The VPS Guide
to
Programming Languages

Boston University Computing Center

Preface

The publication of this manual marks the last in a set of four basic VPS™ manuals - The VPS Handbook, explaining the system, its commands, and control statements; the VPS Utilities Manual, documenting general purpose programs such as the VPS Editor and Loader; VPS System Facilities for Assembler Programmers, describing system functions and macros available at the problem program level; and now the VPS Guide to Programming Languages, describing the use of the major programming languages on VPS. Along with this Guide, the Handbook and the Utilities Manual are mandatory reading for anyone who wishes to write and execute programs on VPS with minimal frustration.

We would like to thank Joe Demty, Manager of User Services, for his contributions and Mary Porter, who so brilliantly captured the need for this manual in her cover design.

Finally, we strongly recommend that you read the introduction before you begin reading the rest of the Guide to avoid unnecessary confusion.

Marian G. Moore
Marjorie A. Orr

June, 1980

Contents

Introduction

VPS (Virtual Processor System) is a high-performance general purpose timesharing system written by the systems staff of the Boston University Computing Center. Unlike some highly specialized interactive systems which are built to perform a unique process (e.g. handle insurance claims, process bank transactions, find parking violators, etc.), VPS is designed to allow users the ability to create, maintain, and run a wide variety of programs and packages.

The programs described in this Guide are called compilers. In general, a compiler is an application program which processes a defined set of language statements (the programming language known to that program) and produces machine language from these statements which can be executed directly by a computer (in our case, an IBM computer).

This is a unique manual in what it does not contain. For instance, it does not contain information on VPS commands or the creation and maintenance of source and data files under VPS. That information may be found in the Handbook and the Utilities Manual and we assume you know it. It does not contain a description of the language the compiler accepts. Each compiler described in the following chapters has a language reference manual for this purpose. It does not contain error messages or codes produced by the compiler. That information can also be found elsewhere.

What it does contain is a description of how the compiler interfaces with the VPS operating system, the VPS control statements needed to compile and execute programs in that language, the file structures available on VPS supported by that particular language, any differences between the compiler's behavior on VPS versus its behavior on its native system, and a list of the publications documenting the language.

Readers may find that their favorite compiler is not documented in this manual (a notable exception is LISP). Certainly, there are many more programming languages available on VPS than are described in this manual; however, time and manual size were factors we had to consider while writing. Therefore, we chose to document only the most widely used languages.

Compiler Support

The compilers documented in this manual are programs "supported by the Computing Center". Loosely speaking that can be defined as a program which is widely used by the computing community, installed by Computing Center personnel, is distributed with adequate documentation for users and the systems staff at the BUCC who must maintain it, and is written by an organization responsible for fixing errors and providing enhancements.

None of the compilers described in this manual were originally written to run under VPS. A few have been modified to run in a mode

which we call "native VPS" - directly using the facilities described in the VPS System Facilties for Assembler Programmers Manual. However, the newer compilers have not been modified and run in an "OS/VS environment". This environment is created through special VPS system programs which emulate the IBM OS/VS operating system. In this manner the compilers need not be modified which allows faster installation of new releases of a compiler, less chance of errors during execution (if there is a compiler problem, usually it is an error in the compiler not in VPS), and the ability to support a much wider range of programs and packages. As time goes by we plan to use this technique on all the compilers supported by the Computing Center.

Example Formats

Three types of examples are used in this manual - VPS control statement examples shown in upper case, program examples shown in lower case, and terminal session excerpts where system responses (*GO, etc.) are shown in upper case to more easily differentiate between what the imaginary user is entering and the associated response. Note that depending upon the terminal type actual responses from VPS may also be in lower case.

No matter how the examples are shown, if a command or statement is to be typed in on a terminal IT SHOULD NOT BE TYPED IN UPPER CASE.

Chapter 1: ALGOL W

The ALGOL compiler supported on VPS is ALGOL W. ALGOL W was written at Stanford University and is based on the ALGOL 60 language. The full language specifications for ALGOL W can be found in "ALGOL W Reference Manual" (STAN-CS-71-230) by Richard L. Sites, published by the Computer Science Department of Stanford University. This manual is available for reference at the Computing Center.

