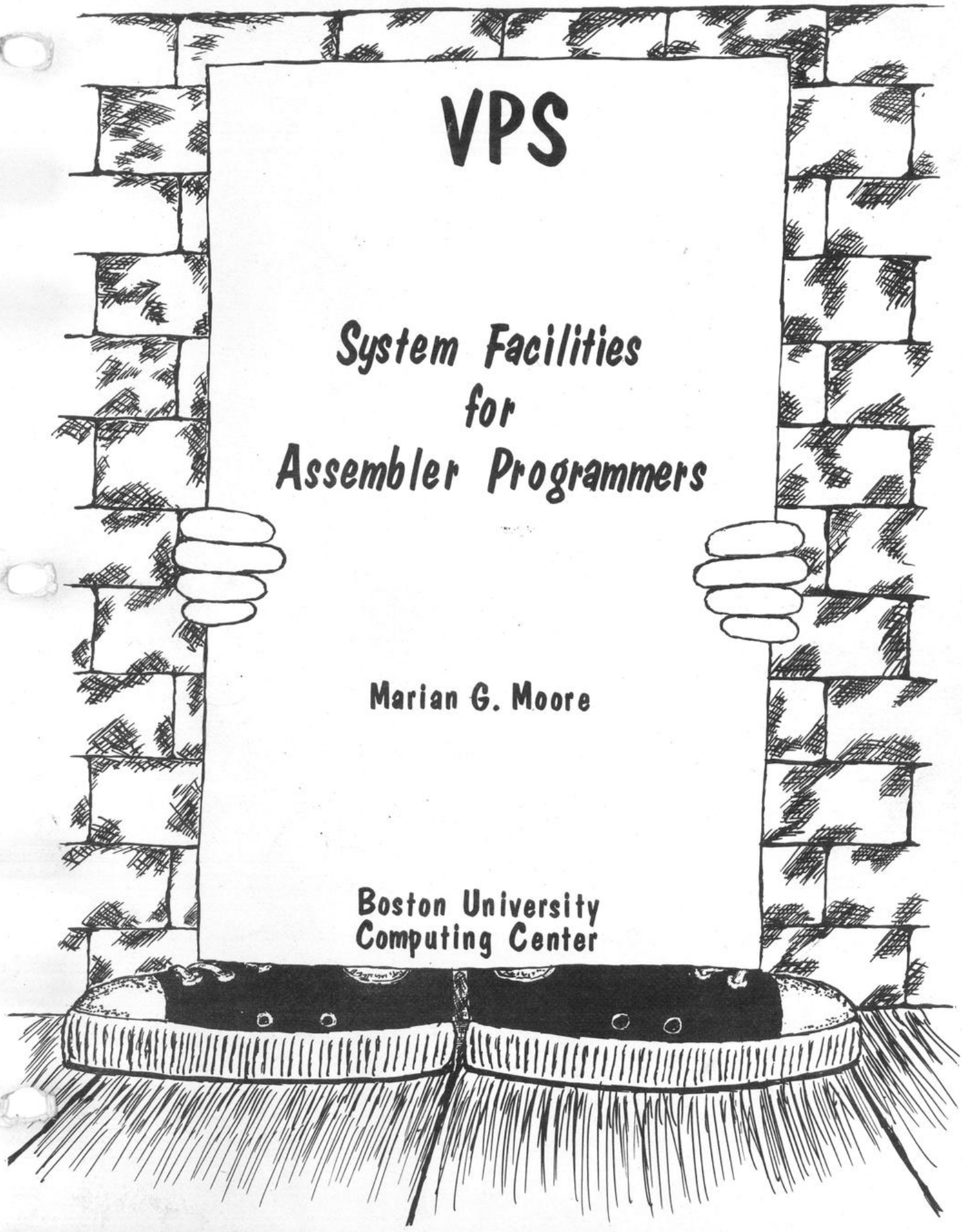


VPS

System Facilities for Assembler Programmers

Marian G. Moore

Boston University
Computing Center

Foreword

This is the first draft of a new manual scheduled for publication by the Boston University Computing Center in the Fall of 1977. Since it is directed to a more sophisticated audience and since it is the first technical document on the VPS system services, it is being made available at this time. As a reader of this draft, your commments and criticisms of the material presented and the manner of presentation are essential to the usefulness of this manual in its final form. A special tail sheet has been provided for your comments. Please send it to the Computing Center or drop it off in Room 3.

This manual was prepared in machine readable form using the VPS Editor and Script facilities. These text editing facilities are available to users of the system and are documented in detail in the VPS User's Guide.

I would like to thank John Porter who not only acted as my editor during the writing of this manual but also designed the cover. Thanks also go to Lewis Nathan who kept me honest by scrupulously comparing the documentation to the actual macros.

Marian G. Moore
Senior Systems Programmer
June, 1977

Introduction

This manual is intended mainly for assembler language programmers who wish to employ VPS system facilities in their programs. It attempts to describe the facilities available, the macro instructions which can be used to request these facilities from the control program, and related data areas.

This manual is divided into two parts. Part I, 'System Facilities', provides explanations and examples for use of the facilities provided. Part II, 'Macro Instructions', contains a description and definition of each VPS macro making it possible for assembler programmers to successfully request services from the VPS control program. The manual was arranged in this manner so that Part II, which is ordered alphabetically, could be used as a quick reference when coding programs.

At the writing of this manual the VPS system is still in an embryonic state. The structure of the macros and documented data areas will undoubtedly change as new facilities are added. The use of these macros does <u>not</u> absolutely guarantee that future enhancements to the system will not impact your programs - BUT IT IS YOUR BEST BET.

Contents

Introduction

VPS (Virtual Processor System) is a timesharing operating system designed and implemented by the systems staff of the Boston University Computing Center. The main goals of VPS are to make available a large number of resources to the timesharing user while, at the same time, provide quick reponse time and efficient use of the hardware facilities.

Many of the internal features of VPS are available to assembler language programmers through the use of macro instructions (described in detail in Part II). These 'system facilities' include: program controlled terminal attention handling, data and file manipulation, extensive message generation and sending facilities, and timing services. In the following section some of the more extensive and complex system facilities are explained and examples of their use shown.

This chapter discusses the facilities available in VPS for user attention handling.

The SETATTN Macro

Under normal operation, if the terminal user strikes the attention key during the execution of a program, a system attention interrupt occurs and the control program handles the attention. After the attention key is struck, the user has a choice of canceling the output of the program, canceling the program, skipping lines of output, etc. In certain circumstances the assembler language programmer may find it advantageous to handle attentions in the program.

The VPS macro, SETATTN, has been provided to allow assembler programmers the ability to establish an attention exit routine in their programs. After the SETATTN macro has been issued, informing the control program of the existence of an exit routine, control will be passed to that routine each time a program attention interrupt occurs during the execution of that program.

The SETATTN macro can also be used to cancel an attention exit routine, which will cause future attentions to be handled by the control program.

VPS Attention Modes

VPS supports two attention mode settings, system mode and program mode. The attention mode desired is set with the /CTL command (see the VPS User's Guide). After an attention exit routine has been established using the SETATTN macro, the exit routine will not receive control until a program attention interrupt occurs. The procedure the terminal user must follow to cause a program attention interrupt depends upon the attention mode setting.

If system mode is set - striking the attention key once causes a system attention interrupt; at this point, typing /ATTN when the keyboard unlocks will cause a program attention interrupt. A program attention interrupt can also be caused in system mode by rapidly striking the attention key twice.

If program mode is set - striking the attention key once causes a program attention interrupt. Striking it rapidly twice will cause a system attention interrupt.

The SETATTN Parameter List

The SETATTN macro requires a parameter list which may be built using the SALIST macro. The list specifies an output buffer address and its length, and an input buffer address and its length. If an output buffer address and length are coded, the contents of the buffer will be printed on the user's terminal when the attention key is struck. If the input buffer address and length are coded, when the attention key is struck, the output message will appear (if coded) and the terminal will unlock awaiting a reply from the user. If either of these options is used, the attention exit routine will not be given

control until after the message is printed and/or the reply has been placed in the buffer.

The parameter list also provides a means of passing a user data address to the exit routine. If this operand is coded, upon entry to the exit routine, register 1 will contain the address of this area.

The Attention Exit Routine

When control is passed to the attention exit routine, registers 0, 1, 13, 14, and 15 will contain the following:

- r0 - the address of the Attention Communications Area (defined by the ATTNCOMM DSECT). The area contains, among other things, the register contents when the user's program was interrupted and, at offset +72, a copy of the parameter list.
- r1 - the address of the user data area (if specified).
- r13 - the save area address.
- r14 - the return address.
- r15 - the address of the attention exit routine.

The attention exit routine should save its registers in the save area provided in register 13 upon entry and restore them upon exit. If the attention routine exits to the address in register 14, the control program will pass control to the next instruction to be executed in the main program after the attention interrupt was taken.

On entry to the attention exit routine the control program will automatically suppress all program attentions until the exit routine has returned control. If the programmer wishes to allow attentions during the execution of the exit routine, the SETATTN macro with the SOFF operand may be issued upon entry to turn off attention suppression. This will cause re-entry to the exit routine if a program attention interrupt occurs during the execution of the exit routine.

Example

In the following example, an attention exit routine is established for the first phase of a two phase program. If the user causes a program attention interrupt during phase 1, the following message will appear:

```
type: phase2 or cancel
```

If 'cancel' is entered, the program will be terminated using the ABEOJ macro. If 'phase2' is entered, a switch is set and control is returned to the main program - the next test of switch in the main program will cause phase2 to be entered. If anything else is entered, control will be returned to the main program and phase 1 will continue.

The exit routine allows attentions to be processed during its execution by issuing the SETATTN macro with the SOFF operand. When phase 2 is entered, the attention exit is canceled.

```
          .
          .
          .
          setattn list=alist,exit=attnexit,save=savearea
phase1    ds    0h
          tm    switch,x'01'          should phase 2 be entered?
          bo    phase2                yes, branch
          .
          .
          .
          b     phase1
phase2    ds    0h
          setattn cancel              cancel exit routine
          .
          .
          .
          using attnexit,r15          addressability for
attnexit  ds    0h                    attention exit routine
          save  (14,12)
          setattn soff                allow attentions
          la    r2,4(,r1)             input buffer address
          clc   0(6,r2),=c'cancel'    request to cancel?
          be    cancel                yes, branch
          clc   0(6,r2),=c'phase2'    request for phase 2?
          bne   *+8                   no, branch
          oi    0(r1),x'01'           yes, set switch
          return (14,12)              return to main program
cancel    ds    0h
          abeoj abmsg                 cancel program
abmsg     abcode u200,' program canceled in phase 1'
          .
          .
          .
savearea  ds    18f
alist     salist obuf=(output,23),ibuf=(input,6),usaddr=exitlist
output    dc    c123' type: phase2 or cancel'
exitlist  ds    0f
switch    dc    x'00'
          ds    0f
input     ds    c16
          .
          .
          .
```

This chapter discusses the data management services available in VPS as well as the physical and logical characteristics of the storage media used at Boston University.

Introduction

The data management programs of the VPS operating system allow the assembler language programmer the ability to efficiently transfer data between the program and direct-access volumes (disks), tape volumes, the user terminal (for programs being run at terminals), and unit record devices - card readers, card punches, and printers (for programs being run on the batch). Using VPS, the programmer may organize and access files in two ways:

- Sequential. Records are placed in physical sequence. Given one record, the location of the next record accessed is determined by its physical position in the file. Sequential organization is used for tape files, unit record devices, terminal input and output, and may be selected for direct-access devices.
- Direct. The records within the file, which must be on a direct-access device, may be accessed in any order. Records may be arranged in sequence according to a key.

The data management programs of VPS comprise two "access methods" - QIO (Queued Input/Output) and DIO (Direct Input/Output). QIO can be used to access sequential files, read direct files either sequentially or randomly by record number, and write direct files. QIO allows device independence when processing sequential files. DIO can be used to read and write direct files and to search direct files by key.

File Record Formats

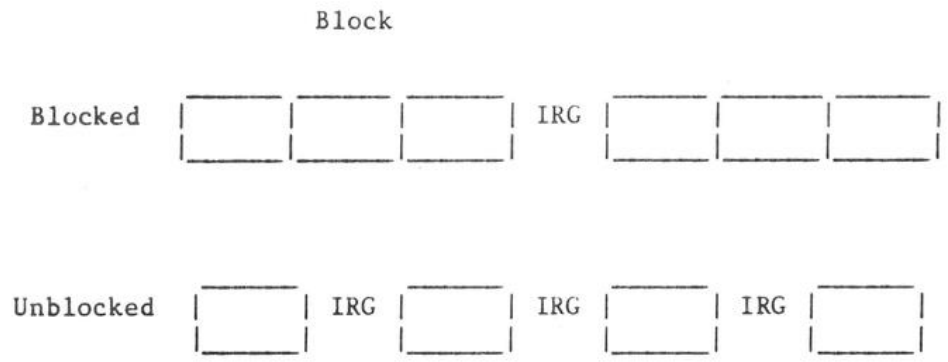
A file is composed of a collection of records that normally have some logical relationship to one another. The record is the basic unit of information used by the processing program.

The process of grouping a number of records before writing them on a volume is called 'blocking'. Each block consists of one or more records. Blocking conserves space on direct-access devices and tape volumes because it reduces the number of interrecord gaps (IRGs) in the file. (IRGs are written by the hardware between adjacent blocks of a disk or tape file.) It can also increase the efficiency of reading or writing a file by reducing the number of input/output operations required to process it.

Records may exist in one of three formats: fixed-length (format F), variable-length (format V), and, for tape only, undefined (format U). The record format of the file is specified on the /FILE statement.

Fixed-Length Records

The size of fixed-length (format F) records is constant for all records in the file. The number of records within a block is constant for every block in the data set, unless the file contains truncated (short) blocks. If the file contains unblocked format F records, one record constitutes one block.

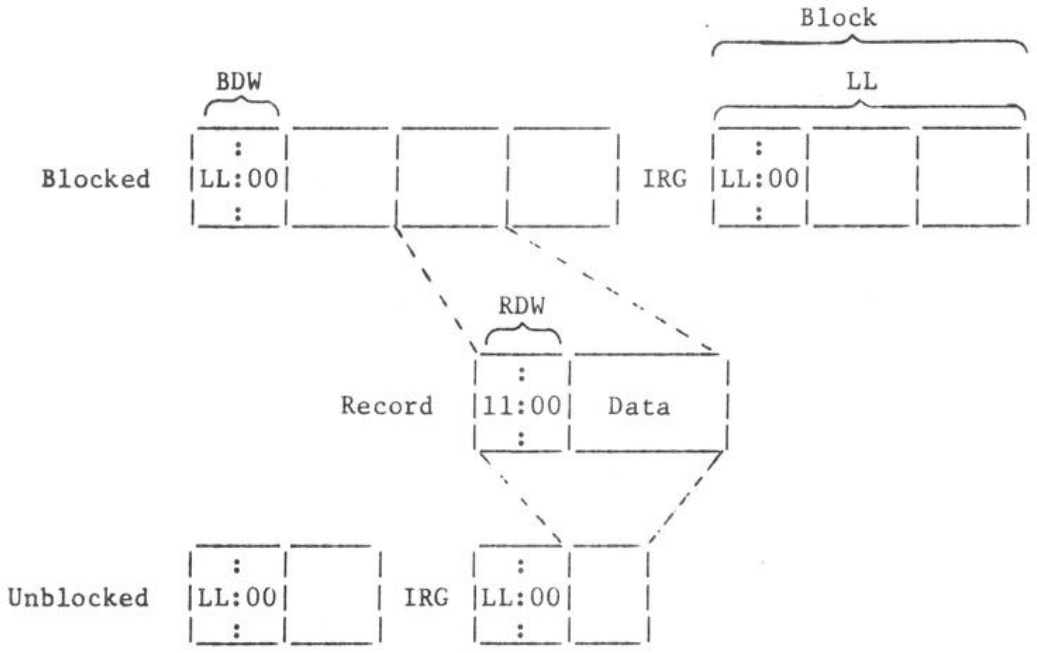


Variable-Length Records

The variable-length format (format V) provides for variable length records and variable length blocks. The first 4 bytes of each record and each block make up a descriptor word containing control information. Assembler programmers must allow for these additional 4 bytes in both input and output operations in programs.

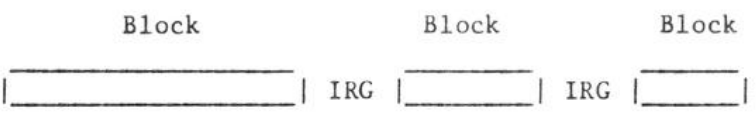
- Block Descriptor Word. A variable length block consists of a Block Descriptor Word (BDW) followed by 1 or more logical records. The first 2 bytes specify the block length (4, for the BDW, plus the total length of all the records within the block) followed by 2 bytes of zeroes. This length can be from 8 to 32,760 bytes. QIO will automatically provide the BDW when the file is written.

- Record Descriptor Word. A variable length record consists of a Record Descriptor Word (RDW) followed by data. The Record Descriptor Word is a 4-byte field describing the record. The first 2 bytes contain the length of the record including the RDW length of 4. The second 2 bytes are zeroes. The length can be from 4 to 32,756. The programmer must provide the RDW when supplying V-format records to QIO.



Undefined Records

Format U permits processing of records that do not conform to the F or V formats for tape files only. Each block is treated as a record.



VPS File Types

VPS provides two types of files for data storage - library files and work files. Library files are sequential files which reside in VPS libraries on direct-access storage devices. Work files are IBM/VS compatible data sets and may reside on tape as sequential files or on disk as direct files.

The choice of file type is dependent upon the size, frequency of use, mode of access, etc. of the file. A complete discussion of VPS file types can be found in the VPS User's Guide.

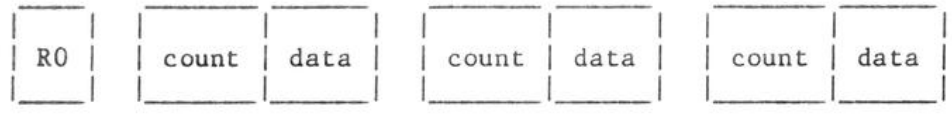
Files on Direct-Access Devices

Regardless of their organization, files created using VPS can be stored on direct-access devices. Each block has a distinct location and a unique address, making it possible to locate any record without extensive searching.

Although direct-access devices differ in physical appearance, capacity, and speed, they are similar in data recording, data checking, and data format. The recording surfaces of each device are divided into many concentric 'tracks'. The number of tracks and their capacity vary with the device.

Information is recorded on all direct-access devices in a standard format. In addition to device data, each track contains a Track Descriptor Record (capacity record or R0) followed by data blocks. There are two possible data formats - count-data and count-key-data.

Track formats



Count Area

The count area contains the location of a data block on a specific track, and defines the size of the key and data areas of that block.

