENDE T STUDY PR GRAM INDEPENDENT STUDY PROGRAM IN
PENDENT STUDY PROGRAM INDEPENDENT STUDY PROGRAM INDEPENDENT STUDY PROGRAM INDE
NDENT STUDY PROGRAM INDEPENDENT STUDY PROGRAM INDEPENDENT STUDY PROGRAM INDEPE
ENT STUDY PROGRAM INDEPENDENT STUDY PROGRAM INDEPENDENT STUDY PROGRAM INDEPEND
T STUDY PROGRAM INDEPENDENT STUDY PROGRAM INDEPENDENT STUDY PROGRAM INDEPENDEN
STUDY PROGRAM INDEPENDENT STUDY PROGRAM INDEPENDENT STUDY PROGRAM INDEPENDENT
UDY PROGRAM INDEPENDENT STUDY PROGRAM INDEPENDENT STUDY PROGRAM INDEPENDENT ST
Y PROGRAM INDEPENDENT STUDY PROGRAM INDEPENDENT STUDY PROGRAM INDEPENDENT STUD
PROGRAM INDEPENDENT STUDY PROGRAM INDEPENDENT STUDY PROGRAM INDEPENDENT STUDY
OGRAM INDEPENDENT STUDY PROGRAM INDEPENDENT STUDY PROGRAM INDEPENDENT STUDY PR
RAM INDEPENDENT STUDY PROGRAM INDEPENDENT STUDY PROGRAM INDEPENDENT STUDY PROG
M INDEPENDENT STUDY PROGRAM INDEPENDENT STUDY PROGRAM INDEPENDENT STUDY PROGRA
```

Appendix A

Solutions to Machine Exercises

Below are suggested solutions to the exercises. Try not to look at them until you have coded and tested your own solutions. Then compare them and try and pick out the advantages of each method with particular reference to the following:

- ease of reading and understanding
- ease of maintenance
- speed of execution.

Solutions to Exercise A.1

```
EX1:  PROC OPTIONS(MAIN);

DCL  1 INRECORD,
       2 INPUT_RADIUS CHAR(2),
       2 FILLER CHAR(78);

DCL  1 HEADING CHAR(60) INIT
       ('          RADIUS                AREA                VOLUME')

DCL  1 OUTRECORD,
       2 FILLER1 CHAR(14) INIT(' '),
       2 OUT_RADIUS CHAR(2),
       2 FILLER2 CHAR(12) INIT(' '),
       2 OUT_AREA PIC 'ZZZZ9V.99',
       2 FILLER3 CHAR(13) INIT(' '),
       2 OUT_VOLUME PIC 'ZZZZZZ9V.99',
       2 FILLER4 CHAR(1) INIT(' ');

DCL  1 RECORD_FOR_BAD_RADIUS,
       2 FILLER1 CHAR(14) INIT(' '),
       2 BAD_RADIUS CHAR(2),
       2 FILLER2 CHAR(44)
         INIT('                 ****                   ****');
```

```
DCL INFILE RECORD ENV(MEDIUM(SYSIPT,2540) F RECSIZE(80));
DCL OUTFILE RECORD OUTPUT ENV(MEDIUM(SYSLST,1403) F RECSIZE(60));
         /* MEDIUM OPTION FOR DOS ONLY */

DCL MORE_DATA BIT(1) INIT('1'B);
ON ENDFILE (INFILE) MORE_DATA = '0'B;

WRITE FILE (OUTFILE) FROM (HEADING);
```

```
DO WHILE (MORE_DATA);
  RADIUS = INPUT_RADIUS; /* CONVERT TO COMPUTATIONAL FORM */
  IF RADIUS <= 0
    THEN DO;
      BAD_RADIUS = INPUT_RADIUS;
      WRITE FILE (OUTFILE) FROM (RECORD_FOR_BAD_RADIUS);
    END;

    ELSE DO;
      OUT_RADIUS = INPUT_RADIUS;
      AREA = 3.1416 * RADIUS ** 2;
      OUT_AREA = AREA;
      OUT_VOLUME = 4 * AREA * RADIUS / 3;
      WRITE FILE (OUTFILE) FROM (OUTRECORD);
    END;
  READ FILE (INFILE) INTO (INRECORD);
END;  /* END OF THE DO WHILE */
END EX1;
```

Solution to Exercise A.2A

```
EXA2: PROC OPTIONS (MAIN);
DCL (NUMBER, MEAN, TOTAL, VARIANCE) FLOAT(6);
DCL 1 INRECORD,
      2 SAMPLE(40) CHAR(2);
DCL COMPSAMP(40) FLOAT(6);
DCL 1 OUTRECORD,
      2 FILLER1 CHAR(22) INIT('     NO. OF SAMPLES = '),
      2 OUTNO PIC `Z9',
      2 FILLER2 CHAR(12) INIT ('     MEAN = '),
      2 OUTMEAN PIC `Z9V.9',
      2 FILLER3 CHAR(16) INIT('     VARIANCE = '),
      2 OUTVAR PIC `ZZZZ9V.9',
      2 FILLER4 CHAR(7) INIT (`');
DCL OUTMEANSTARS CHAR(4) DEFINED OUTMEAN;
DCL OUTVARSTARS CHAR(7) DEFINED OUTVAR;

DCL CARDIN RECORD ENV(MEDIUM(SYSIPT,2540) F RECSIZE(80));
DCL REPORT RECORD OUTPUT ENV(MEDIUM(SYSLST,1403) RECSIZE(70) F);
   /* MEDIUM OPTION DOS ONLY */

DCL MORE_DATA BIT(1) INIT(`1'B);
ON ENDFILE(CARDIN) MORE_DATA = `0'B;
READ FILE(CARDIN) INTO (INRECORD);
```

```
DO WHILE (MORE_DATA);
     NUMBER, TOTAL, VARIANCE = Ø;
     I = 1;  /* INDEX FOR SAMPLE ARRAY */
     DO WHILE (SAMPLE(I) ¬= ` ');
        NUMBER = NUMBER + 1;
        COMPSAMP(I) = SAMPLE (I);  /* CONVERT TO COMPUTATIONAL FORM */
        TOTAL = TOTAL + COMPSAMP(I);  /* ACCUMULATE SUM OF SAMPLES */
        I = I + 1;  /* TRY NEXT ITEM */
        IF I > 40 THEN LEAVE;
     END;
     OUTNO = NUMBER;
     IF NUMBER = Ø THEN DO;
                              OUTMEANSTARS = `****';
                              OUTVARSTARS = `   ****';
                           END;
                       ELSE DO;
                              OUTMEAN, MEAN = TOTAL/NUMBER;
                              DO J = 1 TO I - 1;
                              VARIANCE = VARIANCE + (COMPSAMP(J)-MEAN)**2;
                              END;
                              OUTVAR = VARIANCE/NUMBER;
                           END;
     WRITE FILE(REPORT) FROM (OUTRECORD);
     READ FILE(CARDIN) INTO (INRECORD);
 END; /* END OF THE OUTER DOWHILE */
 END EXA2;
```

Solution to Exercise A.2B

```
EXA3: PROC OPTIONS (MAIN);
DCL CARDIN RECORD ENV(MEDIUM(SYSIPT,2540) F RECSIZE(80));
DCL OUTFILE RECORD OUTPUT ENV(MEDIUM(SYSLST,1403) F RECSIZE(82));
    /* MEDIUM OPTION DOS ONLY */

DCL 1 INRECORD,
        2 ITEM_NUMBER CHAR(6),
        2 DESCRIPTION CHAR(20),
        2 UNIT_COST PIC '99V99',
        2 UNIT_SELL PIC '99V99',
        2 QTY_ON_HAND PIC '9999',
        2 FILLER CHAR(42);
DCL HEADING1 CHAR(82) INIT(' ITEM          ITEM              ON   UNIT
TOTAL ITEM  UNIT TOTAL ITEM    PROFIT');
DCL HEADING2 CHAR(82) INIT('  NO     DESCRIPTION             HAND COST
   COST        SELL SELL PRICE    RATIO');
DCL BLANK_LINE CHAR(82) INIT('');
```

```
DCL 1 OUTRECORD,
        2 OUT_ITEM_NUMBER CHAR(6),
        2 FILLER1 CHAR(2) INIT(''),
        2 OUT_DESCRIPTION CHAR(20),
        2 FILLER2 CHAR(2) INIT(''),
        2 OUT_QTY_ON_HAND PIC 'ZZZ9',
        2 FILLER3 CHAR(2) INIT(''),
        2 OUT_UNIT_COST PIC 'ZZV.99',
        2 FILLER4 CHAR(2) INIT(''),
        2 OUT_TOTAL_COST PIC '$$$,$$9V.99',
        2 FILLER5 CHAR(2) INIT(''),
        2 OUT_UNIT_SELL PIC 'ZZV.99',
        2 FILLER6 CHAR(2) INIT(''),
        2 OUT_TOTAL_SELL PIC '$$$,$$9V.99',
        2 FILLER7 CHAR(3) INIT(''),
        2 PROFIT_RATIO_GROUP,
          3 PROFIT_RATIO PIC 'ZV99',
          3 PERCENT CHAR(4);
DCL PROFIT_RATIO_STARS CHAR(7) DEFINED PROFIT_RATIO_GROUP;
DCL (COMP_UNIT_COST, COMP_UNIT_SELL) FIXED DEC(5,2);
DCL COMP_QTY_ON_HAND FIXED DEC(5);
DCL (COMP_TOTAL_COST,FINAL_TOTAL_COST INIT(0)) FIXED DEC(7,2);
DCL (COMP_TOTAL_SELL,FINAL_TOTAL_SELL INIT(0)) FIXED DEC(7,2);
DCL 1 LAST_OUTRECORD,
        2 FILLER1 CHAR(43) INIT(''),
        2 OUT_FINAL_TOTAL_COST PIC '$$$,$$$V.99',
        2 FILLER 2 CHAR(9) INIT(''),
        2 OUT_FINAL_TOTAL_SELL PIC '$$$,$$$V.99',
        2 FILLER3 CHAR(10) INIT('');
```

```
WRITE FILE (OUTFILE) FROM (HEADING1);
WRITE FILE (OUTFILE) FROM (HEADING2);
WRITE FILE (OUTFILE) FROM (BLANK_LINE);
DCL MORE_DATA BIT(1) INIT('1'B);
ON ENDFILE (CARDIN) MORE_DATA = `0`B;
READ FILE (CARDIN) INTO (INRECORD);
DO WHILE (MORE_DATA);
   OUT_ITEM_NUMBER = ITEM_NUMBER;
   OUT_DESCRIPTION = DESCRIPTION;
   COMP_UNIT_COST = UNIT_COST;
   COMP_UNIT-SELL = UNIT-SELL;
   COMP-QTY_ON_HAND = QTY_ON_HAND;
   COMP_TOTAL-COST = COMP_QTY-ON_HAND * COMP_UNIT_COST;
   FINAL_TOTAL_COST = FINAL_TOTAL-COST + COMP_TOTAL_COST;
   COMP_TOTAL_SELL = COMP_QTY_ON_HAND * COMP_UNIT_SELL;
   FINAL_TOTAL_SELL = FINAL_TOTAL_SELL * COMP_TOTAL_SELL;
   OUT_QTY_ON_HAND = COMP_QTY_ON_HAND;
   OUT_UNIT_COST = COMP_UNIT_COST;
   OUT_TOTAL_COST = COMP_TOTAL_COST;
```

```
   OUT_UNIT_SELL = COMP_UNIT_SELL;
   OUT_TOTAL_SELL = COMP_TOTAL_SELL;
   DCL COMP_PROFIT_RATIO FIXED DEC(3,2);
   COMP_PROFIT_RATIO = ((COMP_UNIT_SELL - COMP_UNIT_COST)/
                        COMP_UNIT_COST + .005;
   IF COMP_PROFIT_RATIO <= .25
      THEN PROFIT_RATIO-STARS = '*******';
      ELSE DO;
            PROFIT_RATIO = COMP_PROFIT_RATIO;
            PERCENT = ' PCT';
            END;
   WRITE FILE(OUTFILE) FROM (OUTRECORD);
   READ FILE (CARDIN) INTO (INRECORD);
END;  /* END OF THE DOWHILE */
WRITE FILE (OUTFILE) FROM (BLANK_LINE);
OUT_FINAL_TOTAL_COST = FINAL_TOTAL_COST;
OUT_FINAL_TOTAL-SELL = FINAL_TOTAL-SELL;
WRITE FILE (OUTFILE) FROM (LAST_OUTRECORD);
END EXA3;
```

Solution to Exercise B

```
EXB: PROC OPTIONS(MAIN);

     /*****************/
     /*  DECLARATIONS  */
     /*****************/
                                                          /* FILES */
     DCL CARFILE RECORD
         ENV(MEDIUM(SYSIPT,254Ø) RECSIZE(8Ø) F);          /* MEDIUM OPTION  */
     DCL PRFILE RECORD OUTPUT                             /*       IS       */
         ENV(MEDIUM(SYSLST,14Ø3) RECSIZE(7Ø) F);          /* DOS/VS  ONLY   */

     DCL 1 NAMEREC               BASED(P),
           2 ACODE               CHAR(1),
           2 MAKE                CHAR(2Ø);

     DCL 1 PRICEREC              BASED(P),
           2 BCODE               CHAR(1)
           2 DETAILS(4),
             3 NPRICE            PIC'(5)9V99',
             3 CPRICE            PIC'(5)9V99',
             3 AGE               PIC'99';

     DCL 1 PRICERECA             BASED(P),
           2 CODE                CHAR(1),
           2 ENTRY(4)            CHAR(16);

     DCL LINE                    CHAR(7Ø);
```

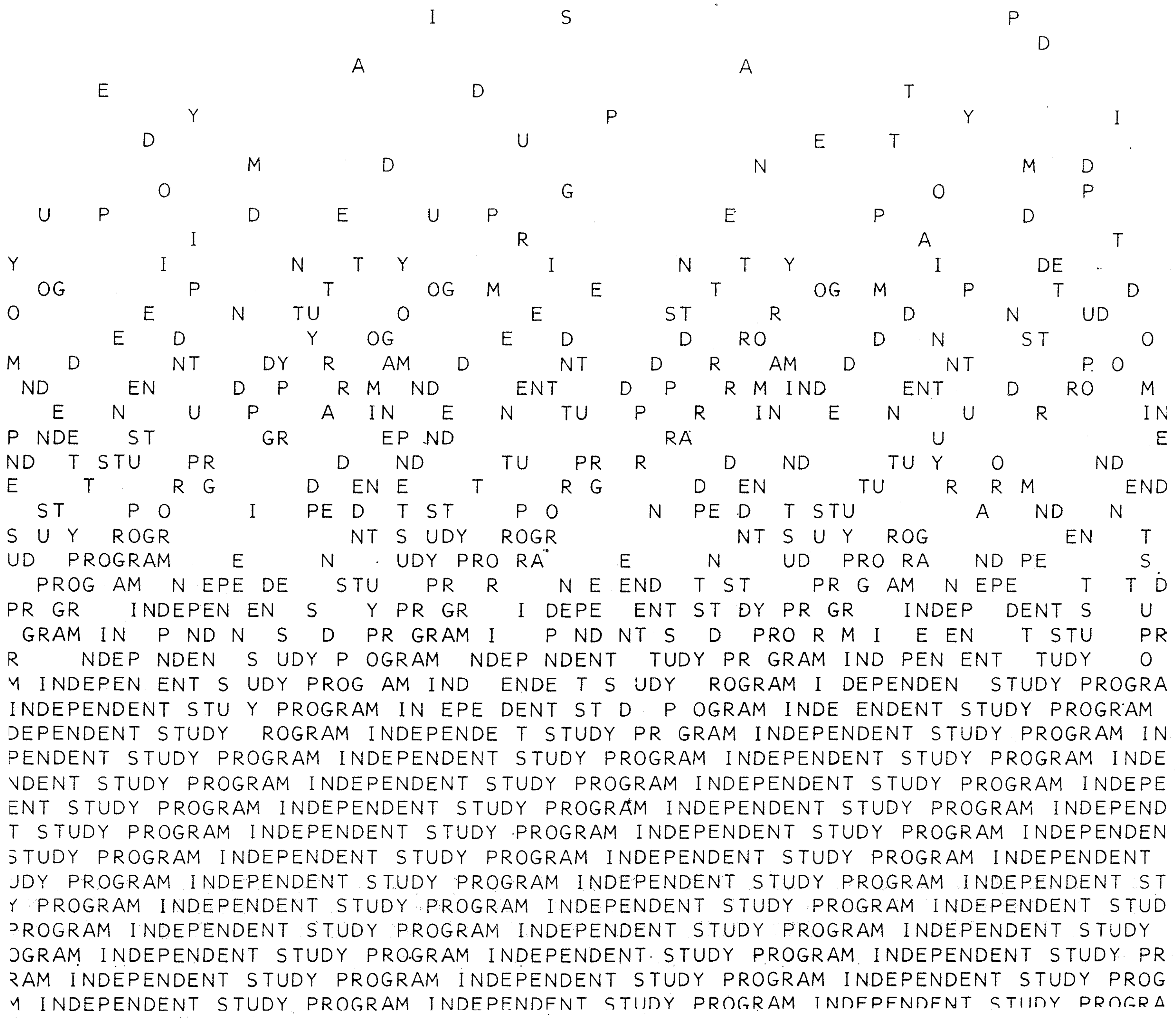
```
DCL 1 HEADING              DEF LINE,
      2 H1                 CHAR(2Ø),
      2 H2                 CHAR(3Ø),
      2 H3                 CHAR(5),
      2 H4                 CHAR(11),
      2 H5                 CHAR(4),

DCL 1 PLINE                DEF LINE,
      2 PMAKE              CHAR(2Ø),
      2 PFILL1             CHAR(2Ø),
      2 PPERCENT           PIC'--9',
      2 PFILL2             CHAR(1),
      2 PDEPREC            PIC'----9V.99'
      2 PFILL3             CHAR(4),
      2 PAGE               PIC'Z9',
      2 PFILL4             CHAR(8),
      2 PFLAG              CHAR(4);

DCL 1 SUMMARY              DEF LINE,
      2 S1                 CHAR(13),
      2 SHASHTOT           PIC'ZZZ,ZZ9V.99',
      2 S2                 CHAR(16),
      2 S3                 CHAR(27),
      2 SDEPREC            PIC'Z9'
      2 S4                 CHAR(1),
      2 S5                 CHAR(1);
```

```
                                         /* WORKING FIELDS */
DCL HASHTOT           FIXED(9,2) INIT(Ø);
DCL NCOST             FIXED(7,2);
DCL PERCENT           FIXED(3,1);
DCL NO                FIXED(3,Ø) INIT(Ø);
DCL MEANPER           FIXED(7,1) INIT(Ø);
DCL CCOST             FIXED(7,2);
DCL YRS               FIXED(3,Ø);
DCL SW                BIT(1) INIT(`Ø'B);
DCL I                 FIXED BIN(15,Ø);
DCL P                 PTR;

/********************/
/*  INITIALIZATION  */
/********************/

H1 = `MAKE';
H2 = `ANNUAL DEPRECIATION: (%';
H3 = `$)';
H4 = `AGE';
H5 = `FLAG';

PUT PAGE;
WRITE FILE(PRFILE) FROM(LINE);
LINE = ` ';
```

```
/*****************/
/*  MAIN LOGIC  */
/*****************/

DO WHILE (SW = 'Ø'B);
  READ FILE(CARFILE) SET(P);
  IF ACODE = 'A'
    THEN DO;                                                /* NAME RECORD */
          PMAKE = MAKE;
         END;
    ELSE DO;
          IF BCODE = 'B'
           THEN DO;                                         /* PRICE RECORD */
                  DO I = 1 TO 4
                         WHILE(ENTRY(I) ¬= ' ');
                    YRS = AGE(I);
                    NCOST = NPRICE(I);
                    CCOST = CPRICE(I);

                    PERCENT = 1ØØ * (1-(CCOST/NCOST)**(1/YRS));
                    IF PERCENT < 1Ø.5 THEN PFLAG = '****';
                    PPERCENT = PERCENT + 0.5;
                    MEANPER = MEANPER + PERCENT;
                    NO = NO + 1;

                    PDEPREC = (NCOST-CCOST)/YRS + 0.005;

                    HASHTOT = HASHTOT + CCOST;
                    PAGE = YRS;
                    WRITE FILE(PRFILE) FROM(LINE);
                    LINE = ' ';
                  END;
                END;
```

```
           ELSE DO;                                         /* TRAILER CARD */
                  S1 = 'HASH TOTAL =';
                  S3 = 'MEAN ANNUAL DEPRECIATION';
                  S5 = '%';
                  SHASHTOT = HASHTOT;
                  SDEPREC = MEANPER/NO + 0.5;
                  WRITE FILE(PRFILE) FROM(LINE);
                  SW = '1'B;
                END;
         END;
END;
END EXB;
```

Appendix B



Appendix B

Additional PL/I Problems and Solutions

The exercises contained on the following pages have been designed as practice problems for students learning PL/I. Included among them are examples of a scientific and commercial nature. An instructor may assign a subset of these problems as required exercises, and recommend others as optional exercises. Depending on their background individual students may elect problems of particular interest to them, and should also be encouraged to work on problems of their own design.

Suggested solutions may be found in the latter part of the appendix. The solutions were tested on an OS System. However, most of these solutions will also work in a DOS environment with appropriate changes to file declarations where applicable.

PL/I Problems

Problem 1. Sum of Squares

Compute the sum of squares of the first 10, the first 50, and the first 100 integers, that is, write a program to evaluate

$$S1 = \sum_{i=1}^{10} i^2,\ S2 = \sum_{i=1}^{50} i^2, \text{ and } S3 = \sum_{i=1}^{100} i^2$$

and print out the three sums S1, S2, and S3.

Problem 2. Minimax

Read a set of numbers from a card into an array. Assuming these numbers to be in random order find the largest and smallest element.

a. Write the program for a fixed (constant) number of elements.

b. Adjust the program so that it will work for a variable number (n) of elements, where the value for n is punched in the leading field of the input card.

c. Convert this program into a subroutine that can be called with any list of numeric data to produce the minimum and maximum element.

Problem 3. Indian Problem

Assume that the Indians who sold Manhattan Island in 1627 for \$24.00 deposited that amount. How much *interest* would they have accumulated in their account if the interest is compounded yearly at 3.5 percent, and rounded to the nearest cent each year? Print out the original principal, the rate, total accumulated interest, and total accumulated balance in the account.

Problem 4. Sort

Consider a set of elements (numeric or alphabetic data) contained in an array in random sequence. Write a program which will sort these elements into ascending sequence.

a. Apply the "interchange" method.

b. Sort by "ranking" (tallying); this technique minimizes the amount of data movement, and incidentally illustrates the use of subscripted subscripts.

Problem 5. Table Look-up

Consider a set of arguments x_i and corresponding functions $f(x_i)$, i=1,2, ... n, arranged in tabular form:

x_1	$f(x_1)$
x_2	$f(x_2)$
x_3	$f(x_3)$
.	.
.	.
.	.
x_n	$f(x_n)$

The choice of tables is left to the student; the x's and f's could be of arithmetic type (for example, tables of integers and their squares), or character strings (e.g., names of people and their telephone numbers).

Assume the pair of tables to be contained in two separate one-dimensional arrays. (The student may want to consider alternative data organizations, such as constructing both tables in a two-dimensional array, or an array of structures, or a structure composed of two arrays.)

Given a 'search' argument $\bar{x}$ in the range $x_1 \leq \bar{x} \leq x_n$ find the x_i (if it exists) equal to $\bar{x}$, and obtain the corresponding $f(x_i)$. Test the program with several search arguments read from cards. Provide for an error routine if the search argument is not contained in the table.

a. Perform sequential search, i.e. scan table starting with x_1.

b. Perform binary search. (For simplicity let table consist of 32 elements.) In order to use this method the table of arguments must be in sequence.

   The search argument is then determined to be in either the lower or upper half of the table by comparing to a middle element. Then the half of the table that does not contain the search argument can be ignored and the process repeated on the remaining half. This process is repeated until the remaining table is reduced to one element. A large table can be searched quickly this way. For example, a table of 1000 arguments would only require 10 lookups to find a required search argument.

   Further, it should be noted that for equally spaced numeric arguments, x_i, the subscripts required to fetch $f(x_i)$ can be computed thus eliminating the need for a table search altogether.

Problem 6. Concatenation of String Elements

Consider a list of character strings of variable length in an array with a variable number of elements. For example, the strings might represent names of cities such as

```
ATLANTA
BOSTON
CHICAGO
DENVER
   .
   .
   .
WASHINGTON
```

Construct a single string composed of the concatenation of all array elements separated by commas.

Problem 7. Square Root

Write a program to compute $y=\sqrt{x}$ by the following iteration method. Obtain successive approximations $y_0, y_1, y_2, \ldots$ whose values will converge toward the root. Let the initial (old) approximation $y_0 = x/2$. Then compute new approximations by repeated evaluation of $y_{i+1} = 1/2\ (yi + x/yi)$ until the condition $|y_{i+1} - yi| < \epsilon$ is satisfied, where ϵ is a constant. (Suggested value for $\epsilon = .001$, which will result in a root correct to three decimal places.) The last evaluated y is the root.

For a concrete example, let x=36. Then $y_0 = x/2 = 18$, and

$y_1 = 1/2\ (18 + 36/18) = 10$,
$y_2 = 1/2\ (10 + 36/10) = 6.8$, etc.

until the absolute value of the difference between a pair of successive y's is less than .001. (Write your own code for the absolute value rather than using the built-in function.)

Problem 8. Polynomial

Write a program to evaluate a set of 3rd degree polynomials

$f(x) = a_3x^3 + a_2x^2 + a_1x + a_0 = ((a_3x + a_2)\ x + a_1)x + a_0$

for x=1,2,...,30.

Generalize the program so that it will work for nth degree polynomials:

$a_nx^n + a_{n-1}x^{n-1} + a_{n-2}x^{n-2} + \ldots + a_1x + a_0$

Problem 9. Series Expansion

Write a program to compute several terms of the series:

$e^x = 1 + x/1! + x^2/2! + x^3/3+ \ldots$

$\sin x = x - x^3/3! + x^5/5! - x^7/7! + \ldots$

$\cos x = 1 - x^2/2! + x^4/4! - x^6/6! + \ldots$

Compare the values computed with those obtained from the built-in functions, for x = 0, .1, .2, etc., up to 1.0.

Problem 10. Integration

Evaluate the definite integral $\int_0^1 (1/1+x^2)\,dx$ by Simpson's rule. Let the interval $h = \Delta x = .01$ and let $y = 1/(1 + x * x)$.

Compute $(h/3)(1 + 4y_1 + 2y_2 + 4y_3 + 2y_4 + \ldots + 2y_{n-2} + 4y_{n-1} + 1/2)$

Ans. $\pi/4$ or about .7853982

Problem 11. Prime Numbers

Generate and list all prime numbers <1000

a. Test for divisibility.

b. Sieve of Eratosthenes. (Hint: Use bit strings to sift out primes.)

Problem 12. Perfect Numbers

In mathematics, a number is considered to be perfect if the sum of all of its divisors (except itself) is equal to the original number.

Example: The divisors of 6 are 1, 2, and 3.