The count area is 9 bytes in length and is defined as follows:

Byte	
0	Track condition flag
1-5	Block location on the volume in CCHHR format
6	Length of the key in bytes (0-255)
7-8	Length of the data in bytes (1-track capacity)

Key Area

Use of the key area is optional and at the discretion of the programmer. When used, the key area of the block contains the identification of the data portion of the block (such as subject code, social sercurity number, or any other uniquely identifying information). The length of the key area is specified in byte 6 of the count area. If the key length is zero, no key area will appear in the block. The maximum key length is 255 bytes. Generally, file organization determines whether keys are used. For example, if a sequential file is processed sequentially, there is no point in writing the file with keys. If, however, there is an appreciable amount of searching required, the blocks should be written with keys.

Data Area

The data area contains the information identified by the count and key areas of the block. The length of the area is specified in the count area.

The QIO Access Method

The QIO access method provides several macro instructions for manipulating sequential and direct files. QIO provides automatic blocking and deblocking of the records written and read. Because the control program synchronizes input/output with processing, programs need not test for completion, errors, or exceptional conditions. After a QGET or QPUT macro instruction is executed, control is not returned to your program until an input area is filled or an output area has been written. Exits to error analysis (SYNAD) and end-of-file (EODAD) routines are automatically taken when necessary.

With QIO the record format used in a program for reading and/or writing is independent of the actual record format of the file. The actual record format of the file is specified on the /FILE statement with the RECFM parameter. The format of the records built by a program for writing, or the record format expected by a program for reading, is specified in the QIO Request Block. QIO will automatically make the appropriate conversion depending upon the request. This allows the assembler program to process a file as format F when it may actually be format V and vice versa.

All QIO operations require a QIO Request Block (QIORB) which informs the control program of the operation to be performed, the unit on which the operation is to be performed, buffer location and length, etc. The QIORB may be built using the QIORB macro or as an inline expansion of the QIO access macros. In the former case, the QIORB can be dynamically modified at execution time by the QIO access macros. The following is a list of the QIO access macros and a description of their function.

QGET - Retrieve a Record

The QGET macro instruction obtains a record from an input file whether it be a disk file, tape file, conversational read unit, etc. QGET allows sequential reading of all unit types and random reading by using relative or absolute record address on disk files and relative record address on tape files. If a direct file with keys is being read with QGET, the key will be read as part of the data. If an end-of-file condition is reached on the file, control will automatically be passed to the user specified EODAD exit routine. Note that for QIO this EODAD exit will only be taken if the end-of-file is encountered when reading a file sequentially. When reading a direct file randomly that has been written sequentially and an end-of-file is encountered, the EOF record (EOFEOFEOF) is placed in the user storage area and control is returned to the user program in the normal way.

QPUT - Write a Record

The QPUT macro instruction places a record into an output file whether it be a disk or tape file, the user terminal, the printer, etc. QPUT allows sequential writing of all unit types, random writing by using relative or absolute record address on direct files, and random writing by using negative relative block address on tapes.

QREW - Rewind a File

The QREW macro instruction repositions a file to the first record in the file regardless of the type of processing. Issuing a QREW macro on a tape file where the last operation to the file was a write will cause an end-of-file (EOF) mark to be written before the tape is repositioned. Issuing the QREW macro on the conversational read unit causes the stack to be cleared. QREW may not be used when processing unit record devices. It should be noted that issuing a QREW macro on a tape file effectively 'closes' that file so that another file on that tape volume may be accessed.

QBSP - Backspace File

The QBSP macro backspaces a file 1 logical record. For example, if a record of a file was read and a QBSP macro was then executed on that file, the next record read would be the same record. QBSP may not be used when processing unit record devices.

QEOF - Write End-of-File

The QEOF macro writes an end-of-file mark in a file which is being processed for output - i.e. records are being written into the file. The QEOF macro can be used only with tape and disk files. If the file was being read, issuing the QEOF macro on the file will reposition to the end of the file. As with QREW, issuing a QEOF macro on a tape file effectively 'closes' the file - allowing another file on that tape volume to be accessed.

Reading the Conversational Read Unit

Assembler language programs, when run at a terminal, may request information to be entered by the terminal user. This is accomplished by executing a QGET macro on the conversational read unit (UNIT=TERMIN is specified on the /FILE statement). If variable length reads are used on the conversational read unit, the Record Descriptor Word (RDW) can be used to detect data overruns. Normally, the second halfword of the RDW (bytes 2 and 3 of the program read area) will be zero. If,

however, a variable length read is issued to the conversational read unit and the typed-in data is larger than the area specified, upon completion of the QGET macro the second halfword of the RDW will contain the difference between the length of the data read and the area specified to receive the data.

QIO Example

In the following example, unit 3, a disk file, is read with variable length reads. If the first byte of the actual record is an EBCDIC 'A' the record is written to a tape file using fixed length writes. When the end-of-file is reached on unit 3, processing will continue at CONT. Note that the actual record formats of the disk and tape file need not be variable and fixed respectively.

```
          .
          .
          .
read      ds    0h
          qget  rb=readrb              read a record
          cli   buff,c'a'              should the record be written
          bne   read                   no, branch
          qput  rb=writerb             yes, write it
          b     read                   go read next record
          .
          .
          .
cont      ds    0h
          .
          .
          .
readrb    qiorb bufad=vbuff,buflen=100,recfm=v,unit=3,eodad=cont
writerb   qiorb bufad=buff,buflen=96,utype=tape
vbuff     ds    x12                    rdw
buff      ds    c196                   actual buffer
          .
          .
          .
```

Reading and Writing Using QIO Record Addressing

Besides sequential reading and writing, disk files may be randomly accessed using the QIO record addressing mode and tape files may be randomly read. It should be noted that tape drives are sequential devices and the use of record addressing on tape files for anything other than reading forward in the file can be highly inefficient. In this mode, the next record read or written is not defined as the next sequential record in the file, but located through a number specified on the QGET or QPUT macro instruction. The two types of record addressing are:

- Absolute. The number of the record specified on the QGET or QPUT macro is relative to the beginning of the file - the first record in the file being record number 1. If, while reading a file using absolute record addressing, a record number of 26 was specified on

the QGET macro, the 26th record of the file would be retrieved.

- Relative. The number of the record specified in the QGET or QPUT macro is relative to the record following the last record read or written. For example, if, while writing a file using relative record addressing, a record number of -1 was specified in the QPUT macro, the QPUT macro would write in the same location of the file as the last record written - i.e. the last record would be overwritten. (This is equivalent to a single backspace and a write.)

Note that relative record addressing can be used when writing tape files provided the record number specified on the QPUT macro is negative.

QIO Record Addressing Example

In the following example every other record of a disk file, unit 2, is read using absolute addressing beginning with the first record. When the end-of-file is reached, the tape is rewound and read again - this time using relative addressing and reading every other record beginning with the second record. The file is processed by the program as fixed-length records.

```
          .
          .
          .
          la    r3,1                    first record number
read1     ds    0h
          qget  rb=rddisk,recno=(3)
          .
          .
          .
          la    r3,2(,r3)               next record number
          b     read1                   go read it
end1      ds    0h
          qrew  rb=rddisk               rewind the disk
* now re-format the qiorb to change rntype and eodad
          qget  rb=rdtape,rntype=rel,eodad=end2,svc=no
read2     ds    0h
          qget  rb=rddisk,recno=1
          .
          .
          .
          b     read2
end2      ds    0h
          .
          .
          .
rddisk    qiorb bufad=buff,buflen=80,unit=2,rntype=abs,eodad=end1
buff      ds    c180
          .
          .
          .
```

The DIO Access Method

The DIO access method can be used to process direct files in a more basic manner than QIO. DIO processes blocks, not records. Therefore, blocking and deblocking of records is the programmer's responsibility. DIO does synchronize I/O scheduling with program execution, so testing for completion is unnecessary. It also provides some facilities for processing direct files which are unavailable with QIO. With DIO the count (as well as key and data) area of the blocks may be read and, for files with keys, key searching can be performed for more efficient random access to the file.

The DIO macro is used to request the control program to perform a specified operation on a direct file. All DIO operations require a DIO Request Block (DIORB) which informs the control program of the type of operation to be performed, the unit number on which the operation is to be performed, buffer location and length, etc. DIORBs may be 'chained' together to perform multiple operations with one DIO macro call. An alternative form of the DIO macro allows the assembler program to modify DIORBs at execution time.

Reading With DIO

Through options specified in the DIORB, the count and/or key and/or data areas of a block may be read. Regardless of whether the file was written with or without keys the amount of data read is dependent on the buffer length specified in the DIORB. If COUNT and KEY are specified and the buffer is large enough, the count-key-data areas will be placed in the buffer. Multiple blocks may also be read by specifying an appropriately large buffer. The block read is the block having the number specified in the BLKNO keyword of the DIO macro instruction - where the first block of the file has BLKNO=1.

Writing With DIO

Blocks may be written randomly into direct files with DIO by using the WRITE option of the DIO or DIORB macro. The block will be written at the block number specified by the BLKNO keyword of the DIO macro. If the file is being written with keys, the key must precede the data in the buffer. As with reading, specifying an appropriately large buffer will allow multiple blocks to be written.

Searching With DIO

Key searches may be performed on direct files by using the search options of the DIORB macro. The key, in the specified buffer, will be compared against the block keys of the file. Depending upon the search option specified - SKEYE, SKEYH, SKEYL - a relationship of true will cause the control program to place the number of the block matched into the first word of the DIORB header, the DIOSBLK field. If the search is unsuccessful, the DIOSBLK field will contain zero. (See the DIORB formats section for information on DIORB differences. Also note that block number feedback will only occur if the search DIORB is the first or only DIORB in the chain. Therefore, placing a search DIORB other than first in a chain will cause unpredictable results.) If READ or WRITE is specified along with one or more of the search options and the search is successful, the data portion of the block will be placed in the buffer area, immediately following the search key (for READ), or the data portion of the block will be replaced by the contents of the buffer area immediately following the search key (for WRITE).

DIORB Formats

As stated previously, DIORBs may be chained together to perform multiple operations. The first 2 positional parameters of the DIORB macro can be used to indicate chained DIORBs, as well as the relationship to the beginning and end of the chain. Chained DIORBs must appear as successive statements in the assembler language program. Chained DIORB lengths differ and are dependent upon their position in the chain. First, or single, DIORBs are generated as - a DIORB header (8 bytes) and a DIORB proper (16 bytes). All other DIORBs in a chain are generated as a single DIORB proper and are 16 bytes in length. The symbols produced by the DSECT option of the DIORB macro should be used to reference information in DIORBs.

Testing For Exceptional Conditions

DIO provides neither EODAD nor SYNAD exits. It is the programmer's responsibility to check for exceptions and errors. After completion of a DIO request, the DIOREQ2 field (offset +1 into the DIORB proper) will be set indicating the completion status of the operation. The relevant bits are:

Bit Position									
0	1	2	3	4	5	6	7		
.	1	.	.	.	.	.	.	DIOCOMP	Request complete
.	.	1	.	.	.	.	.	DIOIOERR	I/O error occurred
.	.	.	1	.	.	.	.	DIORQERR	Request in error
.	.	.	.	1	.	.	.	DIOEOF	EOF read on file

DIO Example - Processing A File Without Keys

In the following example, a direct file, created without keys, is processed using DIO. On each pass through the code, 3 blocks of the file are read, processed, and the second of the 3 is modified and rewritten. The file is format F with a block length of 50 bytes. The alternate form of the DIO macro is used each time to update the block numbers in the read DIORBs before the request is issued. The DSECT option of the DIORB macro is used to address the DIOREQ2 field in the read and write DIORBs to test for successful completion. Note that the read operation in done with chained DIORBs - it could also be accomplished with 1 read of 150 bytes. When an EOF is encountered on the file, processing continues at FINISH.

```
          .
          .
          .
          la    r4,readrb               addressability to read diorb
          la    r5,writerb              and to write diorb
          la    r3,1                    first block number
read      ds    0h
          dio   +2,rb=readrb,blkno=2(r3),svc=no  format last read
          dio   +1,rb=readrb,blkno=1(r3),svc=no  format middle
          dio   rb=readrb,blkno=(r3)    format first and issue req
          using diorb,r4
          tm    dioreq2,diorqerr        end-of-file?
          bo    finish                  yes, branch
```

```
|          tm    dioreq2,dioioerr       error?                          |
|          bnz   readerr                yes, branch                     |
|          drop  r4                                                     |
|          .                                                            |
|          .                                                            |
|          .                                                            |
|          dio   rb=writerb,blkno=1(r3)  write the second of the set    |
|          using diorb,r5                                               |
|          tm    dioreq2,dioioerr+diorqerr  error?                      |
|          bnz   writerr                yes,branch                      |
|          la    r3,3(,r3)              increment block pointer         |
|          b     read                   and read next set of blocks     |
|          drop  r5                                                     |
|          .                                                            |
|          .                                                            |
|          .                                                            |
| finish   ds    0h                                                     |
|          .                                                            |
|          .                                                            |
|          .                                                            |
| readrb   diorb s,+,unit=4,bufad=buff1,buflen=50,opt=read              |
|          diorb +,+,bufad=buff2,buflen=50,opt=read                     |
|          diorb +,e,bufad=buff3,buflen=50,opt=read                     |
| writerb  diorb unit=4,bufad=buff2,buflen=50,opt=write                 |
| buff1    ds    c150                                                   |
| buff2    ds    c150                                                   |
| buff3    ds    c150                                                   |
|          diorb dsect                                                  |
|          .                                                            |
|          .                                                            |
|          .                                                            |
 -----------------------------------------------------------------------
```

DIO Example - Processing A File With Keys

In the following example, a direct file (unit 4), created with keys, is processed with DIO. A sequential file (unit 3), consisting of 80 byte records, is read using format F QIO. Each record contains update information for the direct file - uniquely identified by a 10 byte key. As each record from the sequential file is read, the key from the record is used to search the direct file. If a match occurs, the data area of the matching block is read into the buffer, updated from information on the sequential record, and the block is rewritten. The direct file contains less than 3000 90-byte records (10 bytes for the key followed by 80 bytes of data). When an EOF condition is encountered on the sequential file, processing continues at FINISH.

```
 -----------------------------------------------------------------------
|          .                                                            |
|          .                                                            |
|          .                                                            |
|          la    r3,directr             addressability to read diorb    |
|          la    r4,directw             and to write diorb              |
| read     ds    0h                                                     |
|          qget  rb=update              read an update record           |
|          mvc   key(10),record         move the key to the buffer      |
|          dio   rb=directr             search for matching block       |
```

```
          using diorb,r3
          tm    dioreq2,dioioerr+diorqerr  error?
          bo    readerr                    yes, branch
          icm   r2,b'1111',diosblk         was the search successful?
          bz    notfound                   no, branch
          drop  r3
          .
          .
          .
          dio   rb=directw,blkno=(r2)      write the updated block
          using diorb,r4
          tm    dioreq2,dioioerr+diorqerr  error?
          bnz   writerr                    yes, branch
          b     read                       continue updating
          drop  r4
          .
          .
          .
finish    ds    0h
          .
          .
          .
update    qiorb bufad=record,buflen=80,unit=3,eodad=finish
directr   diorb unit=4,bufad=buffer,buflen=90,blkno=1,scope=3000,   *
                opt=(skeye,read)
directw   diorb unit=4,bufad=buffer,buflen=90,opt=write
record    ds    cl80
buffer    ds    0cl90
key       ds    cl10
buff      ds    cl80
          diorb dsect
          .
          .
          .
```

Chapter 3: Load Modules

This chapter discusses the VPS facilities that allow assembler programs access to load modules that reside in system libraries or user work files.

Load Module Creation

Each time object decks are loaded into virtual storage for execution by the VPS LOADER, they must be processed to resolve external references, relocate addresses, etc. This process is called loading, and it creates a load module in virtual storage which is ready for execution. After execution of the load module in virtual storage, it is deleted.

Since object decks must be processed in this way each time they are loaded, for object decks which are modified infrequently and executed often, this mean a large amount of redundant processing. Much of this redundant processing may be alleviated by linkage editing the object deck(s). Linkage editing involves most of the same processing as loading but instead of creating the load module in virtual storage and executing it, the linkage editor places the load module in a direct access work file. The load module is then ready for use in execution in place of the object decks.

Load modules may be created from an object deck, or set of object decks, and placed in a direct access work file using the VPS /LOAD LINKEDIT command (see the VPS User's Guide for details). The product of the linkage editor is a self-contained 'unit' which, when loaded into virtual storage by the control program, is ready for execution.

Load Module File Structure

Work files which contain load modules are divided into two sections. The first section is the file directory. The directory contains an entry for each load module in the file. Each entry contains information specific to that load module - the load module name, location in the file, entry point location, type of load module, etc. (see the DSECT LKDSECTS for detailed information). Each directory entry is 32 bytes in length.

The second section of the file contains the actual load modules. The location of each load module is specified in its directory entry.

Loading A Load Module Into Virtual Storage

Load modules may be loaded into virtual storage either from work files containing load modules or from system libraries. System libraries are files known to the VPS control program, such as, the system link library (SYS.LINKLIB) and system subroutine libraries (SYS.FORTLIB, SYS.PLIXLIB, etc.).

The VPS macros used to load a load module into virtual storage from an assembler language program are - LMLOAD, LMARG, and SYSLIB. LMLOAD requests the control program to search the file directory for a load module and load it into virtual storage at an address specified by the program. The LMARG macro builds the LMLOAD argument list. The SYSLIB

macro may be used if the programmer wishes the control program to search a list of system libraries and/or work files for the directory entry of the specified load module. Note that if a work file unit number is specified using LMLOAD a valid /FILE command must also be present defining that work file.

Specifying The Load Address

Load modules may be loaded into virtual storage at any location specified by the program - with the exception of low core addresses used by VPS. The standard technique is to load load modules at the next available storage location past the program which is issuing the LMLOAD macro. Acquiring this address dynamically may be accomplished by including a 'dummy' CSECT as the last CSECT of the program and using a V-type address constant for that CSECT name in the CSECT issuing the LMLOAD macro.

Specifying The Length

Besides the load address, the LMLOAD macro also requires the length of the storage area into which the load module is to be loaded. If the load module is to be loaded at the next available storage location past your program, the length of the available storage area may be found by using a special low core cell, defined by the LOCORE DSECT, called $FUBOUND (see Appendix A for further information). $FUBOUND is a fullword containing the upper boundary storage location. If the next available address in user storage is subtracted from the value in $FUBOUND the result will be the length of the available storage area.

Searching The SYSLIB List

A list of system libraries and/or work files may be specified using the SYSLIB macro. The list will be searched in the order specified in the list and the control program will load the first occurrence of the load module.

Passing Control To A Load Module

At the completion of the LMLOAD macro, if the module has been loaded successfully, register 0 will contain the amount of storage actually used by the load module and register 1 will contain the entry point address of the load module. The load module is now ready for execution. Passing control to the load module can, and should, be accomplished using standard linkage conventions - register 13 should contain the address of an available save area; register 1 should contain the address of the parameter list if the load module is expecting one; register 14 should contain the return address; and register 15 should contain the entry point address of the load module.