```
        1 + 2 + 3 = 6
```

Write a program that will determine all perfect numbers from 1 to 100, and which will exhibit as output all the divisors of the number (except itself), and the perfect number.
(Ans: 6, 28)

Problem 13. Factorials and Binomial Functions

a. Write a function procedure for computing factorials (n!).

b. Write a function procedure which returns binomial coefficients n!/(n-k)!k! and utilizes the function procedure written for the factorials.

c. Write a test program to evaluate and print out the binomial coefficients for n = 0, 1, 2, ..., 10 with k ranging from 0 to n. For each n print the results on a separate line.

Problem 14. String Manipulation

a. A character-string of length n (say n=100) may contain an arbitrary number of * (asterisks), possibly none. Also allow for the occurrence of adjacent asterisks. For example a portion of text in the string might be

```
'PL/I*FORTRAN*COBOL**ALGOL*APL*BASIC etc.'.
```

Count the number of asterisks in the string, and extract all substrings delimited by asterisks. (The substrings are sequences of characters appearing to the left of the first asterisk, between successive asterisks, and to the right of the last asterisk.) Print the extracted substrings on consecutive lines leaving a blank line for a null substring.

b. A character-string of varying length may contain blanks freely intermixed with other characters. Replace it by a new string obtained by "squeezing" out all blanks.

c. Scan a bit string of varying length to find the longest sequence of consecutive 1 bits. Print its length and the position of the first and last bit of that sequence within the containing string.

d. Write a subroutine that scans a string (1st argument) for a pattern (2nd argument) counting the frequency of that pattern within the string; put the count into a 3rd argument.

Assume that the first argument has a maximum length of 400 characters, the second argument has a maximum length of 20 characters. Write a test program which invokes the subroutine.

Problem 15. Brand Names

Create brand names by generating the 125 possible combinations (concatenations) of five prefixes, five roots, and five suffixes. Each prefix, root, and suffix is to be a maximum of six characters. Print 5 brand names to a line.

Example

	Prefixes	*Roots*	*Suffixes*
1.	SUPER	DRIVO	MATIC
2.	MAXI	BUICK	------
3.	BEAU	------	------
4.	GRAND	------	------
5.	MINI	------	------

Problem 16. Combinations of Coins

Write a program to calculate the number of different ways a dollar bill can be broken into change (i.e. 1-50¢, 1-25¢, 25-1¢ is one way; 2-25¢, 2-10¢, 6-5¢ is another). Output the result. (Answer is 292).

Problem 17. Calendar Problem

If Y is the year (1901 to 2099) the day of the week (D1) for January 1 counting from SUNDAY=0 may be computed as follows:

```
K = Y - 1901;
D1 = MOD(TRUNC(K/4) + K + 2, 7);
```

Using the value of D1 prepare a calendar for any year read in as an input value. The first page is to look something like this:

```
                                 JANUARY              1981

      SUN          MON          TUE          WED          THUR         FRI          SAT

I------------I------------I------------I------------I------------I------------I------------I
I            I            I            I            I1           I2           I3           I
I            I            I            I            I            I            I            I
I            I            I            I            I            I            I            I
I            I            I            I            I            I            I            I
I------------I------------I------------I------------I------------I------------I------------I
I4           I5           I6           I7           I8           I9           I10          I
I            I            I            I            I            I            I            I
I            I            I            I            I            I            I            I
I            I            I            I            I            I            I            I
I------------I------------I------------I------------I------------I------------I------------I
I11          I12          I13          I14          I15          I16          I17          I
I            I            I            I            I            I            I            I
I            I            I            I            I            I            I            I
I            I            I            I            I            I            I            I
I------------I------------I------------I------------I------------I------------I------------I
I18          I19          I20          I21          I22          I23          I24          I
I            I            I            I            I            I            I            I
I            I            I            I            I            I            I            I
I            I            I            I            I            I            I            I
I------------I------------I------------I------------I------------I------------I------------I
I25          I26          I27          I28          I29          I30          I31          I
I            I            I            I            I            I            I            I
I            I            I            I            I            I            I            I
I            I            I            I            I            I            I            I
I------------I------------I------------I------------I------------I------------I------------I
```

Use record I/O in this problem for both input and output.

Problem 18. Student Performance Problem

An instructor keeps a class record for each student in his courses. Each student's record is punched into one or more cards in *record I/O* format. The input for each student is a *structure* with the following elements

Course name	
Student's name	(last name, first name, middle initial)
Attendance	(an array of 20 items, each a single character, where '1' represents "present", and '0' represents "absent" for 20 lecture periods)
	Three test grades and a final exam grade. (Grades are integers 0 - 100)

At the end of each course the instructor wishes to list each student's performance. The report is printed in record-directed format and contains the following items:

Course name	(print starting in position 5 of first line of a page)
Student's name	(print on the left, 3 lines below course name or previous student's data)
Attendance	(print 'LECTURES ATTENDED' and the number of the next line, indented to the right)
Status	(on the following line, below attendance, print either
'INCOMPLETE'	if absent during a test or exam session; (these sessions are 4, 9, 13, 20), or
'DROPPED'	if absent every session after the third, or
'PASSED'	if average (see below) is 65 or above, or
'FAILED'	if average (see below) is below 65.)
Grades	(print test and exam grades on a single line below Status information, but do not list if status is 'INCOMPLETE' or 'DROPPED'.)
Average	(Print on line below grades, but omit if status is 'INCOMPLETE' or 'DROPPED'. The final exam is twice as difficult as the tests and should be weighted appropriately in the computation of average).

Make up your own test cases for a machine run.

The following card format is suggested for input data:

Course name:	Col. 1-20
Student's name:	Col. 21-40 (10 columns for Last, 9 for First, 1 for Middle)
Attendance:	Col. 41-60
Grades:	Col. 61-72 (3 columns for each of 4 grades)

Problem 19. French Plurals

Read in a series of French words from cards each of which do not exceed 10 characters. Each card may contain 1 to 8 words beginning in columns 1, 11, 21, 31, etc. Print each word and its plural on a separate line. If a word ends in 'S', 'X', or 'Z' the plural is the same. If a word ends in 'AL' the plural is formed by changing the 'L' to 'UX'; if a word ends in 'EU' or 'AU' the plural is formed by adding 'X'; otherwise the plural is formed by adding 'S'.

Problem 20. Check Writing Problem

Using list-directed input read in a series of money values up to 9999.99 and generate the equivalent value in words. Center the result into a 73 character string preceding the words and following the words by asterisks. For each value exhibit on a separate line the original value and the 73 character string. Test with a wide variety of money values.

Problem 21. EASTER

Easter is the first Sunday following the Paschal Full Moon which happens upon or next after March 21. The Paschal Full Moon is the 14th day of a lunar month reckoned according to an ancient ecclesiastical computation and not the astronomical full moon.

To find the date (Gregorian Calendar) of Easter Day apply the following rule:

DIVIDE	*BY*	*QUOTIENT*	*REMAINDER*
The year Y	19	A	
The year Y	100	B	C
B	4	D	E
B+8	25	F	
B-F+1	3	G	
19A+B-D-G+15	30		H
C	4	I	K
32+2E+2I-H-K	7		J
A+11H+22J	451	M	
H+J-7M+114	31	N	P

Then N is the month and P+1 is the day of the month on which Easter falls.

Write a program to develop a table of Easter dates for the 20th Century in the following form:

```
                              EASTER DATES
APRIL   7 1901     APRIL   4 1926     MARCH 25 1951     APRIL  18 1976
  -     - 1902       -     - 1927       -     - 1952      -     - 1977
          .                  .                  .                  .
          .                  .                  .                  .
          .                  .                  .                  .
          .                  .                  .                  .
          .                  .                  .                  .
          .                  .                  .                  .
          .                  .                  .                  .
          .                  .                  .                  .
       1925               1950               1975               2000
```

Problem 22. Chess

a. Eight queens can be placed on a chessboard such that no queen in on the same row, column or diagonal as another queen. Write a program that will determine all such possible arrangements for 8 queens. (Answer 192.)

   Optionally, generalize the problem for N queens on an N by N board and develop the solutions for N = 4, 5, 6, 7, and 8. Illustrate with a board diagram the first solution for each N and print the solutions as a list of row positions. For example, for N=4 one solution is *2 4 1 3* indicating a queen in the second square of the first row, a queen in the fourth square of the second row, etc.

b. In chess the Knight can move from the corner of any orthogonally placed 2 squares by 3 squares rectangle to the opposite corner. Place a Knight on any square of a chessboard. In 63 moves have the Knight visit every other square once and only once. Write a program that will attempt to find such a "Knight's tour". Illustrate the successful tours.

PL/I Solutions

Problem 1. Sum of Squares

```
SUMSQ: PROC OPTIONS(MAIN);
DCL (S1,S2,S3,I) FIXED BIN(31);
S1 = 1;
  DO I = 2 TO 10;
  S1 = S1 + I*I;
  END;
S2 = S1;
  DO I = 11 TO 50;
  S2 = S2 + I * I;
  END;
S3 = S2;
  DO I = 51 to 100;
  S3 = S3 + I*I;
  END;
PUT DATA(S1, S2, S3);
END SUMSQ;
```

```
S1=           385      S2=            42925      S3=             338350;
```

If the user did not code precision and used fixed binary, only the first correct answer would be obtained because the default would be FIXED BIN (15) which can only go up to 32767.

If FIXED DEC of default precision 5 were used, only two correct answers would be obtained as the maximum value for FIXED DEC (5) is 99999.

If FLOAT or FLOAT DEC were specified, all 3 answers above would be obtained, but the solution would be a bit longer in execution time.

Problem 2. Minimax

Solution A: Fixed Number of Elements

```
MINMAXA: PROC OPTIONS(MAIN);
/* FIND LARGEST AND SMALLEST FOR FIXED NUMBER OF ITEMS */
 /* TRY IT FOR A DOZEN ITEMS */
DCL (ARRAY(12), MIN, MAX) FIXED DEC(5);
/* OTHER NUMERIC DATA TYPES ARE, OF COURSE, POSSIBLE */
GET LIST(ARRAY);
MIN,MAX = ARRAY(1);
DO K = 2 to 12;
  IF ARRAY(K) < MIN
    THEN MIN = ARRAY(K);
    ELSE IF ARRAY (K) > MAX
      THEN MAX = ARRAY (K);
END;
PUT DATA(ARRAY,MIN,MAX);
END MINMAXA;
```

Solution B: Variable Number of Elements

```
MINMAXB: PROC OPTIONS(MAIN);
/* FIND LARGEST AND SMALLEST FOR VARIABLE NUMBER OF ITEMS (N),
  WHERE N IS THE FIRST FIELD ON FIRST INPUT CARD */
GET LIST(N);
  B: BEGIN;
  DCL (ARRAY(N),MIN,MAX) FIXED DEC(5);
  GET LIST (ARRAY);
  MIN,MAX = ARRAY(1);
  DO K=2 TO N;
    IF ARRAY(K) < MIN
      THEN MIN = ARRAY(K);
      ELSE IF ARRAY (K) > MAX
        THEN MAX = ARRAY (K);
  END;
  PUT DATA(ARRAY,MIN,MAX);
  END B;
END MINMAXB;
```

Solution C: Minimax as a Subroutine

```
TEST: PROC OPTIONS(MAIN);
 /* THIS IS A SHORT TEST PROGRAM TO TEST MINMAXC,
  THE REQUIRED SUBROUTINE */
DCL LIST(7) FIXED DEC INIT(8,31,14,37,-3,17,0);
DCL (MINI,MAXI) FIXED DEC, NUMBER INIT(7);
CALL MINMAXC (NUMBER,LIST,MINI,MAXI);
PUT DATA(LIST,MINI,MAXI):
MINMAXC: PROC(N,ARRAY,MIN,MAX);
/* SUBROUTINE TO COMPUTE MINIMUM AND MAXIMUM OF AN ARRAY OF N
  ELEMENTS. ASSUMED DATA TYPE IS FIXED BIN(15) FOR N AND FIXED DEC(5)
  FOR OTHER DATA */
 DCL (ARRAY(*), MIN, MAX) FIXED DEC;
/* NOTE USAGE OF THE * HERE FOR AN ARRAY WITH ONE DIMENSION
 PASSED AS A PARAMETER */
  MIN,MAX = ARRAY(1);
  DO K=2 TO N;
    IF ARRAY(K) < MIN
      THEN MIN = ARRAY(K);
      ELSE IF ARRAY (K) > MAX
        THEN MAX = ARRAY (K);
  END;
END MINMAXC;
END TEST;
```

Problem 3: Indian Problem

```
INDIAN: PROC OPTIONS(MAIN);
  /* COMPUTE INTEREST ON $24.00 AT 3.5% INTEREST COMPOUNDED
     ANNUALLY AND ROUNDED TO THE NEAREST PENNY EACH YEAR */
DCL (ORIG_PRINC INIT(24.00), BALANCE, INTEREST) FIXED DEC(11,2);
DCL RATE FIXED DEC (3,3) INIT(.035);
BALANCE = ORIG_PRINC;
DO K = 1628 TO 1981; /* FIRST INTEREST PAID IN 1628, LAST IN 1981 */
  BALANCE = BALANCE * (1 + RATE) +.005;
END;
INTEREST = BALANCE - ORIG_PRINC;
PUT DATA;
END INDIAN;
```

```
 RATE= 0.035          ORIG_PRINC=          24.00
 K=     1982;


      BALANCE=    4667547.87  INTEREST=    4667523.87
```

Problem 4: Sort

Solution A: Interchange Method

```
SORTA: PROC OPTIONS(MAIN);
   /* SORT BY INTERCHANGE--ILLUSTRATE 10 CHARACTER ELEMENTS */
DCL ARRAY(10) CHAR(7) INIT('BAKER','GEORGIE','DOG','ABLE','FOX',
     'CHARLIE','ITEM','HARVEY','JUDY','EASY');
DCL TEMP CHAR(7);
DO UNTIL (INTERCHANGE=0); /* TESTED AT END OF LOOP */
INTERCHANGE =0;
  DO J = 1 TO 9;
    IF ARRAY(J) > ARRAY(J+1)
       THEN DO;
              TEMP = ARRAY(J);
              ARRAY(J) = ARRAY(J+1);
              ARRAY(J+1) = TEMP;
              INTERCHANGE = 1;
            END;
  END;
END; /* IF INTERCHANGE STILL 0 NO INTERCHANGE TOOK PLACE
        AND ARRAY IS NOW SORTED */
PUT EDIT (ARRAY) (COL(1),A(7));
END SORTA;
```

```
ABLE
BAKER
CHARLIE
DOG
EASY
FOX
GEORGIE
HARVEY
ITEM
JUDY
```

Solution B: Ranking

```
SORTR: PROC OPTIONS(MAIN);
/* SORT BY RANKING--EXAMPLE WITH VARIABLE NUMBER OF NUMERIC ELEMENTS */
GET LIST(N);
  B: BEGIN; DCL (AIN, AOUT, RANK) (N) FIXED BIN;
     GET LIST(AIN);
     RANK = 1;
     DO I = 1 TO N-1;
        DO J = I+1 to N;
           IF AIN(I) > AIN(J)
              THEN RANK(I) = RANK(I) + 1;
              ELSE RANK(J) = RANK(J) + 1;
        END;
     END;
     DO I = 1 TO n;
        AOUT(RANK(I)) = AIN(I); /* ARRANGE IN ORDER BY RANK */
     END;
     PUT EDIT('   AIN  RANK  AOUT')(A);
     PUT SKIP(2) EDIT((AIN(I),RANK(I),AOUT(I) DO I = 1 TO N))
        (COL(1),3 F(6));
  END B;
END SORTR;
```

Sample Output With 10 Elements

```
   AIN  RANK  AOUT

    37     2  -256
  -256     1    37
   300     5   126
   222     4   222
   333     6   300
   777     8   333
   126     3   655
   956    10   777
   823     9   823
   655     7   956
```

Problem 5: Table Look-up

Solution A: Sequential Search

```
SEARCHA: PROC OPTIONS(MAIN);
/* ILLUSTRATE SEQUENTIAL SEARCH OF A TABLE */
DCL 1 TABLE (32),
    2 NAME CHAR(20),
    2 EMP_NO CHAR(6);
/* LOAD TABLE ASSUMING 32 INPUT CARDS WITH NAME IN FIRST 20
  POSITIONS FOLLOWED BY EMPLOYEE NUMBER IN NEXT 6 POSITIONS */
GET EDIT(TABLE)(COL(1),A(20),A(6));
DCL A(32) CHAR(20) DEFINED NAME, B(32) CHAR(6) DEFINED EMP_NO;
DCL ARG CHAR(20), FUNC_VALUE CHAR(6);
/* IF TWO TABLES, A AND B ARE DEFINED APPROPRIATELY AND ARG AND
  FUNC_VALUE ARE ALSO DEFINED APPROPRIATELY THEN THE FOLLOWING
  GENERAL CODE CAN BE USED */
DCL MOREDATA BIT(1) INIT('1'B);
ON ENDFILE(SYSIN) MOREDATA = '0'B;
GET LIST(ARG);
DO WHILE(MOREDATA);
   SCANTABLE:  DO K = 1 TO 32;
     IF A(K) = ARG
       THEN DO;
             FUNC_VALUE = B(K);
             PUT SKIP DATA(K,ARG,FUNC_VALUE);
             LEAVE SCANTABLE;
       END;
   END SCANTABLE;
IF K = 33 THEN PUT SKIP LIST('SEARCH ARGUMENT NOT FOUND: ',ARG);
GET LIST(ARG);
END; /* END OF THE DOWHILE */
END SEARCHA;
```

Solution B: Binary Search

```
SEARCHB: PROC OPTIONS(MAIN);
/* ILLUSTRATE BINARY SEARCH OF A SORTED TABLE */
DCL 1 TABLE (32),
    2 NAME CHAR(20),
    2 EMP_NO CHAR(6);
/* LOAD TABLE ASSUMING 32 INPUT CARDS WITH NAME IN FIRST 20
  POSITIONS FOLLOWED BY EMPLOYEE NUMBER IN NEXT 6 POSITIONS */
GET EDIT(TABLE)(COL(1),A(20),A(6));
DCL A(32) CHAR(20) DEFINED NAME, B(32) CHAR(6) DEFINED EMP_NO;
DCL ARG CHAR(20), FUNC_VALUE CHAR(6);
/* IF TWO TABLES, A AND B ARE DEFINED APPROPRIATELY AND ARG AND
  FUNC_VALUE ARE ALSO DEFINED APPROPRIATELY THEN THE FOLLOWING
  GENERAL CODE CAN BE USED */
DCL MOREDATA BIT(1) INIT('1'B);
ON ENDFILE(SYSIN) MOREDATA = '0'B;
GET LIST(ARG);
DO WHILE(MOREDATA);
  NLO = 1; NHI = 32;  /* INITIALIZE FOR FULL TABLE */
    DO WHILE (NHI > NLO);
      NMID = (NLO + NHI)/2;
      IF ARG > A(NMID)      /* DETERMINE WHICH HALF OF TABLE */
        THEN NLO = NMID + 1;/* ARGUMENT IS IN AND RESET      */
        ELSE NHI = NMID;    /* THE PROPER LIMIT              */
    END;
    IF A(NLO) = ARG
      THEN DO;
             FUNC_VALUE = B(NLO);
             PUT SKIP DATA (NLO, ARG, FUNC_VALUE);
      END;
      ELSE PUT SKIP LIST('SEARCH ARGUMENT NOT FOUND: ',ARG);
GET LIST(ARG);
END; /* END OF THE DOWHILE */
END SEARCHB;
```

Problem 6. Concatenation of String Elements

```
BLDSTRING: PROC OPTIONS(MAIN);
/*BUILD A VARIABLE LENGTH STRING SEPARATING THE ELEMENTS BY A COMMA */
GET LIST (N);  /* OBTAIN THE NUMBER OF ELEMENTS */
  B: BEGIN;
      DCL A(N) CHAR(20) VARYING;
      GET LIST (A);
      DCL STRING CHAR(9999) VARYING;
      STRING = A(1);
      DO K = 2 TO N;
          STRING = STRING || ',' || A(K);
      END;
      PUT LIST (STRING);  /* EXHIBIT THE RESULT */
  END B;
END BLKSTRING;
```

Problem 7. Square Root

```
SQROOT: PROC OPTIONS(MAIN);
/* OBTAIN SQUARE ROOT BY ITERATION */
DCL MOREDATA BIT (1) INIT ('1'B);
ON ENDFILE (SYSIN) MOREDATA = '0'B;
GET LIST (ARG);
DO WHILE (MOREDATA);
   Y = ARG/2;                  /* INITIALIZE Y */
   DO UNTIL (YDIF < .001);  /* TEST AT END OF LOOP */
        YNEW = (Y + ARG/Y) /2;
        YDIF = Y - YNEW;
        IF YDIF < 0 THEN YDIF = - YDIF;
        Y = YNEW;
   END;
   PUT EDIT('THE SQUARE ROOT OF ',ARG,' IS ',Y)
        (COL(1), A, F(7,3), A, F(6,3));
   GET LIST (ARG);
END;
END SQROOT;
```

Sample Output

```
THE SQUARE ROOT OF   1.000 IS  1.000
THE SQUARE ROOT OF   2.000 IS  1.414
THE SQUARE ROOT OF   3.000 IS  1.732
THE SQUARE ROOT OF   4.000 IS  2.000
THE SQUARE ROOT OF   5.000 IS  2.236
THE SQUARE ROOT OF   6.000 IS  2.449
THE SQUARE ROOT OF   6.250 IS  2.500
THE SQUARE ROOT OF  17.000 IS  4.123
THE SQUARE ROOT OF 100.000 IS 10.000
THE SQUARE ROOT OF 200.000 IS 14.142
```

Problem 8. Polynomial

```
POLY: PROC OPTIONS(MAIN);
/* EVALUATE THE POLYNOMIAL A(0)+A(1)*X+A(2)*X**2+ . . . +A(N)*X**N  */
GET LIST(N);
  B: BEGIN;
    DCL A(0:N);
    GET LIST(A) COPY;
    DO X = 1 TO 30;
        F = A(N);
        DO K = N-1 TO 0 BY -1;
             F = F*X + A(K);
        END;
        PUT EDIT('X =',x,'  F(X) = ',F)(COL(1),A,F(3),A,F(10,5));
    END;
  END B;
ENDPOLY;
```

Sample Output

```
1 -.1 .01 -.001 .0001
x =   1   F(x) =   0.90910
x =   2   F(x) =   0.83360
x =   3   F(x) =   0.77110
x =   4   F(x) =   0.72160
x =   5   F(x) =   0.68750
x =   6   F(x) =   0.67360
x =   7   F(x) =   0.68710
x =   8   F(x) =   0.73760
x =   9   F(x) =   0.83710
x =  10   F(x) =   1.00000
x =  11   F(x) =   1.24310
x =  12   F(x) =   1.58560
x =  13   F(x) =   2.04910
x =  14   F(x) =   2.65760
x =  15   F(x) =   3.43750
x =  16   F(x) =   4.41760
x =  17   F(x) =   5.62910
x =  18   F(x) =   7.10560
x =  19   F(x) =   8.88310
x =  20   F(x) =  11.00000
x =  21   F(x) =  13.49710
x =  22   F(x) =  16.41759
x =  23   F(x) =  19.80708
x =  24   F(x) =  23.71358
x =  25   F(x) =  28.18745
x =  26   F(x) =  33.28157
x =  27   F(x) =  39.05106
x =  28   F(x) =  45.55356
x =  29   F(x) =  52.84903
x =  30   F(x) =  60.99991
```

Notice that the answers are accurate to about 6 significant digits. The answer for x = 30, for example, should be 61. Greater accuracy can be achieved by declaring the identifiers X, A, and F as DECIMAL FLOAT (15).

Problem 9. Series Expansion