Example

In the following example, a load module - COSIN - is to be loaded into the next available storage location. The control program will search SYS.PLIXLIB and the work file units 3 and 4. The program will then pass control to the load module.

```
         .
         .
         .
         l     r5,=v(dummyc)          get start of free storage
         lr    r6,r5                  retain a copy
         l     r7,$fubound            end of available storage
         sr    r7,r5                  length of free storage
         lmload lmarg=loada,addr=(r6),len=(r7)
         ltr   r15,r15                was load successful?
         bnz   loaderr                no, branch
         lr    r15,r1                 entry address to r15
         la    r13,save               new save area address
         la    r1,parms               parameters for cosin
         balr  r14,r15                call cosin
         .
         .
         .
loada    lmarg modname=cosin,syslib=list
list     syslib list=(sys.plixlib,3,4)
         .
         .
         .
```

Loading By Directory Entry

In certain cases the programmer may wish to read a load module directory entry into storage before loading the actual load module. For example, if more than one module is to be loaded and executed from the same set of libraries and/or work files, acquiring all of the directory entries in one search and then loading each separately, is more efficient than searching for and loading each independently. The VPS macro LMBLDL - in conjunction with LMBLIST, LMARG, and SYSLIB - should be used to read directory entries into virtual storage.

Using The Directory Entry After It Is Read

The LMBLIST macro instruction constructs an area into which a directory entry, or entries, are read by the control program during execution of the LMBLDL macro. After a directory entry has been brought into virtual storage, it can be used with the LMLOAD macro instruction to load the specified module. In this case the control program will not search any libraries since the information it needs to load the module is found in the directory entry. But before the LMLOAD macro can be successfully executed, the directory entry must be moved to the LMLOAD argument list. See the LMBLIST macro description for further information.

Example

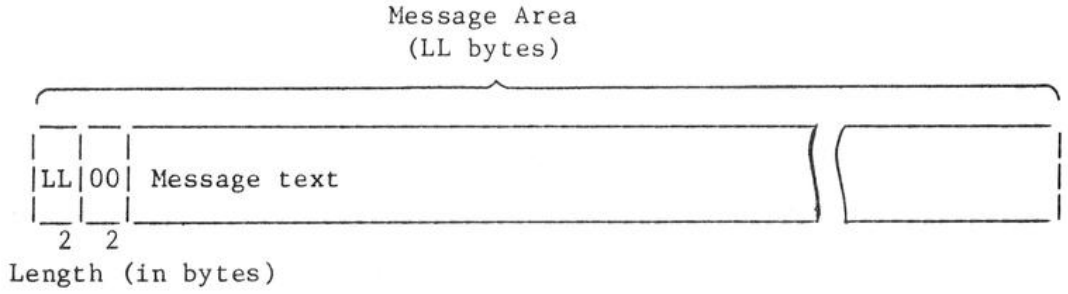
In the following example, two load modules, CONVERT1 and CONVERT2, are to be loaded from work file unit 3. Control is to be passed to each load module in turn. LMBLDL is used to read the directory entries into storage before each is loaded with LMLOAD.

```
          .
          .
          .
          l     r5,=v(dummyc)          get start of free storage
          lr    r6,r5                  retain a copy
          l     r7,$fubound            end of available storage
          sr    r7,r5                  length of free storage
          lmbldl lmarg=barg,addr=bldlst
          ltr   r15,r15                bldl successful?
          bnz   bldlerr                no, branch
          mvc   larg+4(32),bldlst+4    move first directory entry
          lmload lmarg=larg,addr=(r6),len=(r7)
          ltr   r15,r15                load successful?
          bnz   loaderr                no, branch
          lr    r15,r1                 entry address to r15
          la    r13,save               new save area address
          la    r1,parms1              parm address for convert1
          balr  r14,r15                call convert1
          mvc   larg+4(32),bldlst+36   move in next directory entry
          lmload lmarg=larg,addr=(r6),len=(r7)
          ltr   r15,r15                load successful?
          bnz   loaderr                no, branch
          lr    r15,r1                 entry address to r15
          la    r13,save               new save area address
          la    r1,parms2              parm address for convert2
          balr  r14,r15                call convert2
          .
          .
          .
barg      lmarg bldl,unit=3
bldlst    lmblist modname=(convert1,convert2)
larg      lmarg dir,unit=3
          .
          .
          .
```

This chapter discusses the VPS message building and sending facilities which can be used in assembler language programs.

Sending Messages

Assembler language programs may send messages to the terminal user (to the printer, if the program is being run in batch), to another terminal user (another account), or to the system operator using the SMSG macro. The VPS message facility sends messages having a variable length format; that is, the message area, or buffer must be constructed as follows:



Where the first 2 bytes of the area contain the length (LL) of the area, and the second 2 bytes must contain hexadecimal zeros followed by the message text.

Variable length messages may be built either with the MESGV macro or with the BLDMSG macro operating on a MESGB pattern macro. Note that the above format is the same as required by the QIO access method and messages built with these macros can be used by QIO (see Chapter 2 for further information).

Building Messages With MESGV

The MESGV macro may be used to build simple messages - messages which require minimal editing. The MESGV macro can be coded to leave blank spaces in a message and assign a user-specified symbol to that area - allowing easy text insertion during execution.

In the following example 2 messages are sent, the first to the terminal user and the second to the system operator. Before the first message is sent, a converted number is inserted into the message.

```
        .
        .
        .
        l       r2,numrecs              get number of records
        cvd     r2,dfield               convert it to decimal
        unpk    msgal(4),dfield         unpack it into the message
        mvz     msgal+3(1),msgal+2      clear sign
        smsg    *,msga                  send it to terminal
        ltr     r15,r15                 sent successfully?
        bnz     msgerr                  no, branch
```

```
         smsg  op,msgb                send message to operator
         ltr   r15,r15                sent successfully?
         bnz   msgerr                 no,branch
         .
         .
         .
dfield   ds    d
numrecs  ds    f
msga     mesgv ' phase 1: ',msgal,(4),' records read'
msgb     mesgv 'please mount tape - data02'
         .
         .
         .
```

In this example, the terminal user would receive the message:

phase 1: 0038 records read

The system operator would receive the message:

please mount tape - data02

Generating Messages With BLDMSG

VPS provides extensive facilities for complex message generation through the use of the BLDMSG macro. The BLDMSG macro generates variable length messages by editing data areas, specified in a list on the BLDMSG macro, into a programmer-specified message buffer under the control of a MESGB message pattern. The MESGB macro defines the text information, data conversion controls, and formats for the message being built.

Through operands coded on the MESGB macro, BLDMSG provides data conversion - data areas can be automatically converted to decimal and edited into a message; leading zero and leading and trailing blank truncation - significant digits or non-blank characters can be automatically edited into a message; and, for often used information - such as date and time - automatic retrieval, conversion, and editing of this information into the message. Other operands on the MESGB macro can also provide absolute and relative tabbing during message generation, and conversion of data areas to a 'dollars and cents' format. A full description of these and other facilities can be found in the MESGB macro description in Part II of this manual.

After BLDMSG has generated a message in a buffer, the SMSG macro can be used to send the message. Note that since BLDMSG stores the buffer length into the first 2 bytes of the buffer, the actual variable length message begins at offset +2 into the buffer.

The following is an example of BLDMSG message generation. The date and time are to be retrieved, converted, and placed in the message buffer. PHASE, a halfword, is to be converted to decimal and the significant digits placed in the message; as is NUMRECS, a fullword.

```
          .
          .
          .
          bldmsg buffer,pattern,(phase,numrecs),buflen=135
          ltr   r15,r15                 msg built successfully?
          bnz   msgerr                  no, branch
          smsg  *,buffer+2              send it
          ltr   r15,r15                 sent successfully?
          bnz   msgerr                  no, branch
          .
          .
          .
phase     ds    h                       phase number
numrecs   ds    f                       number of records
buffer    ds    c1135
pattern   mesgb ' ',date,' ',time,' phase ',(dec,2),': ',          *
                (dec,4),' records read'
          .
          .
          .
```

In this example, the terminal user would receive a message that might look like the following:

thu mar 03, 1977 13:11:42 phase 3: 472 records read

Sending Messages and Receiving Replies

Assembler language programs may not only send variable length messages but also receive and process replies from these messages using the SMSGR, SMSGRARG, and TWAIT macros. Messages may be sent to and corresponding replies received from the terminal user (if the program is run in the batch, the VPS operator will receive the message), the VPS operator, or another terminal user (another account).

The SMSGR macro is used to send the message and notify the control program that a reply is to be expected. The SMSGR macro passes information to the control program using an argument list. The argument list, which is built using the SMSGRARG macro or constructed as part of the inline expansion of the SMSGR macro, contains: the address of the variable length message, the address and length of the reply area, and posting information to be used by the control program (see below). Messages sent by SMSGR are standard variable length messages which may be built using the MESGV or BLDMSG macros.

Posting the User Program

When the SMSGR macro is executed, the VPS control program receives control and sends the message to the desired destination. Control is then returned to the user program, which will continue execution while the control program awaits the reply from the destination user. When the reply is received and placed in the reply area the control program 'posts' the user program by 'ORing' 1's into a post byte whose address is passed in the SMSGR argument list under the control of a posting

mask, also passed in the argument list.

Therefore, if the programmer wishes to process the reply, the user program must be made to wait until the reply has been received (i.e. the user program has been 'posted'). This can be accomplished using the Test-under-Mask/TWAIT combination (see the TWAIT macro description).

Example using SMSGR and TWAIT

In the following example a message is sent to the VPS operator asking if tape 'DATA14' is ready for mounting. The SMSGRARG macro is used to build the argument list. If the message is sent successfully (see the SMSGR macro description for a list of return codes), the Test-under-Mask/TWAIT combination are used to wait on the post byte STAT until the reply is received. The reply is then checked in the following manner: if the reply is 'yes', execution continues; if it is 'no', execution is terminated; and if it is neither of these, the message is repeated.

```
          .
          .
          .
msgloop   ds    0h
          ni    stat,x'7f'            clear the reply posting bit
          smsgr op,smarg=msgarg
          ltr   15,15                 message sent successfully?
          bnz   msgerr                no, branch
          tm    stat,x'80'
          twait                       wait for the reply
          clc   reparea(3),=cl3'no '  is it 'no'?
          be    end                   yes, branch
          clc   reparea(3),=cl3'yes'  if it is not 'yes' ...
          bne   msgloop               send it again
          .
          .
          .
msgarg    smsgrarg ,opmsg,reply=reparea,replen=3,post=stat,        *
               pbits=x'80'
reparea   dc    cl3' '
stat      dc    x'00'
opmsg     mesgv 'is tape - data14 - ready for mounting? reply yes  *
               or no'
          .
          .
          .
```

This chapter discusses the timing facilities available to assembler language programs in VPS. Besides providing time of day, VPS also provides execution and interval timing facilities.

Time of Day

The time of day may be requested by issuing the TOD macro which returns the time of day in 300ths of a second units in register 0. Below is an example where the time of day is requested and then converted to seconds.

```
          .
          .
          .
          tod
          srdl  r0,32                 move tod to r1
          d     r0,=f'300'            convert to seconds
          st    r1,seconds            store it
          .
          .
          .
```

Execution Timing

The VPS control program provides the ability to request the accumulated CPU time for a terminal session or batch job using the EXTIME macro. EXTIME will return the accumulated CPU time, in 300ths of a second units, in register 0. By issuing the EXTIME macro at different places in a program and subtracting the last value from the new value, the CPU time for different sections of a program can be calculated. In the following example a loop is timed by issuing an EXTIME macro before the loop is entered, after the loop is ended, and then storing the difference.

```
          .
          .
          .
          extime
          st    r0,timer              store accumulated cpu time
loop      ds    0h                    start loop
          .
          .
          .
          bxh   4,6,loop
          extime
          s     r0,timer              calculate difference
          st    r0,looptime           store cpu time for loop
          .
          .
          .
```

Interval Timing

A time interval can be established in a program by using the SETIMER macro, and the time remaining can be tested or canceled by using the TESTIMER macro. Only one time interval can be active at a time.

By issuing the SETIMER macro the programmer may specify either a time of day or an interval value (both in 300ths of a second units) indicating when the time interval is to expire. At that point, the control program will give control to the timer completion routine specified by the SETIMER macro instruction.

The timer completion routine must save and restore registers and, if appropriate, return control to the address in register 14. When the timer completion routine returns, control will be passed to the next instruction in the main program after which the time interval expired. Also specified on the SETIMER macro instruction is the address of a save area which will be passed to the timer completion routine in register 13.

In the following example a time interval of 4 seconds is set for a loop in a program. If the time interval expires before the loop is completed, the timer completion routine will be given control. If the loop completes before the interval expires, the interval will be canceled with the TESTIMER macro. If the timer completion routine is entered, it will save its registers in the save area addressed by register 13 and turn on a switch tested in the loop. At the end of the timer completion routine the registers will be restored and control returned to the address in register 14. The control program will then return control to the next instruction in the loop. When the switch is tested again the loop will terminate.

```
          .
          .
          .
          setimer save,clock=1200,exit=timexit
loop      ds    0h                    start loop
          tm    timer,x'01'           has interval expired?
          bo    loopexit              yes, leave loop
          .
          .
          .
          bxh   2,4,loop
          testimer cancel
loopexit  ds    0h
          .
          .
          .
          using timexit,r15           addressability for
timexit   ds    0h                    timer interrupt routine
          save  (14,12)               save registers
          oi    timer,x'01'           say time has expired
          .
          .
```

```
          .
          return (14,12)              restore registers
          .
          .
          .
save      ds    18f                 savearea for timer routine
timer     dc    x'00'               timer switch
          .
          .
          .
```

Part II: Macro Instructions

Introduction

Service requests are communicated to the VPS control program using a set of macro instructions. These macro instructions are available only when programming in assembler language. All VPS macros and DSECTs reside under the VPS library index - VPSMAC.

The processing of the macro instructions by the assembler results in a macro expansion generally consisting of data and executable instructions in the form of assembler language statements. The data fields are the parameters to be passed to the requested control program routine; the executable instructions generally consist of (where applicable) a branch around the data, instructions to load registers, and a supervisor call (SVC) to give control to the proper control program routine.

Macro Instruction Forms

When written in the standard form, some of the macro instructions result in instructions that store into an in-line parameter list. Out-of-line parameter lists are also supported through the use of a separate set of macros which, instead of generating assembler language instructions, generate data areas which can be used by the various macro instructions requiring parameter lists and generating SVCs. These macros are also documented in the following section.

Macro Instruction Formats

The symbols [], {}, __, and ... are used to indicate how a macro instruction may be written. DO NOT CODE THESE SYMBOLS. Their general definitions are given below:

[] indicates optional operands. The operand enclosed in brackets may or may not be coded, depending on whether or not the associated option is desired or whether a default is to be taken. If more than one item is enclosed in brackets, one or none of the items may be coded.

{} indicates that a choice must be made. One of the operands from the vertical stack must be coded, depending on which of the associated services is desired.

__ indicates a value that is used in default of a specified value. This value is assumed if the operand is not coded.

.. indicates that an operand, or set of operands, may be repeated. The number of repetitions is dependent upon the function desired.

VPS macro instructions follow the rules of OS/VS Assembler Language. When writing VPS macro instructions using the format defined in the following section, write them using the general rules listed below:

-- If a selected operand is written in all capital letters (for example DIR, SOFF, SET), code the operand exactly as shown.

-- If the selected operand is written in lower case, substitute the indicated value, address, or name.

-- If the selected operand is a combination of capital and lower case letters separated by an equal sign (for example LENGTH=number) code the capital letters and equal sign exactly as shown, then make the indicated substitution.

-- Code commas and parentheses exactly as shown, only omit the comma following the last operand coded. The use of commas and parentheses is indicated by brackets and braces following the afore mentioned rules.

Note: Capital letters are used in this manual only to differentiate between symbols and keywords required by the macros and symbols and names selected by the programmer. If a macro definition is being typed in on a terminal it should not be typed in upper case.

VPS Macro Restrictions

In general, the VPS service macros may use registers 0, 1, 14, and 15 in generated code. The programmer is forewarned that dependence upon the contents of one of these registers across the execution of a service macro can lead to disaster. A small number of macros will use other registers. These registers are noted in the section describing that macro.

The VPS service macros are supplied to aid the knowledgeable assembler programmer in writing sophisticated programs. The macros which generate executable code and the macros that generate data areas and parameter lists could be replaced by their assembler language equivalents. THIS IS NOT RECOMMENDED. The executable code and data areas generated by the VPS service macros are subject to change and should, therefore, not be replaced by their assembler language equivalents.

The VPS service macros recognize the equated symbols 'r0' through 'r15' as equates for registers 0 through 15. If equated symbols 'r0' through 'r15' are used in programs that issue VPS service macros they must have the absolute values 0 through 15 respectively to ensure correct execution of the code generated by the macros.

Macro Instructions

The following are the VPS service macro instructions. They are arranged in alphabetical order.

ABCODE - Abend Code

The ABCODE macro builds a parameter list used by the ABEOJ macro to abnormally terminate a program. The ABCODE macro does not generate any instructions. The standard form of the ABCODE macro is written below.

```
[symbol]   ABCODE  {Unnn} {,'message'}
```

Unnn
: specifies the user abend code. The abend code may be any number between 0 and 255 and must be preceded by a U to indicate a user abend.

'message'
: specifies a message which the programmer wishes to have printed at the time of the abend. The message must be enclosed in quotes.

ABEOJ - Abend/End Of Job

The ABEOJ macro instruction is used to terminate a program abnormally. It will cause the abend condition flag to be set, a message specified in the program to be printed, and the program to be terminated. The standard form of the ABEOJ macro instruction is written as follows.

[symbol] ABEOJ $\left\{\begin{matrix}\text{addr}\\(\text{reg})\end{matrix}\right\}$

addr

specifies the address of an area containing the user abend code and message to be printed (see the description of the ABCODE macro for information on defining the area). Alternatively, a register may be specified containing the address of this area. If a register is specified, the register must be enclosed in parentheses and may be designated either symbolically or as an absolute expression.

Note:

Besides the abend message, the control program will also print the contents of the registers and the PSW which were in effect at the point in the macro expansion where the ABEOJ SVC was issued. Since the ABEOJ macro uses register 1, if the programmer wishes to have the contents of register 1 (before the execution of the macro) to be displayed, it should be loaded into another register prior to the issuance of the macro.

ACCMOD - Access Modifier Area

The ACCMOD macro is used to define an Access Modifier Area. The area is used to specify file access attributes for library files created and changed with the LIBRARY and LIBRCLS macros. The Access Modifier Area is addressed through the LIBRARY argument list (LIBARG) or is generated as part of the LIBRCLS argument list (LCLARG). In both cases the Access Modifier Area must be initialized by the user program. The standard form of the ACCMOD macro is written below.

```
[symbol]    ACCMOD   [DSECT]
                     [GEN  ]
```

first positional parameter
specifies whether the Access Modifier Area should be defined. If DSECT is specified the area is not defined. Instead, an Access Modifier DSECT is generated. If GEN is specified, a zeroed Access Modifier Area is generated. GEN is the default.