```
 1   SERIES: PROC OPTIONS(MAIN);
     /* EVALUATE SOME MATHEMATICAL FUNCTIONS BY SERIES EXPANSION */
     /* THEN COMPARE THE RESULTS WITH THE BUILTIN FUNCTIONS      */
 2   DEFAULT RANGE (*) FLOAT DEC VALUE (FLOAT DEC(15));
 3   DCL J FIXED BIN;
 4   PUT EDIT('EXP(X)','SIN(X)','COS(X)',
              'X',('SERIES      BUILTIN' DO J = 1 TO 3))
             (COL(14),A,COL(36),A,COL(58),A,COL(2),A,3(X(4),A));
 5     DO X = 0 TO 1.0 BY .1;
 6        EX, TEX = 1;           /* INITIALIZE SERIES */
 7        SINX, TSINX = X;
 8        COSX, TCOS = 1;
 9           DO K = 1 TO 10;      /* GET TEN MORE TERMS OF SERIES */
10              TEX = TEX * X / K;
11              EX - EX + TEX;
12              TWOK = K + K;
13              TSINX = -TSINX * X * X /(TWOK * (TWOK+1));
14              SINX = SINX + TSINX;
15              TCOSX = -TCOSX * X * X / ((TWOK-1) * TWOK);
16              COSX = COSX + TCOSX;
17           END;
18        PUT SKIP EDIT(X,EX,EXP(X),SINX,SIN(X),COSX,COS(X))
                       (F(3,1), 6 F(11,6));
19     END;
20   END SERIES;
```

Output

```
              EXP(X)                 SIN(X)                 COS(X)
 X     SERIES     BUILTIN    SERIES     BUILTIN    SERIES     BUILTIN
0.0   1.000000   1.000000   0.000000   0.000000   1.000000   1.000000
0.1   1.105171   1.105171   0.099833   0.099833   0.995004   0.995004
0.2   1.221403   1.221403   0.198669   0.198669   0.980067   0.980067
0.3   1.349859   1.349859   0.295520   0.295520   0.955336   0.955336
0.4   1.491825   1.491825   0.389418   0.389418   0.921061   0.921061
0.5   1.648721   1.648721   0.479426   0.479426   0.877583   0.877583
0.6   1.822119   1.822119   0.564642   0.564642   0.825336   0.825336
0.7   2.013753   2.013753   0.644218   0.644218   0.764842   0.764842
0.8   2.225541   2.225541   0.717356   0.717356   0.696707   0.696707
0.9   2.459603   2.459603   0.783327   0.783327   0.621610   0.621610
1.0   2.718282   2.717282   0.841471   0.841471   0.540302   0.540302
```

Note: If statement 2 of the solution were omitted the series computation for EXP(X) would only be accurate to six digits instead of the seven digits shown.

Problem 10. Integration

```
INTEGRAL: PROC OPTIONS(MAIN);
/* EVALUATE INTEGRAL OF (1+(1+X**2)DX OVER RANGE 0 TO 1            */
/* BY USING SIMPSONS RULE WITH INTERVAL .01                         */
DEFAULT RANGE (*) FLOAT DEC VALUE (FLOAT DEC(15));
QUAN = 1.5;    /* INITIALIZE TO SUM OF FIRST & LAST TERMS */
ONE = 1.0;
   DO X = .01 TO .99 BY .01;
        QUAN = QUAN + (3+ONE)/(1 + X*X);
        ONE = -ONE;   /* MAKE 3+ONE ALTERNATE: 4,2,4,2,ETC */
   END;
RESULT = QUAN * .01/3;
PUT EDIT('INTEGRAL OF 1/(1+X**2)DX FROM 0 TO 1 IS '.RESULT)
        (A, F(8,7));
END INTEGRAL;
```

Sample Output

```
INTEGRAL OF 1/(1+X**2)DX FROM 0 TO 1 IS .7853982
```

Problem 11. Prime Numbers

Solution A: Test for Divisibility

```
PRIMEA: PROC OPTIONS(MAIN);
/* GENERATE LIST OF PRIMES LESS THAN 1000 */
DEFAULT RANGE (*) FIXED BIN VALUE(FIXED BIN(31));
DCL PRIME(200);
PRIME(1) = 1;  PRIME(2) = 2; PRIME(3) = 3;
L=3;   /* INDEX FOR LAST ITEM IN PRIME LIST */
  DO K = 5 TO 999 BY 2;   /* K IS NEXT CANDIDATE */
        T: DO N = 2 BY 1;
             QUOT = K/PRIME(N);
             IF K - QUOT * PRIME(N) = 0 THEN LEAVE;
             IF QUOT < PRIME(N) THEN DO;
                                        L=L+1;
                                        PRIME(L)=K;
                                        LEAVE T;
                                     END;
        END;
  END;
PUT EDIT ((PRIME(J) DO J = 1 TO L))(COL(1), 16 F(4));
END PRIMEA;
```

Output

```
   1   2   3   5   7  11  13  17  19  23  29  31  37  41  43  47
  53  59  61  77  71  73  79  83  89  97 101 103 107 109 113 127
 131 137 139 149 151 157 163 167 173 179 181 191 193 197 199 211
 223 227 229 233 239 241 251 257 263 269 271 277 281 283 293 307
 311 313 317 331 337 347 349 353 359 367 373 379 383 389 397 401
 409 419 421 431 433 439 443 449 457 461 463 467 479 487 491 499
 503 509 521 523 541 547 557 563 569 571 577 587 593 599 601 607
 613 617 619 631 641 643 647 653 659 661 673 677 683 691 701 709
 719 727 733 739 743 751 757 761 769 773 787 797 809 811 821 823
 827 829 839 853 857 859 863 877 881 883 887 907 911 919 929 937
 941 947 953 967 971 977 983 991 997
 1   PRIMEB: PROC OPTIONS(MAIN);
     /* GENERATE LIST OF PRIME NUMBERS LESS THAN 1000 BY SIFTING */
 2   DCL SIEVE(1000) BIT(1) INIT ((1000)(1)'1'B) ALIGNED;
 3      DO N = 2 TO 31;
 4           IF SIEVE(N) = '1'B THEN
                  DO K = N*N TO 1000 BY N;
 5                     SIEVE(K) = '0'B;
 6                END;
 7      END;
 8   OPEN FILE (SYSPRINT) LINESIZE (64);
 9   PUT EDIT(( K DO K = 1 TO 1000))(F(4*SIEVE(K)));
     /* THE FORMAT ITEM SUPPRESSES PRINTING WHEN SIEVE(K) = '0'B    */
10   END PRIMEB;
```

The ALIGNED attribute in statement 2 is quite important. Each bit will be put at the beginning of a separate byte. Without the ALIGNED attribute 8 bits will be put into a byte. This will save storage, but accessing the individual bits will be relatively time consuming and cause the execution time of the program to increase about 40%.

Problem 12. Perfect Numbers

```
 1   PERFECT: PROC OPTIONS(MAIN);
     /* OBTAIN AND LIST ALL PERFECT NUMBERS LESS THAN 100 */
 2   DEFAULT RANGE(*) FIXED BIN VALUE (FIXED BIN(31));
 3   DCL DIVISOR(50);
 4   DIVISOR(1)=1;
 5      DO N = 4 TO 100 BY 2;  /* ALL PERFECT NUMBERS ARE EVEN */
 6           L, SUM = 1;
 7           DO D = 2 BY 1 UNTIL (QUOT <= D + 1);
 8              QUOT = N/D;
 9              IF N - QUOT*D = 0
                   THEN DO;
10                    DIVISOR(L+1) = D;
11                    DIVISOR(L+2) = QUOT;
12                    IF QUOT > D THEN L = L+2;
13                                ELSE L = L+1;
14                 END;
15           END;
16           DO K = 2 TO L;
17              SUM = SUM + DIVISOR(K);
18           END;
19           IF SUM = N THEN PUT SKIP(2) EDIT(N,' IS A PERFECT NUMBER')
                                                 (F(3),A)
                                ('ITS DIVISORS ARE',(DIVISOR(K) DO K = 1 TO L))
                                (SKIP, A, (L) F(4));
20      END;
21   END PERFECT;
```

```
6 IS A PERFECT NUMBER
ITS DIVISORS ARE  1  2  3

28 IS A PERFECT NUMBER
ITS DIVISORS ARE  1  2  14  4  7
```

13. Factorial and Binomial Functions.

Solution A: Factorial Function

```
FCTRL: PROC(N) RETURNS (FLOAT DEC(15));
DCL (M,N,VALUE) FLOAT DEC(15);
VALUE = 1;
   DO M = 2 TO N BY 1;
        VALUE = VALUE * M;
   END;
RETURN (VALUE);
END FCTRL;
```

Solution B: Binomial Function

```
BINOM: PROC (N,K) RETURNS (FLOAT DEC(15));
DCL ( N, K ) FLOAT DEC (15);
RETURN (FCTRL(N)/(FCTRL(N-K) * FCTRL(K)) );
END BINOM;
```

Solution C: Test Program

```
TEST: PROC OPTIONS(MAIN);
DEFAULT RANGE(*) FLOAT DEC VALUE(FLOAT DEC(15));
   DO N = 0 TO 10;
   PUT SKIP;
        DO K = 0 TO N;
             PUT EDIT (BINOM(N,K))(F(5));
        END;
   END;
END TEST;
```

Expected Output

```
    1
    1    1
    1    2    1
    1    3    3    1
    1    4    6    4    1
    1    5   10   10    5    1
    1    6   15   20   15    6    1
    1    7   21   35   35   21    7    1
    1    8   28   56   70   56   28    8    1
    1    9   36   84  126  126   84   36    9    1
    1   10   45  120  210  252  210  120   45   10    1
```

Problem 14. String Manipulation

Solution A:

```
STRNGA: PROC OPTIONS(MAIN);
/* SEPARATE FIELDS SEPARATED BY AN ASTERISK */
DCL ARG CHAR(100);
GET LIST(ARG) COPY;
L = 1;      /* SET LOCATOR */
   DO WHILE (L < 100);
        K = INDEX (SUBSTR(ARG,L),'*');
        IF K = 0 THEN K = 102 - L;
        PUT SKIP LIST (SUBSTR(ARG,L,K-1));
        L = L + K;
   END;
END STRNGA;
```

Sample Output

```
'PL/I*FORTRAN*COBOL**ALGOL*APL*BASIC*ASSEMBLER'

PL/I
FORTRAN
COBOL

ALGOL
APL
BASIC
ASSEMBLER
```

Solution B:

```
STRNGB: PROC OPTIONS(MAIN);
/* REMOVE BLANKS FROM A VARYING LENGTH CHARACTER STRING */
DCL (ARG, RESULT) CHAR (100) VARYING;
GET LIST (ARG) COPY;
LIM = LENGTH (ARG);
L = 1;        /* SET LOCATOR */
   DO WHILE (L < LIM);
        K = INDEX(SUBSTR(ARG,L),' ');
        IF K=0 THEN K=LIM+2-L;
        RESULT = RESULT || SUBSTR(ARG,L,K-1);
        L = L + K;
   END;
PUT SKIP LIST(RESULT);
END STRNGB;
```

Sample Output

```
'A  B    CDE  FG   HIJK   LM  NO    PQ  RS  T  U   V  WXYZ'

ABCDEFGHIJKLMNOPQRSTUVWXYZ
```

Solution C:

```
 1   STRNGC: PROC OPTIONS(MAIN);
     /* FIND THE LONGEST STRING OF ONES IN A BIT STRING */
 2   DCL ARG BIT(100) VARYING;
 3   GET LIST(ARG) COPY;
 4   MAXLENGTH = 0;
 5   K = INDEX(ARG,'1'B);
 6   IF K = 0
        THEN PUT SKIP LIST ('NO ONES IN STRING');
 7      ELSE DO;
 8         L = K + 1;
 9         LIM = LENGTH(ARG);
10         DO WHILE (L < LIM);
11            KZ = INDEX(SUBSTR(ARG,L),'0'B);
12            IF KZ > MAXLENGTH
                 THEN DO;
13                  MAXLENGTH = KZ;
14                  KFIRST = L - 1;
15                  KLAST = KFIRST + KZ - 1;
16               END;
17            L = L + KZ;
18            K = INDEX(SUBSTR(ARG,L),'1'B);
19            IF K = 0 THEN L = LIM;
20                     ELSE L = L + K;
21         END;
22         PUT SKIP DATA(KFIRST, KLAST, MAXLENGTH);
23      END;
24   END STRNGC;
```

Sample Output

```
             '001111010101001110000011111110011010010110'B
KFIRST=        23          KLAST=        29          MAXLENGTH=        7;
```

Solution D:

```
 1  STRNGD: PROC (ARG, PAT, COUNT);
    /* SCAN ARG FOR # OF OCCURRENCES OF PAT VALUE, COUNT THEM */
 2  DCL ARG CHAR(400) VARYING, PAT CHAR(20) VARYING;
 3  DCL COUNT FIXED BIN;
 4  LPAT = LENGTH(PAT);
 5  LIM = LENGTH(ARG) + 1 - LPAT;
 6  L = 1;                      /* INITIALIZE LOCATOR */
 7     DO WHILE ( L <= LIM );
 8          K = INDEX (SUBSTR(ARG,L), PAT);
 9          IF K = 0 THEN L = LIM + 1;
10                   ELSE DO;
11                        COUNT = COUNT + 1;
12                        L = L + K - 1 + LPAT;
13                        END;
14     END;
15  END STRNGD;
```

Test Program

```
 1  TESTD: PROC OPTIONS(MAIN);
 2  DCL TESTARG CHAR(100) VARYING, TESTPAT CHAR(2) INIT('IE');
 3  GET LIST(TESTARG) COPY;
 4  DCL STRNGD ENTRY(CHAR(400) VARYING, CHAR(20) VARYING, FIXED BIN);
 5  CALL STRNGD(TESTARG, TESTPAT, KOUNT);
 6  PUT SKIP EDIT('NUMBER OF OCCURRENCES OF ',TESTPAT,' IS',KOUNT)
            (A, A, A, F(4));
 7  END TESTD;
```

Sample Output

```
'SIEVE,BELIEVE,CEILING,TRIEZE,IEIEEIEEEIE'

NUMBER OF OCCURRENCES OF IE IS  7
```

Note that in the test program, TESTD, that the first 2 arguments in statement 5 do not match the parameters indicated in statement 4. The PL/I compiler will create appropriate dummy arguments that do match. Without statement 4 erroneous results would occur when arguments do not match the parameters in the subroutine.

Problem 15. Brand Names

```
BRAND: PROC OPTIONS(MAIN);
DCL (PREFIX, ROOT, SUFFIX)(5) CHAR (6) VARYING;
GET LIST (PREFIX, ROOT, SUFFIX) COPY;
   DO I = 1 TO 5;
        DO J = 1 TO 5;
             PUT SKIP;
             DO K = 1 TO 5;
             PUT EDIT (PREFIX(I)||ROOT(J)||SUFFIX(K))(X(2),A);
             END;
        END;
   END;
END BRAND;
```

Sample Output

'SUPER' 'MAXI' 'BEAU' 'GRAND'' 'MINI'
'DRIVO' 'BUICK' 'DUPER' 'TURBO' 'ZAPPER'
'MATIC' 'WHIZZ' 'JOGGER' 'REVER' 'OOGY'

```
SUPERDRIVOMATIC SUPERDRIVOWHIZZ SUPERDRIVOJOGGER SUPERDRIVOREVER SUPERDRIVOOOGY
SUPERBUICKMATIC SUPERBUICKWHIZZ SUPERBUICKJOGGER SUPERBUICKREVER SUPERBUICKOOGY
SUPERDUPERMATIC SUPERDUPERWHIZZ SUPERDUPERJOGGER SUPERDUPERREVER SUPERDUPEROOGY
SUPERTURBOMATIC SUPERTURBOWHIZZ SUPERTURBOJOGGER SUPERTURBOREVER SUPERTURBOOOGY
SUPERZAPPERMATIC SUPERZAPPERWHIZZ SUPERZAPPERJOGGER SUPERZAPPERREVER SUPERZAPPEROOGY
MAXIDRIVOMATIC MAXIDRIVOWHIZZ MAXIDRIVOJOGGER MAXIDRIVOREVER MAXIDRIVOOOGY
MAXIBUICKMATIC MAXIBUICKWHIZZ MAXIBUICKJOGGER MAXIBUICKREVER MAXIBUICKOOGY
MAXIDUPERMATIC MAXIDUPERWHIZZ MAXIDUPERJOGGER MAXIDUPERREVER MAXIDUPEROOGY
MAXITURBOMATIC MAXITURBOWHIZZ MAXITURBOJOGGER MAXITURBOREVER MAXITURBOOOGY
MAXIZAPPERMATIC MAXIZAPPERWHIZZ MAXIZAPPERJOGGER MAXIZAPPERREVER MAXIZAPPEROOGY
BEAUDRIVOMATIC BEAUDRIVOWHIZZ BEAUDRIVOJOGGER BEAUDRIVOREVER BEAUDRIVOOOGY
BEAUBUICKMATIC BEAUBUICKWHIZZ BEAUBUICKJOGGER BEAUBUICKREVER BEAUBUICKOOGY
BEAUDUPERMATIC BEAUDUPERWHIZZ BEAUDUPERJOGGER BEAUDUPERREVER BEAUDUPEROOGY
BEAUTURBOMATIC BEAUTURBOWHIZZ BEAUTURBOJOGGER BEAUTURBOREVER BEAUTURBOOOGY
BEAUZAPPERMATIC BEAUZAPPERWHIZZ BEAUZAPPERJOGGER BEAUZAPPERREVER BEAUZAPPEROOGY
GRANDDRIVOMATIC GRANDDRIVOWHIZZ GRANDDRIVOJOGGER GRANDDRIVOREVER GRANDDRIVOOOGY
GRANDBUICKMATIC GRANDBUICKWHIZZ GRANDBUICKJOGGER GRANDBUICKREVER GRANDBUICKOOGY
GRANDDUPERMATIC GRANDDUPERWHIZZ GRANDDUPERJOGGER GRANDDUPERREVER GRANDDUPEROOGY
GRANDTURBOMATIC GRANDTURBOWHIZZ GRANDTURBOJOGGER GRANDTURBOREVER GRANDTURBOOOGY
GRANDZAPPERMATIC GRANDZAPPERWHIZZ GRANDZAPPERJOGGER GRANDZAPPERREVER GRANDZAPPEROOGY
MINIDRIVOMATIC MINIDRIVOWHIZZ MINIDRIVOJOGGER MINIDRIVOREVER MINIDRIVOOOGY
MINIBUICKMATIC MINIBUICKWHIZZ MINIBUICKJOGGER MINIBUICKREVER MINIBUICKOOGY
MINIDUPERMATIC MINIDUPERWHIZZ MINIDUPERJOGGER MINIDUPERREVER MINIDUPEROOGY
MINITURBOMATIC MINITURBOWHIZZ MINITURBOJOGGER MINITURBOREVER MINITURBOOOGY
MINIZAPPERMATIC MINIZAPPERWHIZZ MINIZAPPERJOGGER MINIZAPPERREVER MINIZAPPEROOGY
```

A slightly more efficient solution:

```
BRAND: PROC OPTIONS(MAIN);
DCL (PREFIX, ROOT, SUFFIC)(5) CHAR (6) VARYING;
GET LIST (PREFIX, ROOT, SUFFIX) COPY;
             PUT EDIT((((PREFIX(I) || ROOT(J) || SUFFIX(K)
                      DO K=1 TO 5) DO J=1 TO 5) DO I=1 TO 5))
                      (SKIP, 5(X(2),A));
END BRAND;
```

Problem 16. Combination of Coins

```
COINS: PROC OPTIONS(MAIN);
/* DETERMINE NUMBER OF WAYS TO MAKE CHANGE FOR A DOLLAR */
J=0;  /* SET INITIAL COUNT TO ZERO */
A: DO N50= 0 TO 2;          /* N50 IS NUMBER OF HALVES */
     DO N25 = 0 TO 4 - 2 * N50;  /* N25 IS THE NUMBER OF QUARTERS */
       DO N10 = 0 TO 10 - 5 * N50 - CEIL(2.5*N25);
         DO N5 = 0 TO 20 - 10 * N50 - 5 * N25 - 2 * N10;
         /* THERE IS NOW A WAY TO MAKE CHANGE FILLING IN
            BALANCE WITH PENNIES SO ADD 1 TO COUNTER */
           J = J + 1;
END A;
PUT LIST(J);
END COINS;
```

Problem 17. Calendar

```
 1  CALENDAR: PROC OPTIONS(MAIN);
 2  DEFAULT RANGE(*) FIXED BIN;
 3  DCL SYSIN FILE RECORD INPUT
       ENV (F RECSIZE(80) BLKSIZE(80) TOTAL);
 4  DCL SYSPRINT FILE RECORD OUTPUT
       ENV (FB RECSIZE(86) BLKSIZE(860) TOTAL CTLASA);
 5  DCL 1 RECIN, 2 KYR PIC '9999', 2 FILLER PIC '(76)X';
 6  DCL MOHEADING CHAR(86) INIT('1'),
       PMONTH CHAR(9) DEFINED MOHEADING POS(34),
       PYEAR CHAR(4) DEFINED MOHEADING POS(51);
 7  DCL DAYHEADING CHAR(86) INIT('0     SUN         MON         TUE
    WED         THUR        FRI         SAT');
 8  DCL IDASHES CHAR(86) INIT((' |'||(7)'-----------|')),
            CTLIDASHES CHAR(1) DEFINED IDASHES POS(1);
 9  DCL IBLANKS CHAR(86) INIT((' |'||(7)'           |'));
10  DCL IDAYNO CHAR(86);
11  READ FILE(SYSIN) INTO(RECIN);
12  PYEAR,COMPYR=KYR;
13  DCL NDAY(12) INIT(31,28,31,30,31,30,31,31,30,31,30,31);
14  IF MOD(COMPYR,4)=0 THEN NDAY(2)=29;
15  K=COMPYR-1901; NZ=MOD(DIVIDE(K,4,31,0)+K+2,7);
    /* NZ IS NOW THE DAY OF THE WEEK FOR JAN 1 COUNTING FROM SUNDAY=0 */
17  DO MO=1 TO 12;
18  PMONTH=SUBSTR('JANUARY  FEBRUARY MARCH    APRIL    MAY      JUNE     JU
    LY      AUGUST   SEPTEMBER OCTOBER  NOVEMBER DECEMBER ',9*MO-8,9);
19  WRITE FILE(SYSPRINT)FROM(MOHEADING);
20  WRITE FILE(SYSPRINT) FROM (DAYHEADING);
21  CTLIDASHES='0';
22  WRITE FILE(SYSPRINT) FROM (IDASHES);
23  CTLIDASHES=' ';
24  DAYLIM=NDAY(MO);
25  IDAYNO=IBLANKS;
26  K=1; KF=7-NZ;
28  D = 1;     L= 12 * NZ + 3;
30     DO UNTIL (KF>DAYLIM);
31     CALL WEEKOUT;
32     KF=KF+7; L=3;
34     END;
35  NZ=DAYLIM+7-KF;
36  IF NZ > 0
       THEN DO;
37     KF=DAYLIM;
38     IDAYNO=IBLANKS;
39     CALL WEEKOUT;
40     END;
41  END;
42  WEEKOUT: PROC;
43     DO UNTIL (K>KF);
44   SUBSTR(IDAYNO,L,2)=SUBSTR('1 2 3 4 5 6 7 8 9 1011121314151617181920212
    2232425262728293031',D,2);
45     D=D+2; L=L+12;
47     K=K+1;
48     END;
49  WRITE FILE(SYSPRINT) FROM (IDAYNO);
50     DO I=1 TO 4;
51     WRITE FILE(SYSPRINT) FROM(IBLANKS);
52     END;
53  WRITE FILE(SYSPRINT) FROM (IDASHES);
54  END;
55  END;
```

The TOTAL option in statements 3 and 4 allows inline code to be generated for I/O operations on the related files.

Problem 18. Student Performance