Notes:
Since it is the programmer's responsibility to initialize the area, the Access Modifier Area DSECT is described below.

```
LAACCFLD DSECT
LAACD1   DC    XL2'0'          ON/OFF BYTES FOR ACCODE1
LAACD2   DC    XL2'0'          ON/OFF BYTES FOR ACCODE2
         DC    XL2'0'          RESERVED
LAAPTYPE DC    X'0'            PERMIT FIELD TYPE
         DC    X'0'            RESERVED
LAAPFLDS DC    2CL8' '         PERMIT FIELDS
LAALNGTH EQU   *-LAACCFLD
```

LAACD1
defines the permitted file access for all accounts other than those specified in the LAAFLDS field. The first byte (the ON byte) will be 'ORed' to the default (for new files), or existing (for replaced files), access byte. Then the second byte (the OFF byte) will be 'ANDed' to the first result. The result of the last operation will define the access codes. If, for each bit position in the access byte, there is a 1 then that type of access will be allowed. The access byte has the following format:

Bit Position								
0	1	2	3	4	5	6	7	
.	.	.	.	.	.	.	1	Read access
.	.	.	.	.	.	1	.	Execute access
.	.	.	.	.	1	.	.	Purge access

For example, if the default read and purge access is to be kept and execute access is to be allowed, then the program should initialize the first byte with X'02' and the second byte with X'07'.

LAACD2

defines the permitted file access for the account(s) specified in the LAAPFLDS field. These 2 bytes are structured and used in the same manner as the LAACD1 field.

LAAPTYPE

indicates whether the LAAPFLDS field contains an account range. X'0' specifies that the LAAPFLDS field is not being used (the LAACD2 field is also not needed). X'01' specifies that the LAAPFLDS field does contain an account range whose access is defined in the LAACD2 field.

LAAPFLDS

defines an account range which is to have the access attributes to the file as specified in the LAACD2 field. The first 8-character field specifies the first account of the range. It may contain an actual 8-character account or from 1 to 7 characters of an account followed by hexadecimal zeroes indicating where the account comparison is to stop (this corresponds to a '*' in the PERMIT= keyword of the associated VPS terminal commands - see the VPS User's Guide). The second 8-character field specifies the last account of the range. It may contain an actual 8-character account or from 1 to 7 characters of an account followed by hexadecimal FF's indicating where the account comparison is to stop. Obviously, the first account of the range must compare low or equal to the last account of the range.

BLDMSG - Build A Variable Length Message From A Pattern

The BLDMSG macro is used to build a standard v-format message in an area, or buffer, in user storage using a pattern constructed with the MESGB macro. After the BLDMSG macro is executed, the message in the buffer can be sent using the SMSG macro or written using the QIO access method. For further information on the BLDMSG macro, see the description of the MESGB macro. The standard form of the MESGB macro is written below.

```
[symbol]   BLDMSG  {buffer address}{,pattern address}
                   {(reg1)       }{,(reg2)        }
                   [(data address[,data address]...)]
                   [,BUFLEN=length]
                   [,BUFLEN=(reg)]
```

buffer address
specifies the address of the buffer in which the control program is to build the message.

(reg1)
specifies a register containing the address of the message buffer.

pattern address
specifies the address of the MESGB message pattern.

(reg2)
specifies a register containing the address of the MESGB message pattern.

(data address[,data address]...)
specifies a list of data addresses either in the form of a symbol or base and displacement - D(B). Each points to an area containing information which will be converted to EBCDIC text based on options specified in the pattern and inserted into the message buffer at locations also specified in the pattern. If only one data address is specified, the parentheses may be omitted.

BUFLEN=
specifies the length of the message buffer in bytes. If a register is specified, it must be enclosed in parentheses and the value in the register will be used as the buffer length. If BUFLEN is not coded the first 2 bytes of the area specified by buffer address must contain the length of the buffer.

Notes:

When control is returned to the program, register 15 will contain one of the following return codes:

Hexadecimal Code	Meaning
0	Message built successfully.
4	Buffer too small.
8	Too few data elements.
C	Invalid argument.

BRANCH - Branch and Set Condition Code/Program Mask

The BRANCH macro is used to branch to a specified address in user storage without the use of registers. In addition, it will also set the condtion code and the program mask in the PSW when the branch is effected. The standard form of the BRANCH macro instruction is written below.

```
[symbol]   BRANCH  {addr }  [,CCPM=number ]
                   {(reg)}  [,CCPM=(reg)  ]
                            [,CCPM=00     ]
```

addr
is the location to which the branch is to be made. If (reg) is specified, the address in the designated register is used. Execution of the macro causes the 'branch to' address to be stored in bytes 5-7 of the resume PSW effecting the branch.

CCPM=
specifies the condition code and program mask. The condition code/program mask can be coded as either a 2 digit hexadecimal number or as a register. If a register is specified, it must be enclosed in parentheses and the condition code/program mask must be in the leftmost byte of the register. The condition code/program mask will replace byte 2 of the resume PSW. Below is a chart of each bit location in the condition code/program mask and its meaning.

	Condition Code				Program Mask				
Bit	0	1	2	3		4	5	6	7	
	ignored		0	0	zero/equal	1				Fxd. Pt. Overflow
			0	1	<zero/low		1			Dec. Overflow
			1	0	>zero/high			1		Expon. Overflow
			1	1	overflow				1	Significance

When the program mask bit is one, the exception results in an interruption. When the mask bit is zero, no interruption occurs.

BRTRAP - Branch and Trap SVCs

The BRTRAP macro instruction is used to branch to a specified address in user storage without the use of registers and turn on the user SVC trapping bit in the JCB. In addition, it will also set the condition code and program mask in the PSW when the branch is effected. The standard form of the BRTRAP macro is written below.

```
[symbol]    BRTRAP  {addr }  [,CCPM=number]
                    {(reg)}  [,CCPM=(reg)  ]
                             [,CCPM=00     ]
```

addr
: is the location to which the branch is to be made. If (reg) is specified, the address in the designated register is used. Execution of the macro causes the 'branch to' address to be stored in bytes 5-7 of the resume PSW effecting the branch.

CCPM=
: specifies the condition code and program mask. The condition code/program mask can be coded as either a 2 digit hexadecimal number or as a register. If a register is specified, it must be enclosed in parentheses and the condition code/program mask must be in the leftmost byte of the register. The condition code/program mask will replace byte 2 of the resume PSW. Below is a chart of each bit location in the condition code/program mask and its meaning.

	Condition Code				Program Mask				
Bit	0	1	2	3		4	5	6	7
	ignored		0	0	zero/equal	1			Fxd. Pt. Overflow
			0	1	<zero/low		1		Dec. Overflow
			1	0	>zero/high			1	Expon. Overflow
			1	1	overflow				1 Significance

When the program mask bit is one, the exception results in an interruption. When the mask bit is zero, no interruption occurs.

CLRSTOR - Clear Storage

The CLRSTOR macro instruction is used to clear an area in user storage. The storage is 'cleared' by setting it to binary zeroes. Note that the control program will also release any complete virtual pages cleared by the macro. The standard form of the CLRSTOR macro instruction is written below.

```
[symbol]    CLRSTOR  {addr }  [,LENGTH=number]
                     {(reg)}  [,LENGTH=(reg)  ]
```

addr
: is the address of the first byte of the area in user storage to be cleared. If (reg) is specified, the address in the designated register is used.

LENGTH=
: is the number of bytes to be cleared. It can be coded either as a number or non-relocatable symbol, or as a specific register containing the number of bytes to be cleared. If a register is specified, it must be enclosed in parentheses. The register may be designated symbolically or with an absolute expression. If this keyword is omitted the length defaults to the length of addr.

DADSM - Direct Access Device Space Management

The DADSM macro may be used to scratch (delete) or rename a VPS disk work file. The standard form of the DADSM macro is written below.

```
[symbol]   DADSM  [RENAME ]  {,DADARG=addr }
                  [SCRATCH]  {,DADARG=(reg)}

                  [,VOLSER=addr      ]  [,NAME=addr      ]
                  [,VOLSER=(reg)     ]  [,NAME=(reg)     ]
                  [,VOLSER='volid'   ]  [,NAME='filename']

                  [,NEWNAME=addr      ]
                  [,NEWNAME=(reg)     ]
                  [,NEWNAME='newfilename']
```

RENAME

is written as shown and specifies that the work file is to be renamed. Note that this parameter will override the TYPE parameter of the DADSM argument list macro.

SCRATCH

is written as shown and specifies that the work file is to be scratched (deleted) from the volume. Note that this parameter will override the TYPE parameter of the DADSM argument list macro.

DADARG=

specifies the address of the DADSM argument list. It may be specified either as an address or as a register containing the address of the argument list. If a register is coded, it must be enclosed in parentheses. The DADSM argument list should be built using the DADSMARG macro.

VOLSER=

specifies the volume serial number of the device on which the work file resides. It may be specified as: an address of a 6-byte area containing the volume serial number; a register holding the address of a 6-byte area containing the volume serial number; or the actual volume serial number enclosed in quotes. If this parameter is specified, it will replace the volume serial number in the DADSM argument list. Note, VOLSER=(1) may not be coded.

NAME=

specifies the name of the work file to be scratched or renamed. It may be specified as: an address of a 44-byte area containing the name of the work file; a register, holding the address of a 44-byte area containing the work file name; or the actual work file name enclosed in quotes. If this parameter is specified, it will replace the work file name in the DADSM argument list. Note, NAME=(1) may not be coded.

NEWNAME=

(For RENAME requests only)
specifies the new name of the work file. It may be specified as: an address of a 44-byte area containing the new work file name; a register, holding the address of a 44-byte area containing the new work file name; or the actual new name of the work file enclosed in quotes. Note, NEWNAME=(1) may not be coded.

Note:

When control is returned to the program, register 15 will contain one of the following return codes:

Hexadecimal Code	Meaning
0	Operation successfully completed.
4	Invalid argument.
8	I/O error.
C	Volume not mounted.
10	Work file not found.
14	Access not allowed.
18	Duplicate name on volume.

DADSMARG - DADSM Argument List

The DADSMARG macro builds the DADSM argument list. No instructions are generated by this macro. The standard form of the DADSMARG macro is written below.

```
[symbol]   DADSMARG   [DSECT]   [,TYPE=RENAME]    [,VOLSER='volid']
                                 [,TYPE=SCRATCH]

                    [,NAME=addr]          [,NEWNAME=addr]
                    [,NAME='filename']    [,NEWNAME='newfilename']
```

DSECT
specifies that the actual DADSM argument list should not be built. Instead, a DADSMARG DSECT is generated.

TYPE=
specifies the type of DADSM request. Either SCRATCH, to delete a work file, or RENAME, to change a work file name, may be specified.

VOLSER=
specifies the 6-character volume serial number of the device on which the work file resides. The volume serial number must be enclosed in quotes.

NAME=
specifies the name of the work file. It may be coded as either the address of a 44-byte area containing the work file name or as the actual work file name enclosed in quotes.

NEWNAME=
(For RENAME requests only)
specifies the new work file name. It may be coded as either the address of a 44-byte area containing the new work file name or as the actual new work file name enclosed in quotes.

Note:
All of the keywords on the DADSMARG macro may be overridden at execution time using the corresponding keywords on the DADSM macro.

The DFLARG macro is used to construct the DFLIBIN argument list. The DFLARG macro does not generate any instructions. The standard form of the DFLARG macro is written below.

```
[symbol]   DFLARG  [DSECT]  [,UNIT=number]  [,LEVELS=number]
                            [,UNIT=0     ]  [,LEVELS=1     ]
                  [,INDEX='indexname' ]  [,NAME='filename']
                  [,INDEX='USERLIB'   ]
```

DSECT
specifies that the actual DFLIBIN argument list should not be built. Instead, a DFLARG DSECT is generated.

UNIT=
specifies the unit number of the LIBIN unit.

LEVELS=
specifies the resolution limit of the '/INCLUDE' statements read from the file. If a LEVELS parameter is also specified on the /FILE statement for the LIBIN unit, the actual resolution limit will be the smaller of the two values. The default is 1.

INDEX=
specifies the index name of the file to be used as the library input file. It may be from 1 to 8 characters in length and enclosed in quotes. The default index name is USERLIB.

NAME=
specifies the file name of the file to be used as the library input file. It may be from 1 to 8 characters in length and enclosed in quotes.

Note:
All of the keywords on the DFLARG macro may be overridden at execution time using the corresponding keywords on the DFLIBIN macro.

DFTARG - DFTERMIN Argument List

The DFTARG macro may be used to define a termination line for the DFTERMIN macro. No instructions are generated by this macro. The standard form of the DFTARG macro is written below.

```
[symbol]   DFTARG  {'terminationline'}
```

'terminationline'

specifies a 1 to 8 character termination line which will be used to identify the last line to be placed in the terminal input stream. Data lines will be accepted from the user terminal and placed in the terminal input stream until the termination line is entered. The termination line must be enclosed in quotes.

DFTERMIN - Set Terminal Input Stream Parameters

The DFTERMIN macro is used to set, or change, the terminal input stream parameters of a conversational read unit. A conversational read unit is identified by the TERMIN parameter on a /FILE statement - the default for programs run at user terminals is unit 9. Data from a user terminal may be placed in the terminal input stream using the /CLIST command (see the VPS User's Guide) and/or after the program has begun execution by issuing the DFTERMIN macro to set the stream parameters and then reading from the conversational read unit (see the QGET macro). As long as data remains in the terminal input stream, consecutive reads of the conversational read unit will present consecutive lines of data from the terminal input stream to the user program without interaction with the terminal user. The standard form of the DFTERMIN macro is written below.

```
[symbol]   DFTERMIN  [UNIT=number ]  [,COUNT=number          ]
                     [UNIT=(reg)   ]  [,COUNT=(reg)          ]
                     [UNIT=9       ]  [,TLINE='terminationline']
                                      [,TLINE='/END'         ]
                                      [,DFTARG=addr          ]
                                      [,DFTARG=(reg)         ]
```

UNIT=

specifies the conversational read unit number. It may be coded as either a number or non-relocatable symbol or as a register containing the unit number. If a register is specified, it must be enclosed in parentheses. The default is 9.

COUNT=

specifies the number of lines which are to be subsequently read from the user terminal and placed in the terminal input stream. It may be coded as either a number or non-relocatable symbol or as a register containing the number. If a register is specified, it must be enclosed in parentheses.

TLINE=

specifies a 1 to 8 character termination line which will be used to identify the last line to be placed in the terminal input stream. Data lines will be accepted from the user terminal and placed in the terminal input stream until the termination line is entered. The termination line must be enclosed in quotes. If DFTARG is coded the TLINE parameter will replace the termination line specified in the DFTERMIN argument list (see below). If DFTARG is not coded, the argument list will be built in-line.

DFTARG=

specifies the address of an DFTERMIN argument list containing the termination line. It may be coded as either an address or as a register containing the address of the DFTERMIN argument list. If a register is specified, it must be enclosed in parentheses. The DFTARG macro should be used to build a DFTERMIN argument list.

Notes:

COUNT is mutually exclusive of TLINE and DFTARG. That is, the last line to be placed in the terminal input stream is either after the denoted number of lines are read from the user terminal (COUNT) or when the termination line is received (TLINE and/or DFTARG). If none of the three parameters are coded on an DFTERMIN macro, the control program will use - /END - as the termination line.

The DIO macro instruction can be used to read, write, or search for information in a direct access disk file (DSORG=DA is coded on the /FILE statement). The operation(s) performed on the file depend upon the information specified in operands on the DIO macro and in the DIO Request Block(s) which are built with the DIORB macro. DIORBs may be chained together to perform multiple operations on a file with one issuance of the DIO macro. (See the description of the DIORB macro for further information on chaining DIORBs.) Note that the DIO macro alters the contents of register 15. The standard form of the DIO macro is written below.

```
[symbol]   DIO   [+0]  {,RB=addr }  [,UNIT=number ]  [,BUFAD=addr ]
                 [+number] {,RB=(reg)}  [,UNIT=(reg)  ]  [,BUFAD=(reg)]
                                        [,SYSLIB=addr ]
                                        [,SYSLIB=(reg)]

                 [,BUFLEN=number]  [,SCOPE=number]  [,BLKNO=number]
                 [,BUFLEN=(reg) ]  [,SCOPE=(reg) ]  [,BLKNO=(reg) ]

                 [,OPT=(READ [,COUNT] [,KEY])                      ]
                 [,OPT=WRITE                                       ]
                 [,OPT=([SKEYE] [,SKEYH] [,SKEYL] [,READ] [,WRITE])]

                 [,SVC=YES]
                 [,SVC=NO ]
```

first positional parameter
: specifies the DIORB, offset from the beginning of the DIORB chain (specified by the RB= operand), which is to be modified by the information specified on this DIO macro call. It is coded as a plus sign (+) followed by a number or non-relocatable symbol. +0 is the default denoting the first or only DIORB.

RB=
: specifies the address of the first or only DIO Request Block. It may be coded either as an address or as a register, enclosed in parentheses, containing the address of the DIORB.

UNIT=
: specifies the unit number defining the file which is to be processed. It may be specified either as a number or non-relocatable symbol or as a register. If a register is specified, it must be enclosed in parentheses. If this operand is coded, it will replace the unit number specified in the DIORB. This parameter can be specified only when referencing the first or only DIORB in a chain (e.g. first positional parameter is +0). Note, UNIT=(1) may not be coded.

SYSLIB=
: specifies the address of either a fullword containing the unit

number defining the file to be processed or a doubleword containing a system library name as built by the SYSLIB macro instruction. It may be coded either as an address or as a register, enclosed in parentheses, containing the address of the area. If this operand is coded, it will replace the unit number specified in the DIORB. Note, SYSLIB=(1) may not be coded.

BUFAD=

specifies the address of either an area into which a block is to be placed on a read or search operation; or an area containing a block which is to be written on a write operation. It may be specified either as an address or as a register, enclosed in parentheses, containing the address of the area. If this operand is coded, it will replace the 'bufad' specified in the DIORB. Note, BUFAD=(1) may not be coded.

BUFLEN=

specifies the length of the 'bufad' area. It may be specified either as a number or non-relocatable symbol or as a register containing the length. If a register is specified, it must be enclosed in parentheses. If this operand is coded, it will replace the 'buflen' specified in the DIORB. Note, BUFLEN=(1) may not be coded.

BLKNO=

specifies the starting block number for a search, read, or write operation. It may be specified either as a number or non-relocatable symbol or as a register containing the block number. If a register is specified, it must be enclosed in parentheses. If this operand is coded, it will replace the 'blkno' specified in the DIORB. Note, BLKNO=(1) may not be coded.

SCOPE=

specifies the number of blocks to search when performing a key search. It may be specified either as a number or non-relocatable symbol or as a register containing the number. If a register is specified, it must be enclosed in parentheses. If this operand is coded, it will replace the 'scope' specified in the DIORB. Note, SCOPE=(1) may not be coded.

OPT=

specifies the operation to be performed.

WRITE -

specifies that the contents of the 'bufad' area are to be written into the file. If the file is being written with keys, the area must contain the key followed by the block to be written. For files written without keys, the area contains only the block of data and will be written at the block number location specified by the BLKNO= operand.

READ [,COUNT] [,KEY] -

specifies that a read operation is to be performed. If READ is specified, and the file is being read with keys, the key of the

requested block must appear at the beginning of the 'bufad' area. Upon completion of the operation, the key will be immediately followed by the data block. For reading without keys, the block having the number specified by the BLKNO= operand will be placed in the 'bufad' area. If READ,COUNT or READ,COUNT,KEY is specified, the count, key, and data block will be placed in the 'bufad' area. The block read will be the block whose number corresponds to the BLKNO= operand. If READ,KEY is specified, the key and block portion of the record having the number specified in the BLKNO= operand will be placed in the area.