```
STPERF: PROC OPTIONS(MAIN);
DCL 1 RECIN,
        2 COURSE_NAME CHAR(20),
        2 STUDENT_NAME CHAR (20),
        2 ATTENDANCE (20) CHAR (1),
        2 GRADES (4) PIC 'Z99',
        2 FILLER CHAR(8);
DCL HDNGOUT CHAR (26) INIT ('1'),
        COURSE CHAR (20) DEF HDNGOUT POS (6);
DCL 1 NAMEOUT,
        2 CC CHAR (1) INIT ('-'),
        2 STUDENT_NAME_OUT CHAR (20);
DCL 1 ATTENDANCEOUT,
        2 CC CHAR (1) INIT (' '),
        2 LIT CHAR (24) INIT ('     LECTURES ATTENDED: '),
        2 NUMB_ATTENDED PIC 'Z9';
DCL STATUS CHAR (11);
DCL 1 GRADESOUT,
        2 CC CHAR (1) INIT (' '),
        2 GRADESPRINT (4) PIC 'ZZ99';
DCL 1 AVERAGEOUT,
        2 CC CHAR (1) INIT (' '),
        2 AVERAGE PIC 'ZZ99';
DCL MOREDATA BIT (1) INIT ('1'B);
DCL SYSIN FILE RECORD INPUT ENV(F RECSIZE(80) BLKSIZE(80) TOTAL);
DCL SYSPRINT FILE RECORD OUTPUT
        ENV (V BLKSIZE(36) RECSIZE(32) CTLASA TOTAL);
ON ENDFILE (SYSIN) MOREDATA = '0'B;
READ FILE (SYSIN) INTO (RECIN);
DO WHILE (MOREDATA);
   IF COURSE_NAME ¬= COURSE | L > 50
        THEN DO;
             COURSE = COURSE_NAME;
             WRITE FILE (SYSPRINT) FROM (HDNGOUT);
             L = 1;    /* INITIALIZE LINE INDICATOR */
             END;
   STUDENT_NAME_OUT = STUDENT_NAME;
   WRITE FILE (SYSPRINT) FROM (NAMEOUT);
   M, N = 0;
   DO K = 1 TO 3;
        IF ATTENDANCE(K)= '1' THEN M=M+1;
   END;
   DO K=4 TO 20;
        IF ATTENDANCE(K)='1' THEN N=N+1;
   END;
   NUMB_ATTENDED = M+N;
   WRITE FILE (SYSPRINT) FROM (ATTENDANCEOUT);
   IF N = 0
        THEN STATUS = ' DROPPED';
        ELSE IF ATTENDANCE(4)='0' | ATTENDANCE(9)='0'
              | ATTENDANCE(13)='0' | ATTENDANCE(20)='0'
             THEN STATUS = ' INCOMPLETE';
             ELSE DO;
                  AVERAGE=(SUM(GRADES) + GRADES(4) + 2.5)/5;
                  IF AVERAGE >= 65
                       THEN STATUS = ' PASSED';
                       ELSE STATUS = ' FAILED';
                  END;
   WRITE FILE (SYSPRINT) FROM (STATUS);
   L = L+5;      /* INCREMENT LINE INDICATOR BY 2 FOR SKIPPED */
                 /* LINES AND 3 MORE FOR PRINTED LINES        */
   IF STATUS = ' PASSED' | STATUS = ' FAILED'
        THEN DO;
             GRADESPRINT = GRADES;
             WRITE FILE (SYSPRINT) FROM (GRADESOUT);
             WRITE FILE (SYSPRINT) FROM (AVERAGEOUT);
             L = L+2;
        END;
   READ FILE(SYSIN) INTO (RECIN);
END;
END STPERF;
```

Sample Output

```
     ADVANCED CALCULUS



JOHNSON   ALFRED   W
      LECTURES ATTENDED: 18
INCOMPLETE



MAZYRICH  FRIEDA   J
      LECTURES ATTENDED: 20
PASSED
   80  90  75  85
   83



SMITH     JUDY     A
      LECTURES ATTENDED:  4
PASSED
  100  95  95  99
   98



JONES     BERTRAM  Q
      LECTURES ATTENDED:  3
DROPPED



HEATH     DAVID    M
      LECTURES ATTENDED: 19
FAILED
   70  65  60  50
   59
```

Problem 19. French Plurals

```
PLURAL: PROCEDURE OPTIONS(MAIN);
/* FRENCH PLURALS */
DECLARE MOT CHARACTER(10), PLMOT CHARACTER(11), FIN CHARACTER (1),
        FIN2 CHARACTER(2), MOREDATA BIT(1) INIT('1'B);
ON ENDFILE (SYSIN) MOREDATA = '0'B;
GET EDIT (MOT) (A(10));
  DO WHILE (MOREDATA);
   IF MOT ¬= (10)' '
      THEN DO;
             L = INDEX (MOT, ' ') - 1;
             IF L = -1 THEN L = 10;
             FIN = SUBSTR (MOT,L,1);
             FIN2 = SUBSTR (MOT,L-1,2);
             IF FIN='S' | FIN='X' | FIN='Z'
                THEN PLMOT = MOT;
                ELSE IF FIN2 = 'AL'
                   THEN PLMOT = SUBSTR (MOT,1,L-1) || 'UX';
                   ELSE IF FIN2 = 'EU' | FIN2 = 'AU'
                      THEN PLMOT = SUBSTR (MOT,1,L) || 'X';
                      ELSE PLMOT = SUBSTR (MOT,1,L) || 'S';
             PUT SKIP EDIT(MOT, PLMOT)(A,X(2),A);
           END;
   GET EDIT(MOT)(A(10));
  END;
END PLURAL;
```

Sample Output

```
ROSE        ROSES
PRIX        PRIX
CHATEAU     CHATEAUX
CHEVAL      CHEVAUX
MOT         MOTS
BAS         BAS
```

Problem 20. Check Writing

```
T: PROC OPTIONS(MAIN);
DCL COUT CHAR(73) VARYING;
DCL CVAL(0:19) CHAR(10) VARYING INIT('','ONE ','TWO ','THREE ','FOUR ',
   'FIVE ','SIX ','SEVEN ','EIGHT ','NINE ','TEN ','ELEVEN ','TWELVE ',
   'THIRTEEN ','FOURTEEN ','FIFTEEN ','SIXTEEN ','SEVENTEEN ',
   'EIGHTEEN ','NINETEEN '),
XENTY(2:9)CHAR(8)VARYING INIT('TWENTY ','THIRTY ','FORTY ','FIFTY ',
   'SIXTY ','SEVENTY ','EIGHTY ','NINETY ');
DCL VALUE FIXED DECIMAL(6,2);
DCL MOREDATA BIT(1) INIT('1'B);
ON ENDFILE(SYSIN)MOREDATA='0'B;
GET LIST(VALUE);
DO WHILE (MOREDATA);
   KH=VALUE/100; /* # OF HUNDREDS OF DOLLARS */
   KTC=VALUE - 100*KH;  /* # OF DOLLARS WITHOUT HUNDREDS */
   K=(VALUE-TRUNC(VALUE))*100; /* INTEGER NUMBER OF CENTS */
   IF KH > 0
        THEN IF KH > 19 | KH = 10
             THEN DO;
                  J=KH/10;  JJ=KH-10*J;
                  IF JJ=0 THEN COUT=CVAL(J)||'THOUSAND ';
                          ELSE COUT=XENTY(J)||CVAL(JJ)||'HUNDRED ';
                  END;
             ELSE COUT=CVAL(KH)||'HUNDRED ';
        ELSE COUT='';
   IF KTC > 0
        THEN IF KTC > 19
             THEN DO;
                  J=KTC/10;  JJ=KTC-10*J;
                  COUT=COUT||XENTY(J)||CVAL(JJ);
                  END;
             ELSE COUT=COUT||CVAL(KTC);
   IF VALUE > 1.99
        THEN COUT=COUT||'DOLLARS';
        ELSE IF VALUE > .99 THEN COUT=COUT||'DOLLAR';
   IF K > 0 THEN DO;
        IF VALUE > .99 THEN COUT=COUT||' AND ';
        IF K > 19
             THEN DO;
                  J=K/10;  JJ=K-10*J;
                  COUT=COUT||XENTY(J)||CVAL(JJ)||'CENTS';
                  END;
             ELSE IF K > 1 THEN COUT=COUT||CVAL(K)||'CENTS';
             ELSE COUT=COUT||'ONE CENT';
        END;
   L = LENGTH(COUT);  J = (73-L)/2; /* J= # OF * NEEDED AT THE LEFT */
   COUT=REPEAT('*',J-1)||COUT||REPEAT('*',72-L-J);
   PUT SKIP LIST(VALUE,COUT);
   GET LIST(VALUE);
END;    /* END OF THE DO WHILE */
END T;
```

```
   0.01          ********************************ONE CENT*********************************
   0.02          ********************************TWO CENTS********************************
   0.40          *******************************FORTY CENTS*******************************
   0.89          ****************************EIGHTY NINE CENTS****************************
   1.00          *******************************ONE DOLLAR********************************
   1.01          *************************ONE DOLLAR AND ONE CENT*************************
   1.05          ************************ONE DOLLAR AND FIVE CENTS************************
  13.00          ****************************THIRTEEN DOLLARS*****************************
  14.17          ******************FOURTEEN DOLLARS AND SEVENTEEN CENTS*******************
  15.66          *******************FIFTEEN DOLLARS AND SIXTY SIX CENTS*******************
 101.01          ******************ONE HUNDRED ONE DOLLARS AND ONE CENT*******************
1001.01          ******************ONE THOUSAND ONE DOLLARS AND ONE CENT******************
 200.00          ***************************TWO HUNDRED DOLLARS***************************
 220.10          ****************TWO HUNDRED TWENTY DOLLARS AND TEN CENTS*****************
2000.00          **************************TWO THOUSAND DOLLARS***************************
1234.56          *********TWELVE HUNDRED THIRTY FOUR DOLLARS AND FIFTY SIX CENTS**********
2345.67          ******TWENTY THREE HUNDRED FORTY FIVE DOLLARS AND SIXTY SEVEN CENTS******
3040.50          **************THREE THOUSAND FORTY DOLLARS AND FIFTY CENTS***************
7777.77          ***SEVENTY SEVEN HUNDRED SEVENTY SEVEN DOLLARS AND SEVENTY SEVEN CENTS***
```

Problem 21. Easter

```
ETABLE: PROC OPTIONS (MAIN);
/* GENERATE TABLE OF EASTER DATES FOR THE TWENTIETH CENTURY */
DEFAULT RANGE(I:N) FIXED BIN VALUE(FIXED BIN(31));
DCL (KM CHAR(5), KD, KY) (4);
PUT PAGE EDIT ('EASTER DATES')(COL(36),A);
  DO I = 1 TO 25;
     DO J = 1 TO 4;
        KY(J) = 1875 + 25*J + I;
        CALL EASTER (KY(J), M, KD(J));
        IF M = 3 THEN KM(J) = 'MARCH';
                 ELSE KM(J) = 'APRIL';
     END;
     PUT SKIP EDIT ((KM(J), KD(J), KY(J) DO J=1 TO 4))
                   (X(6), A(5), F(3), F(5));
  END;
END ETABLE;
```

```
EASTER: PROC (YEAR, MONTH, DAY);
DEFAULT RANGE(*) FIXED BIN VALUE (FIXED BIN(31));
A = MOD (YEAR,19);
B = YEAR/100;
C = YEAR - 100*B;
D = B/4;
E = B - 4*D;
F = (B+8)/25;
G = (B - F + 1)/3;
H = MOD (19*A + B - D - G + 15, 30);
I = C/4;
K = C - 4*I;
J = MOD (32 + 2*(E+I) - H - K, 7);
M = (A + 11*(H + J + J))/451;
Q = H + J - 7*M + 114;
MONTH = Q/31;
DAY = 1 + Q - MONTH*31;
END EASTER;
```

```
                                   EASTER DATES
 APRIL   7 1901      APRIL   4 1926      MARCH  25 1951      APRIL  18 1976
 MARCH  30 1902      APRIL  17 1927      APRIL  13 1952      APRIL  10 1977
 APRIL  12 1903      APRIL   8 1928      APRIL   5 1953      MARCH  26 1978
 APRIL   3 1904      MARCH  31 1929      APRIL  18 1954      APRIL  15 1979
 APRIL  23 1905      APRIL  20 1930      APRIL  10 1955      APRIL   6 1980
 APRIL  15 1906      APRIL   5 1931      APRIL   1 1956      APRIL  19 1981
 MARCH  31 1907      MARCH  27 1932      APRIL  21 1957      APRIL  11 1982
 APRIL  19 1908      APRIL  16 1933      APRIL   6 1958      APRIL   3 1983
 APRIL  11 1909      APRIL   1 1934      MARCH  29 1959      APRIL  22 1984
 MARCH  27 1910      APRIL  21 1935      APRIL  17 1960      APRIL   7 1985
 APRIL  16 1911      APRIL  12 1936      APRIL   2 1961      MARCH  30 1986
 APRIL   7 1912      MARCH  28 1937      APRIL  22 1962      APRIL  19 1987
 MARCH  23 1913      APRIL  17 1938      APRIL  14 1963      APRIL   3 1988
 APRIL  12 1914      APRIL   9 1939      MARCH  29 1964      MARCH  26 1989
 APRIL   4 1915      MARCH  24 1940      APRIL  18 1965      APRIL  15 1990
 APRIL  23 1916      APRIL  13 1941      APRIL  10 1966      MARCH  31 1991
 APRIL   8 1917      APRIL   5 1942      MARCH  26 1967      APRIL  19 1992
 MARCH  31 1918      APRIL  25 1943      APRIL  14 1968      APRIL  11 1993
 APRIL  20 1919      APRIL   9 1944      APRIL   6 1969      APRIL   3 1994
 APRIL   4 1920      APRIL   1 1945      MARCH  29 1970      APRIL  16 1995
 MARCH  27 1921      APRIL  21 1946      APRIL  11 1971      APRIL   7 1996
 APRIL  16 1922      APRIL   6 1947      APRIL   2 1972      MARCH  30 1997
 APRIL   1 1923      MARCH  28 1948      APRIL  22 1973      APRIL  12 1998
 APRIL  20 1924      APRIL  17 1949      APRIL  14 1974      APRIL   4 1999
 APRIL  12 1925      APRIL   9 1950      MARCH  30 1975      APRIL  23 2000
```

Problem 22: Chess

A: Queens