[SKEYE] [,SKEYH] [,SKEYL] [,READ] [,WRITE] -
specifies a search operation is to be performed. The key in the 'bufad' area is compared against the block keys using the comparison operand specified - SKEYE (equal), SKEYH (high), or SKEYL (low) - until a match is found. The matching block number will be returned in the first word of the DIORB header. (See the DIORB macro for information on DIORB lengths.) Note, that the comparison operands may be used in combination (e.g. SKEYE,SKEYH implies a match will be made when the block key is equal to or higher than the one specified in the 'bufad' area). The search starts at the block number specified by the BLKNO= operand and continues for the number of blocks specified in the SCOPE= operand. If READ is specified and a match is found, the data portion of the block will be placed in the 'bufad' area. If WRITE is specified and a match is found, the data portion of the block will be replaced with the contents of the 'bufad' area.

If this operand is coded, it will replace the options specified in the DIORB.

SVC=
specifies whether the DIO macro expansion is to include the DIO SVC to perform the operation. The default is YES. If NO is specified, the macro expansion will contain all the necessary instructions - except for the SVC. This form of the macro should be used when formatting chained DIORBs.

Notes:

In general, the amount of data read or written is dependent upon the length of the 'bufad' area specified by the BUFLEN= operand. For example, if OPT=(READ,COUNT,KEY) is specified, and the 'bufad' area is only large enough to contain the count and key; only the count and key will be placed in the area. Alternately, multiple block reads and writes may be performed by specifying an appropriately large 'bufad' area. Note also that when reading or writing with keys, the BUFLEN= operand should have a value equal to n*(key length + data length), where n is the number of blocks to be read or written.

For further information on direct access disk I/O refer to Chapter 2.

DIORB - Build A DIO Request Block

The DIORB macro builds a DIO Request Block which is used with the DIO macro to perform input/output operations on direct access disk files. DIO Request Blocks may be 'chained' together to perform multiple operations on a direct access file with one issuance of the DIO macro. Chained DIORBs must appear as successive statements in an assembler language program. The DIORB macro does not generate any instructions. The standard form of the DIORB macro is written below.

```
[symbol]   DIORB   [S     ] [,E] [,UNIT=number] [,BUFAD=addr]
                   [+     ] [,+] [,UNIT=0     ] [,BUFAD=0   ]
                   [DSECT ]

                   [,BUFLEN=number] [,SCOPE=number] [,BLKNO=number]
                   [,BUFLEN=0     ] [,SCOPE=0     ] [,BLKNO=0     ]

                   [,OPT=(READ [,COUNT] [,KEY])                    ]
                   [,OPT=WRITE                                     ]
                   [,OPT=([SKEYE] [,SKEYH] [,SKEYL] [,READ] [,WRITE])]
```

first positional parameter
: specifies the location of this DIORB in a chain of DIORBs. S is the default and indicates that this is the first or only DIORB in the chain - depending on the second positional parameter. A plus sign (+) indicates that this DIORB is a member of a DIORB chain - other than the first. If DSECT is specified, the actual DIO Request Block is not built; instead, a DIORB DSECT is generated.

second positional parameter
: specifies whether a chained DIORB follows the one currently being built. The default is E indicating that this is the last (or only) DIORB in the chain. A plus sign (+) indicates that another DIORB follows this one in the chain. Note that if the defaults are taken for the first and second positional parameters, the DIO Request Block is built as a single, unchained DIORB.

UNIT=
: specifies the unit number defining the file which is to be processed. It may be specified either as a number or as a non-relocatable symbol. UNIT can be specified only on the first DIORB macro of a chain or on a single, unchained DIORB.

BUFAD=
: specifies the address of either an area into which a block is to be placed on a read or search operation; or an area containing a block which is to be written on a write operation.

BUFLEN=
: specifies the length of the 'bufad' area. It may be specified either as a number or as a non-relocatable symbol.

BLKNO=
specifies the starting block number for a search, read, or write operation. It may be specified either as a number or as a non-relocatable symbol.

SCOPE=
specifies the number of blocks to search when performing a key search. It may be specified either as a number or as a non-relocatable symbol.

OPT=
specifies the operation to be performed.

WRITE -
specifies that the contents of the 'bufad' area are to be written into the file. If the file is being written with keys, the area must contain the key followed by the block to be written. For files written without keys, the area contains only the block of data and will be written at the block number location specified by the BLKNO= operand.

READ [,COUNT] [,KEY] -
specifies that a read operation is to be performed. If READ is specified, and the file is being read with keys, the key of the requested block must appear at the beginning of the 'bufad' area. Upon completion of the operation, the key will be immediately followed by the data block. For reading without keys, the block having the number specified by the BLKNO= operand will be placed in the 'bufad' area. If READ,COUNT or READ,COUNT,KEY is specified, the count, key, and data block will be placed in the 'bufad' area. The block read will be the block whose number corresponds to the BLKNO= operand. If READ,KEY is specified, the key and block portion of the record having the number specified in the BLKNO= operand will be placed in the area.

[SKEYE] [,SKEYH] [,SKEYL] [,READ] [,WRITE] -
specifies a search operation is to be performed. The key in the 'bufad' area is compared against the block keys using the comparison operand specified - SKEYE (equal), SKEYH (high), or SKEYL (low) - until a match is found. The matching block number is placed in the first word of the DIORB header (see the Notes section). Note, that the comparison operands may be used in combination (e.g. SKEYE,SKEYH implies a match will be made when the block key is equal to or higher than the one specified in the 'bufad' area). The search starts at the block number specified by the BLKNO= operand and continues for the number of blocks specified in the SCOPE= operand. If READ is specifed and a match is found, the data portion of the block will be placed in the 'bufad' area. If WRITE is specified and a match is found, the data portion of the block will be replaced with the contents of the 'bufad' area.

If this operand is coded, it will replace the options specified in

Notes:

All of the keywords on the DIORB may be overridden at execution time using the corresponding keywords on the DIO macro.

The length of the DIORB generated by this macro is dependent on its position in the chain. First or single DIORBs consist of a header (8 bytes) and a DIORB proper (16 bytes). Chain members consist of only a DIORB proper. The symbols produced by the DSECT option of this macro should be used to reference information in a DIORB.

DLYEXC - Delay Execution

The DLYEXC macro instruction is used to delay the execution of the next instruction for a specified number of seconds. DLYEXC generates a supervisor call. When the time interval has elapsed, execution continues at the instruction following the SVC. The standard form of the DLYEXC macro is written below.

```
[symbol]    DYLEXC  [SECS=number]
                    [SECS=(reg)]
                    [SECS=0    ]
```

SECS=

specifies the number of seconds execution is to be delayed. It can be coded either as a number or non-relocatable symbol, or as a specific register containing the number of seconds to delay execution. If a register is specified, it must be enclosed in parentheses. The register may be designated symbolically or with an absolute expression. If this keyword is omitted the default of SECS=0 is used and the execution of the macro is effectively a no-op.

The EXTIME macro is used to obtain the accumulated CPU time for the current batch job or terminal session. The CPU time is returned in 300ths of a second in register 0. The EXTIME macro instruction is coded as follows.

```
[symbol]    EXTIME
```

The LCLARG macro builds a LIBRCLS argument list. No instructions are generated by this macro. The standard form of the LCLARG macro is written below.

```
[symbol]   LCLARG   [DSECT]  [,TYPE=SAVE]  [,UNIT=number]
                               [,TYPE=RSAV]
                               [,TYPE=CLEAR]

                   [,INDEX='indexname']  [,NAME='filename']
                   [,INDEX='USERLIB']

                   [,ACMNAME=symbol]
```

DSECT
specifies that the actual LIBRCLS argument list should not be built. Instead, an LCLARG DSECT is generated.

TYPE=
specifies the type of LIBRCLS operation to be performed. SAVE indicates that the current contents of the output library file is to be saved as a permanent file. RSAV indicates that the current contents of the output library file is to replace an existing library file. CLEAR indicates that the contents of the output library file is to be cleared.

UNIT=
specifies the unit number of the LIBOUT file on which the LIBRCLS operation is to be performed.

INDEX=
(For SAVE or RSAV only)
specifies a 1 to 8 character index name under which the library output file is to be saved or the index name of the file which the library output file is to replace. The index name must be enclosed in quotes. The default index name is USERLIB.

NAME=
(For SAVE or RSAV only)
specifies a 1 to 8 character file name under which the library output file is to be saved or the file name of the file which the library output file is to replace. The file name must be enclosed in quotes.

ACMNAME=
(For SAVE or RSAV only)
specifies the symbol to be used in the label field of the Access Modifier Area of the argument list. This parameter should be specified only when the programmer wishes to specify something

other than the default file attributes for an output library file which is to be saved or is to replace an existing library file. The default file attributes for a saved file are the same as those in the user profile (see the VPS User's Guide). The default file attributes for a replacement file are the same as the file it is replacing. The symbol specified may be used to easily address the Access Modifier Area since the user's program is responsible for initializing the area. See the ACCMOD macro for information on initializing and modifying the area.

LIBARG - LIBRARY Argument List

The LIBARG macro builds the LIBRARY argument list. No instructions are generated by this macro. The standard form of the LIBARG macro is written below.

```
[symbol]   LIBARG   [DSECT]   [,TYPE=PURGE  ]   [,INDEX='indexname']
                              [,TYPE=RENAME ]   [,INDEX='USERLIB'  ]
                              [,TYPE=ACCESS ]

                    [,NAME='filename']   [,NEWINDX='indexname']
                                         [,NEWINDX='USERLIB' ]

                    [,NEWNAME='filename']   [,NEWACCT='account']

                    [,ACCMOD=addr]
```

DSECT
specifies that the actual LIBRARY argument list should not be built. Instead, a LIBARG DSECT is generated.

TYPE=
specifies the type of LIBRARY request. PURGE indicates that the file is to be purged from the VPS library. RENAME indicates that the library file is to be renamed. ACCESS indicates that the file attributes of the library file are are to be replaced by those in the Access Modifier Area (see below).

INDEX=
specifies the 1 to 8 character index name of the file being purged, renamed, or modified. The index name must be enclosed in quotes. The default is USERLIB.

NAME=
specifies the 1 to 8 character file name of the file being purged, renamed, or modified. The file name must be enclosed in quotes.

NEWINDX=
(RENAME only)
specifies the new index name of the renamed file. It must be from 1 to 8 characters in length and enclosed in quotes. The default is USERLIB.

NEWNAME=
(RENAME only)
specifies the new file name of the renamed file. It must be from 1 to 8 characters in length and enclosed in quotes.

NEWACCT=

(RENAME only)
specifies the new account of the renamed file. It must be from 1 to 8 characters in length and enclosed in quotes.

ACCMOD=
(For RENAME or ACCESS only)
specifies the address of the Access Modifier Area. This parameter need only be used when ACCESS is specified or when RENAME is specified and the programmer wishes to also change the file attributes of the file being renamed. The Access Modifier Area should be built using the ACCMOD macro.

Note:
All of the keywords on the LIBARG macro may be overridden at execution time using the corresponding keywords on the LIBRARY macro.

LIBRARY - Library File Management

The LIBRARY macro may be used to purge, rename, or change the file attributes of an existing library file. The standard form of the LIBRARY macro is written below.

```
[symbol]   LIBRARY  [PURGE  ] {,LIBARG=addr  } [,INDEX=addr          ]
                    [RENAME ] {,LIBARG=(reg)} [,INDEX=(reg)         ]
                    [ACCESS ]                 [,INDEX='indexname'   ]

                    [,NAME=addr          ] [,NEWINDX=addr          ]
                    [,NAME=(reg)         ] [,NEWINDX=(reg)         ]
                    [,NAME='filename'    ] [,NEWINDX='indexname'   ]

                    [,NEWNAME=addr       ] [,NEWACCT=addr          ]
                    [,NEWNAME=(reg)      ] [,NEWACCT=(reg)         ]
                    [,NEWNAME='filename' ] [,NEWACCT='account'     ]

                    [ACCMOD=addr ]
                    [ACCMOD=(reg)]
```

PURGE
: is written as shown and specifies that the library file is to be purged from the VPS library. Note that this parameter will override the TYPE parameter of the LIBRARY argument list.

RENAME
: is written as shown and specifies that the library file is to be renamed. Note that this parameter will override the TYPE parameter of the LIBRARY argument list.

ACCESS
: is written as shown and specifies that the file attributes of the library file are to be replaced by those specified in the Access Modifier Area (see below). Note that this parameter will override the TYPE parameter of the LIBRARY argument list.

LIBARG=
: specifies the address of the LIBRARY argument list. It may be specified either as an address or as a register containing the address of the argument list. If a register is coded, it must be enclosed in parentheses. The LIBRARY argument list should be built using the LIBARG macro.

INDEX=
: specifies the index name of the file to be purged, renamed, or modified. It may be specified as: an address of an 8-byte area containing the index name; a register holding the address of an 8-byte area containing the index name; or the actual index name enclosed in quotes. If this parameter is specified, it will replace the index name field of the LIBRARY argument list. Note,

INDEX=(1) may not be coded.

NAME=

specifies the file name of the library file to be purged, renamed, or modified. It may be specified as: an address of an 8-byte area containing the file name; a register holding the address of an 8-byte area containing the file name; or the actual file name enclosed in quotes. If this parameter is specified, it will replace the file name field of the LIBRARY argument list. Note, NAME=(1) may not be coded.

NEWINDX=

(RENAME only)
specifies the new index name which the library file is to be renamed to. It may be specified as: an address of an 8-byte area containing the new index name; a register holding the address of an 8-byte area containing the new index name; or the actual new index name enclosed in quotes. If this parameter is specified, it will replace the new index name field of the LIBRARY argument list. Note, NEWINDX=(1) may not be coded.

NEWNAME=

(RENAME only)
specifies the new file name which the library file is to be renamed to. It may be specified as: an address of an 8-byte area containing the new file name; a register holding the address of an 8-byte area containing the new file name; or the actual new file name enclosed in quotes. It this parameter is specified, it will replace the new file name field of the LIBRARY argument list. Note, NEWNAME=(1) may not be coded.

NEWACCT=

(RENAME only)
specifies the new account for the renamed library file. It may be specified as: an address of an 8-byte area containing the new account; a register holding the address of an 8-byte area containing the new account; or the actual new account enclosed in quotes. If this parameter is specified, it will replace the new account field of the LIBRARY argument list. The unprivileged user may specify only her/his own account for this parameter. Note, NEWACCT=(1) may not be coded.

ACCMOD=

(For RENAME or ACCESS only)
specifies the address of the Access Modifier Area. It may be specified either as an address or as a register containing the address of the area. If a register is coded, it must be enclosed in parentheses. The Access Modifier Area need only be supplied when ACCESS is specified or when RENAME is specified and the programmer wishes to also change the file attributes of the file being renamed. The Access Modifier Area should be built using the ACCMOD macro. If this parameter is specified, it will replace the address of the Access Modifier Area specified in the LIBRARY argument list. Note, ACCMOD=(1) may not be coded.

Note:
When control is returned to the program register 15 will contain one of the following return codes:

Hexadecimal Code	Meaning
0	LIBRARY operation successfully completed.
4	Not enough core available.
8	I/O error.
C	Access not allowed.
10	Index not found.
14	File not found.
18	Invalid library file name.
1C	Account over library limit.
20	Invalid new library file name (RENAME).
24	Name already in use.
28	No space in index.

The LIBRCLS macro may be used to close an output library file with one of the following options: save the current contents of the output library file as a permanent library file; replace an existing library file with the current contents of the output library file; or clear the current contents of the output library file. An output library file is identified as a unit having LIBOUT specified in the UNIT keyword of the /FILE statement (see the VPS User's Guide). The standard form of the LIBRCLS macro is written below.

```
[symbol]   LIBRCLS  [SAVE   ] {,LCLARG=addr  } [,UNIT=number]
                    [RSAV   ] {,LCLARG=(reg)} [,UNIT=(reg)  ]
                    [CLEAR  ]

                    [,INDEX=addr          ] [,NAME=addr          ]
                    [,INDEX=(reg)         ] [,NAME=(reg)         ]
                    [,INDEX='indexname'   ] [,NAME='filename'    ]
```

SAVE
: is written as shown and specifies that the current contents of the library output file is to be saved as a permanent library file. If this parameter is coded, it will override the TYPE parameter specified in the LIBRCLS argument list.

RSAV
: is written as shown and specifies that the current contents of the output library file is to replace an existing library file. If this parameter is coded, it will override the TYPE parameter specified in the LIBRCLS argument list.

CLEAR
: is written as shown and specifies that the current contents of the output library file is to be cleared. If this parameter is coded, it will override the TYPE parameter specified in the LIBRCLS argument list.

LCLARG=
: specifies the address of an area containing the LIBRCLS argument list. It may be coded as an address or as a register containing the address of the argument list. If a register is specified, it must be enclosed in parentheses. The LIBRCLS argument list should be built using the LCLARG macro.

UNIT=
: specifies the unit number of a LIBOUT unit on which the LIBRCLS operation is to be performed. It may be coded either as a number or non-relocatable symbol or as a register containing the unit number. If a register is specified, it must be enclosed in parentheses. If this parameter is coded, it will override the unit number specified in the LIBRCLS argument list. Note,

UNIT=(1) may not be coded.

INDEX=

(For SAVE and RSAV only)
specifies the index name under which the library output file is to be saved or the index name of the file which the output library file is to replace. It may be specified as: an address of an 8-byte area containing the index name; a register holding the address of an 8-byte area containing the index name; or the actual index name enclosed in quotes. If this parameter is specified, it will replace the index name specified in the LIBRCLS argument list. Note, INDEX=(1) may not be coded.

NAME=

(For SAVE and RSAV only)
specifies the file name under which the library output file is to be saved or the file name of the file which the output library file is to replace. It may be specified as: an address of an 8-byte area containing the file name; a register holding the address of an 8-byte area containing the file name; or the actual file name enclosed in qoutes. If this parameter is specified, it will replace the file name specified in the LIBRCLS argument list. Note, NAME=(1) may not be coded.

Notes:

File attributes may be specified for saved and replaced library files through the use of the LCLARG and ACCMOD macros.

When control is returned to the program register 15 will contain one of the following return codes:

Hexadecimal

Code	Meaning
0	LIBRCLS operation successfully completed.
4	Argument invalid.
8	Unit number invalid.
C	Library output file empty.
10	Unable to obtain core.
14	I/O error.
18	Access to file not allowed.
1C	Index not found.
20	Name already in use (SAVE).
24	Invalid file name.
28	Account over library limit.
2C	No space in index.

The LMARG macro constructs an argument list for either the LMBLDL or the LMLOAD macro instruction. The LMARG macro does not generate any instructions. The LMARG macro instruction is written as follows:

```
[symbol]   LMARG   [DIR   ][,MODNAME=name]   [,UNIT=number  ]
                   [BLDL  ]                   [,SYSLIB=addr ]
                   [LOAD  ]
```

BLDL
is written as shown and specifies that the macro is to construct an argument list for the LMBLDL macro instruction. Note, if BLDL is specified, MODNAME may not be coded.