```
QUEENS: PROC OPTIONS(MAIN);
/* SOLVE THE QUEENS PROBLEM FOR SQUARE BOARDS */
DEFAULT RANGE(A:Z) FIXED BIN;
DO N= 4 TO 8;  /* LETS GET SOLUTIONS FOR SQUARE BOARDS UP TO 8 BY 8 */
BEGIN;
       DCL Q(N), P(N);
/* IN SCANNING FOR A SOLUTION Q(J)=1 IF COLUMN J HAS NOT BEEN USED. */
/* P(L) WILL BE THE POSITION OF THE QUEEN ON ROW L. */
NSOL=0;
DO I=1 TO N;
Q=1; Q(I)=0; P(1)=I;
  ROWINC: DO L=2 BY 1;
    PROC: DO J=1 BY 1;
      COLTEST: DO WHILE(Q(J)¬=0);
        DIAGTEST: DO K=1 TO L-1;
          IF ABS(J-P(K))=L-K THEN LEAVE COLTEST;
        END DIAGTEST;
        P(L)=J;
        IF L<N THEN DO;
                       Q(J) = 0;
                       LEAVE PROC;
                    END;
/* WE HAVE A SOLUTION */
IF NSOL=0 THEN PUT PAGE EDIT('FIRST SOLUTION FOR PLACING ',N,
   ' QUEENS ON ',N,' BY ',N,' SQUARE BOARD SO THAT',
   'NO QUEEN IS ON THE SAME ROW, COLUMN, OR DIAGONAL IS ILLUSTRATED')
   ((3)(A,F(2)),A,SKIP,A)
   ((('|   ' DO J=1 TO N-1),'|   |',('----' DO J=1 TO N-1),'-----',
     ('|   ' DO J=1 TO N-1),'|   |','Q' DO K=1 TO N),
     ('|   ' DO J=1 TO N-1),'|   |',('----' DO J=1 TO N-1),'-----')
   (SKIP,(N)A,SKIP(0),(N)A,SKIP,(N)A,SKIP(0),COL(4*P(K)-1),A);
/* PRINT ROW POSITIONS */  PUT SKIP EDIT(P)(F(2));
      NSOL=NSOL+1;
      J = N;
      LEAVE COLTEST;
      END COLTEST;
      IF J = N THEN
        DO UNTIL (J < N);
          L = L - 1;
          IF L = 1 THEN LEAVE ROWINC;
          J = P(L);
          Q(J) = 1;
        END;
    END PROC;
  END ROWINC;
END;
END;  /* END OF THE BEGIN BLOCK */
PUT EDIT('NUMBER OF SOLUTIONS FOR ',N,' BY ',N,' IS',NSOL)
      (SKIP(2),A,F(1),A,F(1),A,F(4));
END;  /* END OF THE OUTER DO LOOP */
END QUEENS;
```

Output

FIRST SOLUTION FOR PLACING 4 QUEENS ON 4 BY 4 SQUARE BOARD SO THAT NO QUEEN IS ON THE SAME ROW, COLUMN, OR DIAGONAL IS ILLUSTRATED

```
+---+---+---+---+
I   I Q I   I   I
+---+---+---+---+
I   I   I   I Q I
+---+---+---+---+
I Q I   I   I   I
+---+---+---+---+
I   I   I Q I   I
+---+---+---+---+
 2 4 1 3
 3 1 4 2
```

NUMBER OF SOLUTIONS FOR 4 BY 4 IS 2

FIRST SOLUTION FOR PLACING 5 QUEENS ON 5 BY 5 SQUARE BOARD SO THAT NO QUEEN IS ON THE SAME ROW, COLUMN, OR DIAGONAL IS ILLUSTRATED

```
+---+---+---+---+---+
I Q I   I   I   I   I
+---+---+---+---+---+
I   I   I Q I   I   I
+---+---+---+---+---+
I   I   I   I   I Q I
+---+---+---+---+---+
I   I Q I   I   I   I
+---+---+---+---+---+
I   I   I   I Q I   I
+---+---+---+---+---+
 1 3 5 2 4
 1 4 2 5 3
 2 4 1 3 5
 2 5 3 1 4
 3 1 4 2 5
 3 5 2 4 1
 4 1 3 5 2
 4 2 5 3 1
 5 2 4 1 3
 5 3 1 4 2
```

NUMBER OF SOLUTIONS FOR 5 BY 5 IS 10

FIRST SOLUTION FOR PLACING 6 QUEENS ON 6 BY 6 SQUARE BOARD SO THAT NO QUEEN IS ON THE SAME ROW, COLUMN, OR DIAGONAL IS ILLUSTRATED

```
+---+---+---+---+---+---+
I   I Q I   I   I   I   I
+---+---+---+---+---+---+
I   I   I   I Q I   I   I
+---+---+---+---+---+---+
I   I   I   I   I   I Q I
+---+---+---+---+---+---+
I Q I   I   I   I   I   I
+---+---+---+---+---+---+
I   I   I Q I   I   I   I
+---+---+---+---+---+---+
I   I   I   I   I Q I   I
+---+---+---+---+---+---+
 2 4 6 1 3 5
 3 6 2 5 1 4
 4 1 5 2 6 3
 5 3 1 6 4 2
```

NUMBER OF SOLUTIONS FOR 6 BY 6 IS 4

FIRST SOLUTION FOR PLACING 7 QUEENS ON 7 BY 7 SQUARE BOARD SO THAT NO QUEEN IS ON THE SAME ROW, COLUMN, OR DIAGONAL IS ILLUSTRATED

```
+---+---+---+---+---+---+---+
I Q I   I   I   I   I   I   I
+---+---+---+---+---+---+---+
I   I   I Q I   I   I   I   I
+---+---+---+---+---+---+---+
I   I   I   I   I Q I   I   I
+---+---+---+---+---+---+---+
I   I   I   I   I   I   I Q I
+---+---+---+---+---+---+---+
I   I Q I   I   I   I   I   I
+---+---+---+---+---+---+---+
I   I   I   I Q I   I 1 I   I
+---+---+---+---+---+---+---+
I   I   I   I   I   I Q I   I
+---+---+---+---+---+---+---+
 1 3 5 7 2 4 6
 1 4 7 3 6 2 5
 1 5 2 6 3 7 4
 1 6 4 2 7 5 3

 .
 . etc.
 .
```

NUMBER OF SOLUTIONS FOR 7 BY 7 IS 40

FIRST SOLUTION FOR PLACING 8 QUEENS ON 8 BY 8 SQUARE BOARD SO THAT NO QUEEN IS ON THE SAME ROW, COLUMN, OR DIAGONAL IS ILLUSTRATED

```
+---+---+---+---+---+---+---+---+
I Q I   I   I   I   I   I   I   I
+---+---+---+---+---+---+---+---+
I   I   I   I   I Q I   I   I   I
+---+---+---+---+---+---+---+---+
I   I   I   I   I   I   I   I Q I
+---+---+---+---+---+---+---+---+
I   I   I   I   I   I Q I   I   I
+---+---+---+---+---+---+---+---+
I   I   I Q I   I   I   I   I   I
+---+---+---+---+---+---+---+---+
I   I   I   I   I   I   I Q I   I
+---+---+---+---+---+---+---+---+
I   I Q I   I   I   I   I   I   I
+---+---+---+---+---+---+---+---+
I   I   I   I Q I   I   I   I   I
+---+---+---+---+---+---+---+---+
  1 5 8 6 3 7 2 4
  1 6 8 3 7 4 2 5
  1 7 4 6 8 2 5 3
  1 7 5 8 2 4 6 3
  2 4 6 8 3 1 7 5

  .
  . etc.
  .
```

NUMBER OF SOLUTIONS FOR 8 BY 8 IS 92

B: Knight's Tour

```
KNIGHT: PROC OPTIONS(MAIN);
/* KNIGHTS TOUR PROBLEM */
DCL 1 LOC(64),      /* FOR EACH SQUARE */
      2 NUMBR,      /* NUMBER OF LEGAL KNIGHT MOVES */
      2 LEGAL(8);   /* LIST OF SQUARES WHERE KNIGHT MAY MOVE */
DCL 1 LOCSAVE(64), 2 NNN, 2 LLLL(8);
DCL (BOARD(64), ROW) FIXED BIN;
DO I = 1 TO 64;            /* FILL LOCSAVE TABLE WITH LIST OF */
   NNN(I) = 0;             /* LEGAL MOVES WHEN BOARD IS EMPTY */
   ROW = (I-1)/8 + 1;
   J = ROW + ROW + I +8;      /* J IS A 2 DIGIT CODED ROW, COLUMN */
   /* K IS A POTENTIALLY LEGAL KNIGHT MOVE.  IN THE CENTRAL PART OF */
   /* THE BOARD ALL 8 POSSIBILITIES FOR K ARE LEGAL, BUT FOR MANY   */
   /* SQUARES AT OR NEAR THE EDGES THERE ARE LESS THAN 8 LEGAL MOVES*/
   DO K = J-21,J-19,J-12,J-8,J+8,J+12,J+19,J+21;
      NEWROW = K/10;
      DO;
         IF NEWROW < 1  |  NEWROW > 8 THEN LEAVE;
         NEWCOL = K - NEWROW*10;
         IF NEWCOL < 1  |  NEWCOL > 8 THEN LEAVE;
         NNN(I) = NNN(I) + 1; /* UP COUNT OF LEGAL MOVES--SQUARE I */
         /* CALCULATE SQUARE # FOR LEGAL MOVE AND PUT IN ARRAY:    */
         LLLL(I, NNN(I)) = (NEWROW-1)*8 + NEWCOL;
      END;
   END;
END;
OPEN FILE(SYSPRINT) PAGESIZE(54);  /* 3 BOARD DISPLAYS PER PAGE */
DO M = 1 TO 64;            /* TRY ALL POSSIBLE STARTING MOVES */
   LOC = LOCSAVE;          /* FETCH THE LIST OF LEGAL MOVES */
   L = M;                  /* INITIALIZE L--SQUARE NUMBER FOR MOVE K */
   BOARD(L) = 1;           /* PUT 1 IN THE STARTING POSITION */
   DO K = 2 TO 64;         /* LOOP TO GENERATE SUBSEQUENT MOVES */
      /* DELETE REFERENCES TO L FROM LEGAL ARRAYS BY COMPRESSION */
      DO JL = 1 TO NNN(L);
         J = LLLL(L,JL);
         JJLIM = NUMBR(J);
         JJLOOP: DO JJ = 1 TO JJLIM;
            IF LEGAL(J,JJ) = L
               THEN DO;
                          NUMBR(J) = NUMBR(J) - 1;
                          LEGAL(J,JJ) = LEGAL(J,JJLIM);
                          LEAVE JJLOOP;
                    END;
         END JJLOOP;
      END;
      /*MOVE TO SQUARE FROM WHICH LEAST NUMBER OF LEGAL MOVES REMAIN:*/
      N = NUMBR(L);
      IF N = 0 THEN LEAVE;
      MINPTR = LEGAL(L,1);
      MINVAL = NUMBR(MINPTR);
      DO J = 2 TO N;    /* TRY TO FIND A SMALLER NUMBER */
         LL = LEGAL(L,J);
         IF NUMBR(LL) < MINVAL
            THEN DO;
                    MINVAL = NUMBR(LL);
                    MINPTR = LL;
                 END;
      END;
      /* ZERO OUT THE NUMBER FOR OLD L AND UPDATE CURRENT LOCATION: */
      NUMBR(L) = 0;
      L = MINPTR;
      BOARD(L) = K;
   END;
   IF N > 0 THEN PUT SKIP (2) EDIT (BOARD) (SKIP(2), 8 F(3));
END;
END KNIGHT;
```

Sample Output

```
20 41 16  1 36 39 14 63
17  2 19 40 15 64 35 38
42 21 50 45 58 37 62 13
 3 18 43 48 61 52 59 34
22 49 46 51 44 57 12 53
 7  4 25 56 47 60 33 30
26 23  6  9 28 31 54 11
 5  8 27 24 55 10 29 32

55 14 43 26  1 16 63 20
42 27 56 15 62 19  2 17
13 54 51 44 25 58 21 64
28 41 34 57 52 61 18  3
35 12 53 50 45 24 59 22
40 29 38 33 60 49  4  7
11 36 31 46  9  6 23 48
30 39 10 37 32 47  8  5

14 11 30 43 16  1 34 45
29 42 15 12 57 44 17  2
10 13 56 31 50 33 46 35
41 28 51 58 55 60  3 18
24  9 40 61 32 49 36 47
27 64 25 52 59 54 19  4
 8 23 62 39  6 21 48 37
63 26  7 22 53 38  5 20
```

Shown here is the second page of the output. The solution generates 64 successful tours of which 8 are circuitous. The first tour shown here is circuitous.