DIR
is written as shown and specifies that the macro is to construct an argument list for the LMLOAD macro instruction containing a 32-byte area into which the programmer must move a valid directory entry before execution of the LMLOAD macro instruction. (See the LMBLIST macro for more information on moving the directory entry to the argument list.) Note, if DIR is specified, MODNAME may not be coded.

LOAD
is written as shown and specifies that the macro is to construct an argument list for the LMLOAD macro instruction indicating that the load module search is by load module name. LOAD is the default.

MODNAME=
specifies a 1 to 8 character alphanumeric load module name which the macro will place in the LMLOAD argument list. If MODNAME is not specified the field will be initialized to blanks. Note that the keyword may be overridden at execution time by the corresponding keyword on the LMLOAD macro instruction.

UNIT=
specifies the work file unit number the control program is to search for the directory entry (if the argument list is being constructed for the LMBLDL macro instruction) or the actual load module (if the argument list is being constructed for the LMLOAD macro instruction). Note that the keyword may be overridden at execution time by the corresponding keyword on the LMBLDL or LMLOAD macro instruction.

SYSLIB=
specifies the address of a list of libraries and/or work file units the control program is to search. The list is built using the SYSLIB macro instruction. Note that the keyword may be

overridden at execution time by the corresponding keyword on the LMBLDL or LMLOAD macro instruction.

If neither UNIT nor SYSLIB are specified, the system link library (SYS.LINKLIB) will be searched - this also may be overridden at execution time by either the UNIT or SYSLIB keywords on the LMBLDL or LMLOAD macro instructions.

The LMBLDL macro instruction is used to read a directory entry of a load module, or list of load modules, into user storage. A directory entry may be read from the system link library (SYS.LINKLIB), one of the system subroutine libraries (e.g. SYS.FORTLIB, SYS.PLIXLIB, etc.), or a work file. The programmer can alternatively specify a list of system libraries and/or work file units which the control program will search as a concatenated list - loading the first occurrence of the directory entry. Note, the LMBLDL macro will destroy the contents of register 13. The standard form of the LMBLDL macro is written as follows.

```
[symbol]   LMBLDL  [LMARG=addr   ] {,ADDR=lmbldl list address}
                   [LMARG=(reg)  ] {,ADDR=(reg)              }

                   {,UNIT=number                             }
                   {,UNIT=(reg)                              }
                   {,SYSLIB=addr                             }
                   {,SYSLIB=(reg)                            }
                   {,SYSLIB=(LIST,sys library | unit,...)    }
```

LMARG=

specifies the address of an area containing the LMBLDL argument list. It may be specified either as an address or as a register. If a register is used, it must be enclosed in parentheses and the address in the register will be used as the address of the argument list. The LMBLDL argument list is built using the LMARG macro and contains request information. If LMARG is not specified, an in-line argument list is built. Note, LMARG=(1) may not be coded.

ADDR=

specifies the address of the LMBLDL list where the directory entry (or entries) is to be placed. The LMBLDL list is constructed using the LMBLIST macro. If (reg) is specified, the LMBLDL list address will be taken from the register specified. Note, ADDR=(1) may not be coded.

UNIT=

specifies a work file unit which the control program is to search for the directory entry of the load module specified. It may be coded either as a number or non-relocatable symbol or as a register. If a register is coded, it must be enclosed in parentheses and the value in that register will be used as the work file unit number.

SYSLIB=

specifies a list of libraries the control program is to search to find the load module directory entry. It may be coded either as an address or as a register (enclosed in parentheses) containing the address of a concatenation list. This list is built using the

SYSLIB macro instruction and can contain either system library names and/or work file unit numbers. An alternative form of this keyword may be used to code the actual list. If this form is used, the list must be enclosed in parentheses and the first subparameter must be LIST followed by a list of system library names and/or work file unit numbers - arranged in the order that they are to be searched. In this case the concatenation list will be built in-line.

Notes:

If neither the UNIT nor the SYSLIB keywords are coded, the system link library (SYS.LINKLIB) will be searched. When control is returned to the program, register 15 will contain one of the following return codes:

Hexadecimal

Code	Meaning
0	Directory entries loaded successfully.
4	Not all of the directory entries were found.
8	None of the directory entries were found.
10	I/O error reading directory entry.
14	Invalid argument.

LMBLIST - LMBLDL List

The LMBLIST macro is used to construct the LMBLDL directory entry list for the LMBLDL macro. The LMBLIST macro does not generate instructions. The LMBLIST macro instruction is written below.

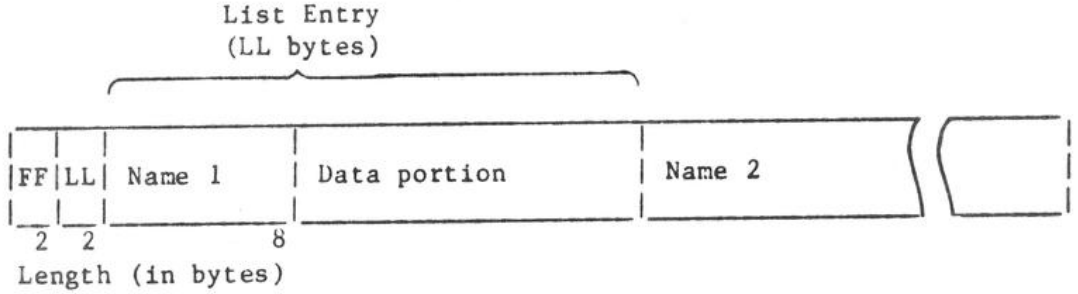
```
[symbol]   LMBLIST  {MODNAME=(load module name,load module name,...)}
                    {MODNAME=number                                 }

                    [ELEN=number]
                    [ELEN=32    ]
```

MODNAME=
specifies a list of load module names enclosed in parentheses. For each load module name the LMBLIST macro will generate a directory entry (or partial directory entry) consisting of an 8-byte field containing the load module name followed by the data portion of the directory entry to be filled in by the execution of the LMBLDL macro. If number is coded, that value of directory entries will be constructed with each name field initialized to blanks.

ELEN=
specifies the entry length in bytes to be used when constructing each directory entry. If ELEN is not coded - each directory entry will be 32 bytes - the maximum directory entry length. The minimum length is 9 bytes. Note, if the directory entry is to be used (after it has been completed by the LMBLDL macro) by a LMLOAD macro to load a load module - the maximum length must be specified (or defaulted to).

Notes:
The LMBLIST macro constructs a list structure of the following format:



where -
FF contains the number of entries in the list.
LL contains the length of each entry.

If after execution of the LMBLDL macro the programmer wishes to use a directory entry in the execution of the LMLOAD macro to load a load module, the specified directory entry must be moved to the LMLOAD argument list at offset +4 (the FF and LL fields should not be moved).

LMLOAD - Load A Load Module

The LMLOAD macro is used to load a load module into user storage. A load module may be loaded from the system link library (SYS.LINKLIB), one of the system subroutine libraries (e.g. SYS.FORTLIB, SYS.PLIXLIB, etc.), or a work file. The programmer can alternatively specify a list of system libraries and/or work file units which the control program will search as a concatenated list - loading the first occurrence of the load module. An alternative form of the LMLOAD macro allows the programmer to specify a directory entry (obtained using the LMBLDL macro) instead of the load module name thereby avoiding a search of the file directory. Upon completion of the macro, register 0 will contain the actual length of the load module and register 1 will contain the load module entry point address. NOTE, LMLOAD will destroy the contents of register 13. The standard form of the LMLOAD macro is written below.

```
[symbol]   LMLOAD   [DIR] [,LMARG=addr   ] {,ADDR=addr } {,LEN=addr }
                          [,LMARG=(reg)  ] {,ADDR=(reg)} {,LEN=(reg)}
                          [,MODNAME=name ]
                          [,MODLOC=addr  ]
                          [,MODLOC=(reg) ]

                          [,UNIT=number                              ]
                          [,UNIT=(reg)                               ]
                          [,SYSLIB=addr                              ]
                          [,SYSLIB=(reg)                             ]
                          [,SYSLIB=(LIST,sys library | unit,...)     ]
```

DIR

is written as shown. If DIR is coded, a valid directory entry must be included in the LMLOAD parameter list (see the macros LMBLDL and LMARG) and the MODNAME and SYSLIB parameters may not be specified. If DIR is not specified, library directory or directories will be searched for the load module.

LMARG=

specifies the address of an area containing the LMLOAD argument list. It may be specified either as an address or as a register. If a register is coded it must be enclosed in parentheses and the address in the register will be used. The LMLOAD argument list is built using the LMARG macro and contains request information, a load module name and, if DIR is specifed, a directory entry (see the LMBLIST macro for information on moving the directory entry to the LMLOAD argument list). If LMARG is not specified, an in-line parameter list is built. Note, LMARG=(1) may not be coded.

MODNAME=

specifies a 1 to 8 character alphanumeric load module name which the control program is to load into user storage. If the MODNAME keyword is coded and the LMARG keyword is also coded, the module name specified replaces the one in the LMLOAD parameter list.

MODLOC=
specifies the address of an 8 byte area containing the load module name left-justified and padded to the right with blanks. It may be coded either as an address or as a register. If a register is coded, it must be enclosed in parentheses and the address in the register specified will be used.

ADDR=
specifies the address of the area into which the load module is to be loaded. It may be coded either as a relocatable or non-relocatable symbol or as a register. If a register is specified, it must be enclosed in parentheses and the address in the register is used. Note, ADDR=(1) may not be coded.

LEN=
specifies the length of the area into which the load module is to be loaded. It may be coded either as a number or non-relocatable symbol or as a register. If a register is specified, it must be enclosed in parentheses and the value in the register will be used as the length. Note, LEN=(1) may not be coded.

UNIT=
specifies a work file unit which the control program is to search for the load module. It may be coded either as a number or non-relocatable symbol or as a register. If a register is specified, it must be enclosed in parentheses and the value in the register will be used as the unit number.

SYSLIB=
specifies a list of libraries the control program is to search to find the load module. It may be coded either as an address or as a register (enclosed in parentheses) containing the address of a concatenation list. This list is built using the SYSLIB macro instruction and can contain either system library names and/or work file unit numbers. An alternative form of this keyword may be used to code the actual list. If this form is used, the list must be enclosed in parentheses and the first subparameter in the list must be LIST followed by a list of system library names and/or work file unit numbers - arranged in the order they should be searched. In this case the concatenation list will be built in-line.

Notes:
If neither the UNIT nor the SYSLIB keywords are coded, the system link library (SYS.LINKLIB) will be searched. When control is returned to the program, register 15 will contain one of the following return codes:

Hexadecimal

Code	Meaning
0	Module loaded successfully.
4	Module name not found.

8	Module found but could not be relocated.
C	Area length too small.
10	I/O error while reading load module from disk.
14	Invalid load argument or load module.

The MESGB macro is used to build a pattern which when referenced by the BLDMSG macro constructs a variable length message. The MESGB macro does not generate any instructions. The standard form of the MESGB macro is written below.

```
[symbol]   MESGB  {'message'                    }   ,...
                  |DATE                        |
                  |TIME                        |
                  |TVAL                        |
                  |CPUTM                       |
                  |(DEC,length[,L])            |
                  |(FDEC,length,column[,L])    |
                  |(HEX,length)                |
                  |$VAL                        |
                  |ACCT                        |
                  |(CHAR,length)               |
                  |(FCHAR,length)              |
                  |VMSG                        |
                  |FVMSG                       |
                  |(ATAB,location)             |
                  |(RTAB,location)             |
                  {                            }
```

'message'
specifies a message (or portion of a message) and must be enclosed in quotes.

DATE
specifies that the next 16 bytes of the message are to contain the current date in the format:

```
www_mmm_dd,_yyyy
  where,
    ww - is the weekday - abbreviated.
   mmm - is the month - abbreviated if necessary.
    dd - is the day of the month.
  yyyy - is the year.
```

The control program will automatically convert the current date to this format and place it in the message when the BLDMSG macro is executed referencing this pattern.

TIME
specifies that the next 8 bytes of the message are to contain the current time in the format:

```
hh:mm:ss
  where,
  hh - is the hour (0-24).
  mm - is the minute.
  ss - is the second.
```

The control program will automatically convert the current time (the time at execution of the BLDMSG macro) to this format and place it in the message when the BLDMSG macro is executed.

TVAL

specifies that the next data address in the BLDMSG data address list is the address of a fullword containing a timer value in 300ths of a second units. The control program will convert the value automatically to the format - hh:mm:ss - placing the significant digits starting at the next location in the message buffer.

CPUTM

specifies that the next data address in the BLDMSG data list is the address of a fullword containing CPU timer units (1 unit = 4096 microseconds) which is to be converted to the format:

ss.hh
where,
ss - is seconds.
hh - is hundredths of seconds.

The siginificant digits of which will be placed starting at the next location of the message buffer.

(DEC,length[,L])

specifies that the next data address in the BLDMSG data address list is the address of a binary number which is to be converted to decimal and the significant digits placed starting at the next location in the message buffer (at execution of the BLDMSG macro). The length of the data in bytes is specified by length. The length must be between 1 and 4, and the default is 4. If L is specified the data will be taken as an unsigned binary number. If L is not coded and the data is negative, a minus sign will precede the converted number in the message buffer.

(FDEC,length,column[,L])

specifies that the next data address in the BLDMSG data address list is the address of a binary number which is to be converted to decimal and placed in the next location in the message buffer. The length of the data is specified by length. The length must be between 1 and 4, and 4 is the default. The width of the converted field in the message buffer is specified by column. The field will be right justified and padded to the left with blanks. The default for column is 10. If L is specified the data will be taken as an unsigned binary number. If L is not coded and the data is negative, a minus sign will precede the converted number in the message buffer.

(HEX,length)

specifies that the next data address in the BLDMSG data address list is the address of a unsigned binary number which is to be converted to a printable hexadecimal number and placed starting at

the next location in the message buffer. The length of the data in bytes is specified by length. The length must be between 1 and 255.

$VAL
specifies that the next data address in the BLDMSG data address list is the address of a fullword containing a value in cents which the control program will convert to dollars and cents and place in the next location of the message buffer in the format:

$dd...d.cc

Leading zeros will be suppressed.

ACCT
specifies that the account number is to be moved to the next locations in the message buffer. Trailing blanks will be truncated.

(CHAR,length)
specifies that the next data address in the BLDMSG data address list points to a character string which is to be moved to the next location in the message buffer. Length specifies the number of bytes in the string. Leading and trailing blanks will not be moved.

(FCHAR,length)
specifies that the next data address in the BLDMSG data address list points to a character string which is to be moved to the next location in the message buffer. Length specifies the number of bytes to be moved. Leading and trailing blanks will be moved.

VMSG
specifies that the next data address in the BLDMSG data list is the address of a standard v-format message (e.g. built using the MESGV macro), and that it is to be placed in the message buffer starting at the next location. Leading and trailing blanks will not be moved.

FVMSG
specifies that the next data address in the BLDMSG data list is the address of a standard v-format message, and that it is to be placed in the message buffer starting at the next location. Leading and trailing blanks will be moved.

(ATAB,location)
specifies the next message text location in the buffer relative to the beginning of the message (0). This operand provides an absolute 'tab' facility during messsage generation.

(RTAB,location)

specifies the next message text location in the buffer relative to the current location. The location field must be a number between -128 and 127. This operand provides a relative 'tab' facility during message generation.

Notes:

The MESGB macro allows multiple operands. Any of the MESGB operands - 'message', DATE, TIME, etc. - may appear more than once. Care should be taken in building the data address list for the BLDMSG macro since the order and number of data address entries is controlled by the operands in the MESGB macro expansion.

MESGV - Generate A Variable Length Message

The MESGV macro is used to build a variable length message which is to be sent using the SMSG macro or written using the QIO access method. The MESGV macro does not generate any instructions. The standard form of the MESGV macro is written below.

```
[symbol]    MESGV   {[symbol],'message'}  ...  [,symbol],'message'
                    {[symbol],(length) }       [,symbol],(length)
```

'message'
specifies a message (or portion of a message) and must be enclosed in quotes.

(length)
specifies that the MESGV macro is to initialize an area at the current location in the message to blanks for a length of (length) in bytes.

symbol
specifies a relocatable symbol the MESGV macro is to assign to the next byte location of the message being built. The symbol will have a length attribute of (length) or the number of characters in 'message', whichever it precedes.

Notes:
The MESGV macro allows multiple operands - message text operands and (length) operands may appear more than once. Symbols preceding any of the operands must be unique. The symbol operand allows the programmer addressability to field(s) in the message which are to be dynamically created or modified during the execution of the program.

PSTCOD - Retrieve Post Code

The PSTCOD macro instruction is used to retrieve the post code from the JCB and zero the field. The post code is set by means of the /POST command entered from the terminal during execution of the program. The /POST command is entered as follows:

/POST code

where code is a zero to 8 byte alphanumeric character string.

Note that since the execution of the PSTCOD macro zeroes the field, a unique post code may be retrieved only once. The standard form of the PSTCOD macro instruction is written below.

[symbol] PSTCOD $\left\{\begin{matrix}\text{addr}\\(\text{reg})\end{matrix}\right\}$

addr

is the address of an 8 byte area in which the post code will be returned. Upon completion of the macro the 8 byte area will contain either the post code - left justified and padded to the right with blanks; or 8 bytes of hexadecimal zeroes indicating that no /POST command has been entered at all or no /POST command has been entered since the last execution of the PSTCOD macro. If (reg) is specified, the address in the designated register is used as the post code return area.

QBSP - Backspace File (QIO)

The QBSP macro instruction causes a file to be backspaced. If a file is being read, a backspace operation will reposition the file so that a subsequent sequential read will read the same record as the last read previous to the backspace. If a file is being written, a backspace operation will cause the next sequential write to write a record in the file at the same location as the last write previous to the backspace. The file must be a tape or direct access file. The standard form of the QBSP macro is written below.

```
[symbol]   QBSP  [unit      ]   [,RB=addr  ] [,SYNAD=addr  ]
                 [(reg1)     ]   [,RB=(reg) ] [,SYNAD=(reg) ]
                 ['ddname'   ]
                 [UTYPE=DISK ]
                 [UTYPE=TAPE ]
                 [,SVC=YES   ]
                 [,SVC=NO    ]
```

unit
: specifies the unit number defining the file which is to be backspaced. It may be specified either as a number or as a non-relocatable symbol. If this operand is coded, it will replace the unit number specified in the QIORB.

(reg1)
: specifies a register containing the unit number of the file which is to be backspaced. If this operand is coded, it will replace the unit number specified in the QIORB.

'ddname'
: specifies a ddname (see the VPS User's Guide for a description of the ddname parameter on the /FILE statement) defining the file which is to be backspaced. If this operand is coded, it will replace the ddname specified in the QIORB.

UTYPE=
: specifies the unit type (see the VPS User's Guide for a description of the unit type parameter on the /FILE statement) defining the file which is to be backspaced. If this operand is coded, it will replace the unit type specified in the QIORB. Note that the control program will search (in unit number order) the /FILE statements and choose the first occurrence of the specified unit type.

RB=
: specifies the address of a QIO Request Block (the QIORB can be built with the QIORB macro). It may be coded either as an address or as a register, enclosed in parentheses, containing the address of the QIORB. If this operand is not coded, the QBSP macro will

build the QIORB as part of the inline expansion of the macro.

SYNAD=

specifies the address of a routine which is to be given control if the backspace operation could not be completed successfully. It may be specified either as an address or as a register, enclosed in parentheses, containing the address of the routine. If this operand is coded, it will replace the address specified in the QIORB.

SVC=

specifies whether the QBSP macro expansion is to include the QIO SVC to perform the backspace operation. The default is YES. If NO is specified, the macro expansion will contain all the necessary instructions - except for the SVC. This form of the macro can be used to format the QIORB.

QEOF - Write EOF (QIO)

The QEOF macro causes an end-of-file (EOF) record to be written at the next record location in a file. The file must be a tape or direct access file being used for output - records are being written into the file. If the file has been used as an input file, issuing the QEOF macro on the file will cause an error. The standard form of the QEOF macro is written below.

```
[symbol]   QEOF  [unit      ]   [,RB=addr  ] [,SYNAD=addr  ]
                 [(reg1)    ]   [,RB=(reg) ] [,SYNAD=(reg) ]
                 ['ddname'  ]
                 [UTYPE=DISK]
                 [UTYPE=TAPE]
                 [,SVC=YES  ]
                 [,SVC=NO   ]
```

unit
: specifies the unit number defining the file into which the EOF record is to be written. It may be specified either as a number or as a non-relocatable symbol. If this operand is coded, it will replace the unit number specified in the QIORB.

(reg1)
: specifies a register containing the unit number of the file into which the EOF record is to be written. If this operand is coded, it will replace the unit number specified in the QIORB.

'ddname'
: specifies a ddname (see the VPS User's Guide for a description of the ddname parameter on the /FILE statement) defining the file into which the EOF record is to be written. If this operand is coded, it will replace the ddname specified in the QIORB.

UTYPE=
: specifies the unit type (see the VPS User's Guide for a description of the unit type parameter on the /FILE statement) defining the file into which the EOF record is to be written. If this operand is coded, it will replace the unit type specified in the QIORB. Note that the control program will search (in unit number order) the /FILE statements and choose the first occurrence of the specified unit type.

RB=
: specifies the address of a QIO Request Block (the QIORB can be built with the QIORB macro). It may be coded either as an address or as a register, enclosed in parentheses, containing the address of the QIORB. If this operand is not coded, the QEOF macro will build the QIORB as part of the inline expansion of the macro.

SYNAD=
specifies the address of a routine which is to be given control if the EOF operation could not be completed successfully. It may be specified either as an address or as a register, enclosed in parentheses, containing the address of the routine. If this operand is coded, it will replace the address specified in the QIORB.

SVC=
specifies whether the QEOF macro expansion is to include the QIO SVC to perform the EOF operation. The default is YES. If NO is specified, the macro expansion will contain all the necessary instructions - except for the SVC. This form of the macro can be used to format the QIORB.

The QGET macro instruction causes a record to be read from a file and placed in a designated area of user storage. The location of the record in the file which is to be read depends upon the physical structure of the file and information specified in the QIO Request Block (QIORB). If the file is being read sequentially, the next logical record will be placed in user storage. If the file is a direct access file and a record number is specified, the record placed in the user storage area will depend upon its physical location in the file (if absolute addressing is used) or location from the last record read (if relative addressing is used). The standard form of the QGET macro is written below.

```
[symbol]   QGET  [addr    ] [,length  ] [,unit          ]   [,RB=addr  ]
                 [(reg1)  ] [,(reg2)  ] [,(reg3)        ]   [,RB=(reg) ]
                                        [,'ddname'      ]
                                        [,UTYPE=DISK    ]
                                        [,UTYPE=LIBIN   ]
                                        [,UTYPE=LIBOUT  ]
                                        [,UTYPE=SYSIN   ]
                                        [,UTYPE=SYSOUT  ]
                                        [,UTYPE=TAPE    ]
                                        [,UTYPE=TERMIN  ]
                                        [,UTYPE=TERMOUT ]

                 [,RECFM=F  ] [,EODAD=addr  ] [,SYNAD=addr  ]
                 [,RECFM=V  ] [,EODAD=(reg) ] [,SYNAD=(reg) ]

                 [,RECNO=number ] [,RNTYPE=ABS ] [,SVC=YES ]
                 [,RECNO=(reg)  ] [,RNTYPE=REL ] [,SVC=NO  ]
```

addr
: specifies the address of an area in user storage into which the record is to be read. If this operand is coded, it will replace the 'bufad' field in the QIORB.

(reg1)
: specifies a register containing the address of an area in user storage into which the record is to be read. If this operand is coded, it will replace the 'bufad' field in the QIORB.

length
: specifies the length of the area into which the record is to be read. It may be specified as either a number or a non-relocatable symbol. If this operand is coded, it will replace the 'buflen' field specified in the QIORB.

(reg2)
: specifies a register containing the length of the area into which the record is to be read. If this operand is coded, it will replace the 'buflen' field specified in the QIORB.

unit
specifies the unit number defining the file from which the record is to be read. It may be specified either as a number or as a non-relocatable symbol. If this operand is coded, it will replace the unit number specified in the QIORB.

(reg3)
specifies a register containing the unit number from which the record is to be read. If this operand is coded, it will replace the unit number specified in the QIORB.

'ddname'
specifies a ddname (see the VPS User's Guide for a description of the ddname parameter on the /FILE statement) defining a file from which the record is to be read. If this operand is coded, it will replace the ddname specified in the QIORB.

UTYPE=
specifies the unit type (see the VPS User's Guide for a description of the unit type parameter on the /FILE statement) defining a file from which the record is to be read. If this operand is coded, it will replace the unit type specified in the QIORB. Note that the control program will search (in unit number order) the /FILE statements and choose the first occurrence of the specified unit type.

RB=
specifies the address of a QIO Request block (the QIORB can be built with the QIORB macro). It may be coded either as an address or as a register, enclosed in parentheses, containing the address of the QIORB. If this operand is not coded, the QGET macro will build the QIORB as part of the inline expansion of the macro.

RECFM=
specifies the format of the record as it is to appear in user storage. If F (fixed) is specified, the record will appear left-justified in the user storage area, truncated or padded with blanks as necessary. If V (variable) is specified, the user area will contain a standard v-format record (see Chapter 2 for a description of a v-format record). If this operand is coded, it will replace the record format specified in the QIORB.

EODAD=
specifies the address of a routine which is to be given control if the end-of-file is reached. It may be coded either as an address or as a register, enclosed in parentheses, containing the address of the routine. If this operand is coded, it will replace the address specified in the QIORB.

SYNAD=

specifies the address of a routine which is to be given control if the read operation could not be completed successfully. It may be specified either as an address or as a register, enclosed in parentheses, containing the address of the routine. If this operand is coded, it will replace the address specifed in the QIORB.

RECNO=

specifies either the absolute or relative record number, depending upon the RNTYPE specified in the QIORB, of the record to be read from the file. It may be coded as a number or a non-relocatable symbol or as a register, enclosed in parentheses, containing the record number. This operand can be coded only when reading direct access files (i.e. DSORG=DA appears on the associated /FILE statement) and reading tapes with relative record addressing.

RNTYPE=

specifies whether the record number in the RECNO keyword is an absolute record number (relative to the beginning of the file - the first record being record number 1) or a record number relative to the current position in the file. If this operand is coded, it will replace the RNTYPE field in the QIORB. This operand can be coded only when reading direct access files (i.e. DSORG=DA appears on the associated /FILE statement) and tapes with relative record addressing.

SVC=

specifies whether the QGET macro expansion is to include the QIO SVC to perform the read operation. The default is YES. If NO is specified, the macro expansion will contain all the necessary instructions - except for the SVC. This form of the macro can be used to format a QIORB.

QIORB - Build A QIO Request Block

The QIORB macro builds a QIO Request Block which is to be used with the QGET, QPUT, QEOF, QREW, and QBSP macros to perform input/output operations on files using the QIO access method. The QIORB macro does not generate any instructions. The standard form of the QIORB macro is written below.

```
[symbol]   QIORB  [DSECT]  [,BUFAD=addr]  [,BUFLEN=number]
                                           [,BUFLEN=0]

                  [,RECFM=F]  [,UNIT=number]          [,EODAD=addr]
                  [,RECFM=V]  [,UNIT=0]
                              [,DDNAME=ddname]
                              [,UTYPE=DISK]
                              [,UTYPE=LIBIN]
                              [,UTYPE=LIBOUT]
                              [,UTYPE=SYSIN]
                              [,UTYPE=SYSOUT]
                              [,UTYPE=TAPE]
                              [,UTYPE=TERMIN]
                              [,UTYPE=TERMOUT]

                  [,SYNAD=addr]  [,RNTYPE=ABS]
                                 [,RNTYPE=REL]

                  [,OPT=([NOEDIT] [,NOTRAN])]
```

DSECT
: specifies that the actual QIO Request Block should not be built. Instead, a QIORB DSECT is generated.

BUFAD=
: specifies the address of an area in user storage where the control program is to place a record, if a QGET operation is performed, or retrieve a record for writing, if a QPUT operation is performed.

BUFLEN=
: specifies the length of the BUFAD area. It may be coded either as a number or as a non-relocatable symbol. If this operand is not coded, a default of 0 is used.

RECFM=
: specifies the format of the records to be processed. If F is specified, the file will be processed as a standard f-format file. If V is specified, the file will be processed as a standard v-format file. See Chapter 2 for a description of record and file formats.

UNIT=
: specifies the unit number defining the file to be processed. It

may be coded as either a number or a non-relocatable symbol.

DDNAME=
specifies a ddname (see the VPS User's Guide for a description of the ddname parameter on the /FILE statement) defining the file to be processed.

UTYPE=
specifies the unit type (see the VPS User's Guide for a description of the unit type parameter on the /FILE statement) defining the file to be processed. Note the control program will search (in unit number order) the /FILE statements and choose the first occurrence of the specified file type.

EODAD=
specifies the address of a routine which is to be given control if an end-of-file condition is encountered while processing the file.

SYNAD=
specifies the address of a routine which is to be given control if an operation could not be completed successfully.

RNTYPE=
(for direct access files only)
specifies whether the file is to be processed in absolute addressing mode or relative addressing mode. See Chapter 2 for a description of randomly accessing direct access files.

OPT=
(for the conversational read unit only)
specifies the type of internal processing the control program will not perform when reading or writing conversationally from the terminal. If NOEDIT is specified, the control characters will not be removed from the a line when reading, and not inserted into the line when writing. If NOTRAN is specified, the line will not be translated to EBCDIC from terminal code when reading, and will not be translated to terminal code when writing. If NOTRAN is specified, NOEDIT is assumed. If only one option is specified, it need not be enclosed in parentheses.

Note:
Many of the keywords on the QIORB macro may be overridden at execution time using the corresponding keywords on the QGET, QPUT, QEOF, QREW, and QBSP macro instructions.

QPUT - Write Record (QIO)

The QPUT macro instruction causes a record to be written to a file from a designated area in user storage. The location of the record written in the file depends upon the physical structure of the file and information specified in the QIO Request Block (QIORB). If the file is being written sequentially, the record will be placed in the next logical location in the file. If the file is a direct access file and a record number is specified, the record will be placed in the file relative to the beginning of the file (if absolute addressing is used) or relative to the last record written (if relative addressing is being used). The standard form of the QPUT macro is written below.

```
[symbol]   QPUT  [addr    ]  [,length  ]  [,unit          ]   [,RB=addr  ]
                 [(reg1)  ]  [,(reg2)  ]  [,(reg3)        ]   [,RB=(reg) ]
                                          [,'ddname'      ]
                                          [,UTYPE=DISK    ]
                                          [,UTYPE=LIBIN   ]
                                          [,UTYPE=LIBOUT  ]
                                          [,UTYPE=SYSIN   ]
                                          [,UTYPE=SYSOUT  ]
                                          [,UTYPE=TAPE    ]
                                          [,UTYPE=TERMIN  ]
                                          [,UTYPE=TERMOUT ]

                 [,RECFM=F   ]  [,EODAD=addr  ]  [,SYNAD=addr  ]
                 [,RECFM=V   ]  [,EODAD=(reg) ]  [,SYNAD=(reg) ]

                 [,RECNO=number]  [,RNTYPE=ABS]  [,SVC=YES]
                 [,RECNO=(reg)  ]  [,RNTYPE=REL]  [,SVC=NO ]
```

addr
: specifies the address of an area in user storage from which the record is to be written. If this operand is coded, it will replace the 'bufad' field in the QIORB.

(reg1)
: specifies a register containing the address of an area in user storage from which the record is to be written. If this operand is coded, it will replace the 'bufad' field in the QIORB.

length
: specifies the length of the area from which the record is to be written. It may be specified as either a number or a non-relocatable symbol. If this operand is coded, it will replace the 'buflen' field specified in the QIORB.

(reg2)
: specifies a register containing the length of the area from which the record is to be written. If this operand is coded, it will replace the 'buflen' field specified in the QIORB.

unit
specifies the unit number defining the file into which the record is to be written. It may be specified either as a number or as a non-relocatable symbol. If this operand is coded, it will replace the unit number specified in the QIORB.

(reg3)
specifies a register containing the unit number into which the record is to be written. If this operand is coded, it will replace the unit number specified in the QIORB.

'ddname'
specifies a ddname (see the VPS User's Guide for a description of the ddname parameter on the /FILE statement) defining a file into which the record is to be written. If this operand is coded, it will replace the ddname specified in the QIORB.

UTYPE=
specifies the unit type (see the VPS User's Guide for a description of the unit type parameter on the /FILE statement) defining a file into which the record is to be written. If this operand is coded, it will replace the unit type specified in the QIORB. Note that the control program will search (in unit number order) the /FILE statements and choose the first occurrence of the specified unit type.

RB=
specifies the address of a QIO Request block (the QIORB can be built with the QIORB macro). It may be coded either as an address or as a register, enclosed in parentheses, containing the address of the QIORB. If this operand is not coded, the QPUT macro will build the QIORB as part of the inline expansion of the macro.

RECFM=
specifies the format of the record as it appears in user storage. If F (fixed) is specified, the record appears left-justified in the user storage area. If V (variable) is specified, the user area contains a standard v-format record (see Chapter 2 for a description of a v-format record). If this operand is coded, it will replace the record format specified in the QIORB.

EODAD=
specifies the address of a routine which is to be given control if the end-of-file is reached. It may be coded either as an address or as a register, enclosed in parentheses, containing the address of the routine. If this operand is coded, it will replace the address specified in the QIORB.

SYNAD=

specifies the address of a routine which is to be given control if the write operation could not be completed successfully. It may be specified either as an address or as a register, enclosed in parentheses, containing the address of the routine. If this operand is coded, it will replace the address specifed in the QIORB.

RECNO=

specifies either the absolute or relative record number, depending upon the RNTYPE specified in the QIORB, of the record to be written into the file. It may be coded as a number or a non-relocatable symbol or as a register, enclosed in parentheses, containing the record number. This operand can be coded only when writing direct access files (i.e. DSORG=DA appears on the associated /FILE statement).

RNTYPE=

specifies whether the record number in the RECNO keyword is an absolute record number (relative to the beginning of the file - the first record being record number 1) or a record number relative to the current position in the file. If this operand is coded, it will replace the RNTYPE field in the QIORB. This operand can be coded only when writing direct access files (i.e. DSORG=DA appears on the associated /FILE statement).

SVC=

specifies whether the QPUT macro expansion is to include the QIO SVC to perform the write operation. The default is YES. If NO is specified, the macro expansion will contain all the necessary instructions - except for the SVC. This form of the macro can be used to format a QIORB.

QREW - Rewind File (QIO)

The QREW macro instruction causes a file to be rewound. The QREW macro may be used to rewind tape files, direct access files, and the conversational read unit (which effectively clears the stack). For tapes and direct access files that are being written, an end-of-file (EOF) record is written before the file is rewound. The standard form of the QREW macro is written below.

```
[symbol]   QREW  [unit      ]   [,RB=addr  ] [,SYNAD=addr  ]
                 [(reg1)    ]   [,RB=(reg) ] [,SYNAD=(reg) ]
                 ['ddname'  ]
                 [UTYPE=DISK]
                 [UTYPE=TAPE]
                 [UTYPE=TERMIN]

                 [,SVC=YES]
                 [,SVC=NO ]
```

unit
specifies the unit number defining the file which is to be rewound. It may be specified either as a number or as a non-relocatable symbol. If this operand is coded, it will replace the unit number specified in the QIORB.

(reg1)
specifies a register containing the unit number of the file which is to be rewound. If this operand is coded, it will replace the unit number specified in the QIORB.

'ddname'
specifies a ddname (see the VPS User's Guide for a description of the ddname parameter on the /FILE statement) defining the file which is to be rewound. If this operand is coded, it will replace the ddname specified in the QIORB.

UTYPE=
specifies the unit type (see the VPS User's Guide for a description of the unit type parameter on the /FILE statement) defining the file which is to be rewound. If this operand is coded, it will replace the unit type specified in the QIORB. Note that the control program will search (in unit number order) the /FILE statements and choose the first occurrence of the specified unit type.

RB=
specifies the address of a QIO Request Block (the QIORB can be built with the QIORB macro). It may be coded either as an address or as a register, enclosed in parentheses, containing the address of the QIORB. If this operand is not coded, the QREW macro will build the QIORB as part of the inline expansion of the macro.

SYNAD=

specifies the address of a routine which is to be given control if the rewind operation could not be completed successfully. It may be specified either as an address or as a register, enclosed in parentheses, containing the address of the routine. If this operand is coded, it will replace the address specified in the QIORB.

SVC=

specifies whether the QREW macro expansion is to include the QIO SVC to perform the rewind operation. The default is YES. If NO is specified, the macro expansion will contain all the necessary instructions - except for the SVC. This form of the macro can be used to format the QIORB.

SALIST -- Set Attention Parameter List

The SALIST macro is used to build the SETATTN parameter list. The SETATTN macro instruction is used to specify an attention exit routine. The SALIST macro instruction does not generate instructions. The format of the SALIST macro is coded below.

```
[symbol]    SALIST  [OBUF=(output buffer address,length)]
                    [OBUF=(0,0)                        ]

                    [,IBUF=(input buffer address,length)]
                    [,IBUF=(0,0)                        ]

                    [,USADDR=user address]
                    [,USADDR=0           ]
```

OBUF=

specifies the output buffer address and length of the output buffer. The output buffer contains a message which is to be printed at the user's terminal when a program attention interrupt occurs. This message is printed before the attention exit routine is given control. If OBUF is not coded, the address and length will be initialized to 0.

IBUF=

specifies the input buffer address and the length of the input buffer. The input buffer will be used for responses from the terminal user after the output buffer has been printed. The attention exit routine will not be given control until the control program has placed the user's reply into this buffer. If IBUF is not coded, the address and length will be initialized to 0.

USADDR=

specifies the address of an area containing information which the programmer wishes to have passed to the attention exit routine when it is given control. Upon entry to the attention exit routine, register 1 will contain the address of this area. If USADDR is not coded, the address will be initialized to 0.

Note:

The keywords on the SALIST macro may be overridden at execution time using the corresponding keyword on the SETATTN macro.

SETATTN - Set Terminal Attention Exit

The SETATTN macro instruction is used to specify the address of an attention exit routine that is to be given control when a program attention interrupt occurs during the execution of a program at a terminal. The SETATTN macro may also be used to cancel the last attention exit routine established by the program or to turn off attention suppression in the exit routine. The format of the SETATTN macro is written as follows:

```
[symbol]   SETATTN  [CANCEL]  {,LIST=addr }  {,EXIT=addr }  {,SAVE=addr }
                    [SET   ]  {,LIST=(reg)}  {,EXIT=(reg)}  {,SAVE=(reg)}
                    [SOFF  ]

                    [,OBUF=(output buffer address,length)]
                    [,OBUF=((reg),length)                ]

                    [,IBUF=(input buffer address,length) ]
                    [,IBUF=((reg),length)                ]

                    [,USADDR=user address]
                    [,USADDR=(reg)       ]
```

CANCEL
: is written as shown. CANCEL indicates to the control program that the last attention exit routine established by the program is to be ignored. If CANCEL is specified, no other parameters need be coded on the SETATTN macro statement.

SET
: is written as shown. SET indicates to the control program that a terminal attention exit is being established. SET is the default.

SOFF
: is written as shown. SOFF indicates to the control program that attention suppression should be turned off. This form of the SETATTN macro can be used in the attention exit routine to allow attention interrupts during the execution of the routine; otherwise, attention interrupts will be suppressed until the attention exit routine returns control to the control program. If SOFF is specified, no other parameters need be coded on the SETATTN macro statement.

LIST=
: specifies the address of the SETATTN parameter list. This area is defined using the SALIST macro and contains the user address, input buffer address, input buffer length, output buffer address, and output buffer length. If (reg) is specified, the address in the designated register is used. NOTE, LIST=(1) may not be coded.

EXIT=

specifies the address of the entry point of the routine to be given control when an attention interrupt is received. If (reg) is specified, the address in the designated register is used.

SAVE=

specifies the address of a save area 18 fullwords in length and fullword aligned. If (reg) is specified, the address in the designated register is used. Upon entry to the attention exit routine, register 13 will contain the address of this save area - allowing standard subroutine linkage. NOTE, SAVE=(1) may not be coded.

OBUF=

specifies the output buffer address and length of the output buffer. If this keyword is used, the output buffer address and length in the SETATTN parameter list will be replaced. The output buffer contains a message which is to be printed on the user's terminal when a program attention interrupt occurs. This message is printed before the attention exit routine is given control. The buffer address may also be specified as a register (enclosed in parentheses). The macro will use the address in the designated register as the output buffer address.

IBUF=

specifies the input buffer address and length of the input buffer. If this keyword is coded, the input buffer address and length in the SETATTN parameter list will be replaced. The input buffer will be used for responses from the terminal user after the output buffer has been printed. The attention exit routine will not be given control until the control program has placed the terminal user's reply into this buffer. The buffer address may also be coded as a register (enclosed in parentheses). The macro will use the address in the designated register as the address of the input buffer.

USADDR=

specifies the address of an area containing information which the programmer wishes to have passed to the attention exit routine when it is given control. If this keyword is specified, the user address in the SETATTN parameter list will be replaced. If (reg) is specified, the address in the designated register is used. Upon entry to the attention exit routine, register 1 will contain the address of this area.

The SETIMER macro instruction is used to set a timer interrupt and specify the address of a timer completion routine. The timer interrupt may be specified as either a specific time of day or as an interval in 300ths of a second units. In either case, when the timer interval expires, the timer completion routine is given control. Only one timer interval is in effect at a time. A second SETIMER macro instruction issued before the first time interval expires overrides the first interval and exit routine. The standard form of the SETIMER macro is written below.

```
[symbol]   SETIMER  {addr    } {,CLOCK=number } {,EXIT=routine addr}
                    {(reg)   } {,CLOCK=(reg)  } {,EXIT=(reg)      }
                               {,TOD=number   }
                               {,TOD=(reg)    }
```

addr
: is the address in user storage of a save area 18 fullwords in length and fullword aligned. If (reg) is specified, the address in the designated register is used. Upon entry to the timer completion routine, register 13 will contain the address of this save area - allowing standard subroutine linkage. Note, (0) or (1) may not be coded.

CLOCK=
: is either a number which is the interval size in 300ths of a second units or a register containing the interval size in 300ths of a second units. If a register is specified it must be enclosed in parentheses.

TOD=
: is either a number which is the time of day in 300ths of a second units at which the timer interval is to expire or a register containing the time of day in 300ths of a second units. If a register is specified it must be enclosed in parentheses.

EXIT=
: is the address of the timer completion routine which is to receive control when the interval expires. If a register is specified, the address in the designated register is used. Note, EXIT=(1) may not be specified.

SETTRP - Set Trap Area

The SETTRP macro is used to replace the system defined (or current) SVC trapping and interupt handling control area with a new one. The area is defined using the TRAPAREA macro and allows the programmer to trap SVCs, program interrupts, and/or abends. The standard form of the SETTRP macro is written below.

[symbol] SETTRP $\left\{\begin{matrix}\text{addr} \\ \text{(reg)}\end{matrix}\right\}$

addr
: is the address of the new trap area. If (reg) is specified, the address in the designated register is used.

The SMSG macro is used to send a variable length message to the terminal user, another terminal user (another account), or to the operator. The message may be a message built by either the MESGV macro or the BLDMSG macro, in conjunction with the MESGB macro (see the descriptions of these macros for further details); or the message may be specified as part of the SMSG macro. The standard form of the SMSG macro is written below.

```
[symbol]   SMSG  [account address]  {,'message'      }
                 [OP              ]  {,message address}
                 [(reg1)          ]  {,(reg2)         }
                 [*               ]
                 [_               ]
```

account address
: specifies the address of an 8-byte field containing the account number to which the message is to be sent, left justified and padded to the right with blanks.

OP
: is written as shown and specifies that the message is to be sent to the VPS system operator.

(reg1)
: specifies a register containing the message destination code. The specified register must contain one of the following:
  0 - the message will be sent to the system operator.
  -1 - the message will be sent to the terminal user.
  account address - same as above.

*
: is written as shown and specifies that the message is to be sent to the terminal user. * is the default.

'message'
: specifies the message to be sent and must be enclosed in quotes.

message address
: specifies the address of the message to be sent. This may be either an area of user storage containing a message constructed by the MESGV macro or an area initialized by the BLDMSG macro. If the latter is used, the message address must be the area address +2 bytes (since BLDMSG stores the length of the area in the first 2 bytes).

(reg2)
: specifies a register containing the message address following the guidelines listed above.

Notes:

When control is returned to the program, register 15 will contain one of the following return codes:

Hexadecimal Code	Meaning
0	Message sent successfully.
4	Destination not found.
8	Not receiving messages.
C	Invalid argument.

The SMSGR macro may be used to send a message and have a reply returned to the program. The message may be sent to the terminal user, another terminal user (another account), or to the VPS operator. The SMSGR macro is used to send the message and notify the control program that a reply is expected. Control will be returned to the user program immediately after the message is sent. If the programmer wishes the reply to be processed by the program, the TWAIT macro should be used to cause the user program to wait until the reply is received. The standard form of the SMSGR macro is written below.

```
[symbol]   SMSGR  [*   ]  [,addr      ]  [,SMARG=addr   ]  [,REPLY=addr   ]
                  [OP  ]  [,(reg)     ]  [,SMARG=(reg)  ]  [,REPLY=(reg)  ]
                  [addr]  [,'message' ]
                  [(reg)]  [          ]

                  [,REPLEN=number]  [,POST=addr  ]
                  [,REPLEN=(reg) ]  [,POST=(reg) ]

                  [,PBITS=number]
                  [,PBITS=(reg) ]
```

first positional parameter

specifies the message destination. '*' indicates that the message should be sent to the user terminal (if the program is being run in the batch, the message will be sent to the VPS operator). OP indicates that the message is to be sent to the VPS operator. If an address is specified, the area pointed to by the address is an 8-byte field containing the destination account. If a register is specified, it must be enclosed in parentheses and contain one of the following: 0 - indicating the message is to be sent to the VPS system operator; -1 - indicating the message is to be sent to the terminal user; or an address of an 8-byte area containing the destination account. The default is '*'.

second positional parameter

specifies the message to be sent. It may be specified as: an address of an area containing a variable length message; a register holding the address of an area containing a variable length message; or the actual message to be sent enclosed in quotes. Note that the message may be built using the MESGV macro or the BLDMSG macro (if the latter is used, the message address is the area address +2 bytes). If this parameter is specified, it will override the message address in the SMSGR argument list.

SMARG=

specifies the address of the SMSGR argument list. It may be specified as an address of the argument list or a register containing the argument list address. If a register is coded, it must be enclosed in parentheses. The SMSGR argument list should be built using the SMSGRARG macro. If this parameter is not specified, an inline argument list will be built.

REPLY=

specifies the address of an area where the reply to the message is to be placed by the VPS control program. It may be specified either as an address or as a register containing the address of the reply area. If a register is specified, it must be enclosed in parentheses. When the reply is placed in the reply area, it will be left-justified and padded to the right with blanks. If this parameter is coded, it will override the reply address in the SMSGR argument list. Note, REPLY=(1) may not be coded.

REPLEN=

specifies the length of the reply area. It may be specified as either a number or non-relocatable symbol or as a register containing the length of the reply area. If a register is coded, it must be enclosed in parentheses. If this parameter is specified, it will override the reply length in the SMSGR argument list. Note, REPLEN=(1) may not be coded.

POST=

specifies the address of a 1-byte area the control program is to modify (under control of the PBITS parameter) indicating that the reply has been received and placed in the reply area. It may be specified either as an address or as a register containing the address enclosed in parentheses. See the TWAIT macro for information on the use of the post byte in a user program. Note, POST=(1) may not be coded.

PBITS=

specifies the bit positions in the post byte which the control program is to 'OR' with 1's when the reply has been received and placed in the reply area. It may be specified either as a number or non-relocatable symbol or as a register containing the bit positions in bits 24-31 of the register. If a register is specified, it must be enclosed in parentheses. Note, PBITS=(1) may not be coded.

Note:

When control is returned to the program register 15 will contain one of the following return codes:

Hexadecimal Code	Meaning
0	Message sent successfully.
4	Destination not found.
8	Destination not receiving messages.
C	Invalid argument.

SMSGRARG - SMSGR Argument List

The SMSGRARG macro builds an SMSGR argument list. This macro does not generate any instructions. The standard form of the SMSGRARG macro is written below.

```
[symbol]   SMSGRARG  [DSECT     ]  [,REPLY=addr]  [,REPLEN=number]
                     [addr       ]                [,REPLEN=0     ]
                     ['message'  ]
                     [,POST=addr]  [,PBITS=number]
                                   [,PBITS=0     ]
```

first positional parameter

specifies the message to be sent. It may be specified as either the address of an area containing a variable length message or as the actual message enclosed in parentheses. Note that the message may be built using either the MESGV macro or the BLDMSG macro (if the latter is used, the message address is the area address +2 bytes). If DSECT is specified for the first positional parameter, the SMSGR argument list is not built. Instead, a SMSGRARG DSECT is generated.

REPLY=

specifies the address of an area where the reply to the message is to be placed by the VPS control program. When the reply is placed in the reply area, it will be left-justified and padded to the right with blanks.

REPLEN=

specifies the length of the reply area. It may be coded as either a number or a non-relocatable symbol.

POST=

specifies the address of a 1-byte area the control program is to modify (under control of the PBITS parameter) indicating that the reply has been received and placed in the reply area. See the TWAIT macro for information on the use of the post byte in a user program.

PBITS=

specifies the bit positions in the post byte which the control program is to 'OR' with 1's when the reply has been received and placed in the reply area. It may be specified as either a number or a non-relocatable symbol.

Note:

All of the keywords on the SMSGRARG macro may be overridden at execution time using the corresponding keywords on the SMSGR macro.

SYSLIB - System Library and Work File List

The SYSLIB macro instruction builds a search list (concatenation list) of system library names and/or work file unit numbers which can be used with either the LMBLDL or LMLOAD macro instructions. The SYSLIB macro does not generate any instructions. The standard form of the SYSLIB macro instruction is written below.

```
[symbol]   SYSLIB   {LIST=(sys library | unit,sys library | unit,...)}
```

LIST=

specifies a list of system library names and/or work file unit numbers arranged in the order that they are to be searched. The list must be enclosed in parentheses and the entries separated by commas. If only one name or unit number is coded, the parentheses may be omitted.

TESTIMER - Test Timer

The TESTIMER macro instruction causes the control program to return in register 0 the amount of time (in 300ths of a second units) remaining in a timer interval previously set by a SETIMER macro instruction. If a time interval has not been set, register 0 contains 0 on completion of the macro. TESTIMER can also be used to cancel the remaining time interval.

The TESTIMER macro instruction is written as follows:

```
[symbol]    TESTIMER    [CANCEL]
```

CANCEL
: is written as shown. It indicates that the remaining time interval and exit routine are to be canceled (ignored). If CANCEL is not designated, the unexpired portion of the time interval remains in effect.

TOD - Time Of Day

The TOD macro instruction is used to obtain the time of day in 300ths of a second. The time of day is returned in register 0. The TOD macro is coded as follows.

```
[symbol]   TOD
```

TRAPAREA - User Defined SVC/ABEND/PI Trapping Area

The TRAPAREA macro instruction builds an alternative SVC/Abend/Program Interrupt trapping area in user storage used by the SETTRP macro to replace the system defined (or current) table. The TRAPAREA macro does not generate any instructions. The standard form of the TRAPAREA macro instruction is written below.

```
[symbol]   TRAPAREA  {TRAP=([ABEND] [,ALL] [,API] [,SPI] [,SVC])}
                     [,SUPR=([AMSG] [,SMSG])]  [,ABEXIT=addr]
                     [,PIEXIT=addr]  [,SVCEXIT=addr]
                     [,SVCMASK=ALL                          ]
                     [,SVCMASK=OS                           ]
                     [,SVCMASK=(number | range,...)         ]
```

TRAP=
specifies the conditions the programmer wishes to trap. The options are:
ABEND - all program abends.
ALL - all conditions (ABEND, API, SPI, and SVC).
API - arithemtic program interrupts (interrupt codes 08-0F).
SPI - serious program interrupts (interrupt codes 01-07).
SVC - the supervisor calls listed in the SVCMASK keyword.
If only one option is coded the parentheses may be omitted.

SUPR=
specifies supression of the PSW and general purpose register contents messages for either an arithmetic program interrupt or serious program interrupt. If AMSG is coded the messages will be supressed if an API occurs. If SMSG is coded the messages will be suppressed if an SPI occurs. If only one option is specified, the parentheses may be omitted.

ABEXIT=
specifies the address of a routine which will be given control if an abend occurs and ABEND is coded in the TRAP keyword.

PIEXIT=
specifies the address of a routine which will be given control if a program interrupt occurs and either API or SPI is coded in the TRAP keyword.

SVCEXIT=
specifies the address of a routine which will be given control if an SVC is executed which has been masked by the SVCMASK keyword and the BRTRAP macro has been issued.

SVCMASK=

specifies the SVCs to be trapped. If ALL is coded, all SVCs will be trapped. If OS is coded, SVCs 0 through 127 will be trapped. If a list is specified, it must be enclosed in parentheses and each entry must be separated by commas. Each entry in the list can either be a single SVC number or a range. If a range is specified, it must be enclosed in parentheses and the beginning and ending numbers must be separated by a comma.

Notes:

If a trap exit is taken, information in the trapping area may be of use in the routine (e.g. PSW contents, general purpose register contents, etc). The PISVCTRP DSECT should be used to reference the trapping area since the area is likely to change.

TWAIT - Task Wait

The TWAIT macro is used to suspend program execution pending the occurrence of some 'posting' event (e.g. a response to a SMSGR request). This macro is always immediately preceded by a Test-under-Mask instruction which tests one or more bits in a programmer-defined post byte. A condition code mask is defined for the TWAIT macro in the same manner as the mask which is used for the Branch-on-Condition instruction. This mask determines whether suspension of execution is to occur. This macro may be thought of as a 'wait on condition' instruction - that is, the program will be suspended as long as the test instruction produces a condition code corresponding to a 1 bit in the mask. The standard form of the TWAIT macro is written below.

[symbol] TWAIT [number / 8]

number
: specifies the mask used to determine whether suspension of execution is to occur. It may be specified as either a number or as a non-relocatable symbol and is defined in exactly the same way as the branch mask for the Branch-on-Condition instruction. The default is 8 which specifies that program execution will be suspended as long as the posting bit(s) in the post byte are zeroes.

Below is a list of some of the VPS data areas which can be referenced from assembler language programs. Each of the data areas has a corresponding DSECT of the same name residing under the VPS library index - VPSMAC. Programmers are cautioned that there is no guarantee that these data areas will not change. Therefore, the symbols defined in the DSECTs should always be used when referring to fields in the data areas instead of hard coded offsets. This will lessen the impact of modifications or enhancements to the control program data areas.

Note, in the data areas listed below, information and constants may be referenced. However, a protection exception will occur if a problem program attempts to modify any of these areas.

AMSL (Assigned Main Storage Locations) -
defines the low-address storage locations of the virtual machine. AMSL contains frequently used constants, addresses of other data areas, boundary information, etc. To use the symbols defined in the AMSL DSECT, the following statements should appear in the program:

```
AMSL
USING AMSL,0
```

JCB (Job Control Block) -
contains control and status information related to the user's job or terminal session. $JCBPTR in the AMSL DSECT contains the address of the associated JCB. $PVMJCB in the AMSL DSECT contains the address of the PVM JCB, which is always the first in the JCB chain.

LKDSECTS (Link DSECTs) -
this DSECT contains all the data areas pertaining to load modules.

QIOPREFX (QIO Prefix) -
contains information relevant to the QIO Access Method. There is a QIO Prefix for each unit defined by a batch job or the execution of a program during a terminal session. $SYSUAV in the AMSL DSECT contains the address of the QIO Prefix vector. The first word of this vector contains the highest defined unit number, followed by successive words each containing the address of the QIO Prefix for that unit (if defined) or zero (if not defined). For example, if 'n' is the defined unit number, n*4 bytes from the beginning of the QIO Prefix vector is the address of the QIO Prefix for that unit. (See UNITDEF below.)

REPLYBLK (Reply Block) -
is built by the sytem for an outstanding reply (see the SMSGR macro). The REPLYBLK is chained from the JCBs of both the sender and the receiver.

TCB (Terminal Control Block) -
contains status and control information specific to the terminal being used for this terminal session. This control block is pointed to by the JCBTCB field in the JCB. For a job running on the batch, the JCBTCB field will contain zero.

UNITDEF (Unit Definition) -
is contained within the QIO Prefix and contains information pertaining to the unit from the associated /FILE statement (i.e. logical record length, block size, etc.). Note, UNITDEF must appear in a program physically before QIOPREFX since some symbols defined in the UNITDEF DSECT are used in the QIOPREFX DSECT.

VPS SYSTEM FACILITIES FOR ASSEMBLER PROGRAMMERS Reader Comments Form

As a reader of this draft, your comments and criticisms of the material presented and the manner of presentation are essential to the usefulness of this manual in its final form. Please comment in the space below, giving specific page and paragraph references where possible.

This sheet can be sent to the Computing Center or dropped off in Room 3.