ALGOL W is a fast student-oriented compiler with excellent dianostics and extensive debugging facilities. ALGOL W has not been modified to run under VPS - rather, special VPS support programs provide an "OS/VS" environment in which this compiler and its compiled programs run. ALGOL W is a one-step compile and execute facility that does not produce permanent object code.

Invoking ALGOL W

The ALGOL W compiler is invoked by placing the following control statement before the first ALGOL source statement of a program:

```
/LOAD ALGOLW
```

Normally, this is done by placing the /LOAD statement first in a source file with the VPS editor when the source file is created.

ALGOL W Control Statements

ALGOL W recognizes two special control statements. The first, %ALGOL, signals the beginning of an ALGOL W source program and its format is shown below.

```
%ALGOL [Time=sss ] [,Pages=ppp ] [,MARGin=72 ]
       [    010 ] [       009 ] [        80 ]
```

TIME=
: specifies the maximum execution time in seconds to be alotted for the ALGOL source program. 10 seconds is the default. If the time limit is exceeded, the execution is terminated. TIME= may be abbreviated as T=.

PAGES=
: specifies the maximum number of pages to be generated by execution of the source program. 9 pages is the default. If the page limit is exceeded, the program execution will be terminated. PAGES= may be abbreviated as P=.

MARGIN=
: specifies the number of columns to be read on each input data record. The default value is 80, unless the source statements are sequence numbered, in which case the compiler assumes the data records are sequenced also and 72 is the default.

%ALGOL must begin in column 1, followed by a blank, followed by any of the above options. The options may be entered in any order but must be separated by commas.

The %DATA statement is used in the input stream to separate an ALGOL program from its data. %DATA must begin in column 1.

The ALGOL W compiler allows multiple compilations and executions in one job stream as long as the source and data are correctly delineated with %ALGOL and %DATA statements. The following shows the standard statement sequence for ALGOL programs:

```
/LOAD ALGOLW
%ALGOL options
   ALGOL source statements
%DATA
   program data
.
.
.
%ALGOL options
   ALGOL source statements
%DATA
   program data
```

The %ALGOL statement is optional for the first source program in the input stream if the programmer wishes to select all of the defaults. The %DATA statement is optional for any source program if the program has no data in the input stream.

Compiler Options

The following is a list of the ALGOL W compiler options and a description of their use. A full description can be found in the Stanford "ALGOL W Reference Manual". Each option must begin in column 1 of a record and must appear between a %ALGOL control statement and the next % statement. The option is in effect for that source program only.

Option	Description
$LIST $NOLIST	Specifies whether the source statements should be listed. $LIST is the default on the batch. $NOLIST is the default at the terminal.
$TITLE,"..."	Places "..." (up to 30 characters) as a title in the middle of the heading line.
$SYNTAX	Specifies that the compiler is to analyze the program for syntax errors, but not execute it.
$NOCHECK	Specifies that subscript and undefined variable checking is to be turned off.

$DEBUG,0 $DEBUG,1 $DEBUG,2 $DEBUG,3(m) $DEBUG,4(m)	Activates the tracing, statement counting, and post-mortem dump facilities of ALGOL W. $DEBUG,0 specifies that minimum debugging options are in effect. $DEBUG,1 (the default) specifies that a post-mortem dump of all variables should be produced if the program terminates abnormally. $DEBUG,2 produces the same output as 1 plus statement counts will also be printed. $DEBUG,3(m) produces the same output as 2 plus a statement-by-statement trace of each value stored is also printed. $DEBUG,4(m) produces the same output as 3 plus a statement-by-statement trace of each value fetched is also printed. If tracing is specified ($DEBUG,3 or $DEBUG,4) and the standard procedure TRACE is not used, then each ALGOL statement will be traced in symbolic form the first "m" times it is executed.
$NORM,a,b	Activiates the floating-point significance tracing facilities of ALGOL W. This facility interprets the operation of each floating-point add and subtract executed by the program, counting the number of base 16 digits of preshift and postshift. If these shifts exceed the limits specified by "a" and "b" respectively, then a one-line SIGNIFICANCE ERROR message is printed. The parameters "a" and "b" have a range of from 0 to 16.

Example

The following example shows an ALGOL program where, if abnormal termination occurs, a post-mortem dump of all variables along with statement counts is to be printed.

```
/load algolw
%algol
$debug,2
 begin
 integer sum,count,numb;
 while true do
 begin;
 sum := count := 0;
 readon(numb); writeon(numb);
 while numb~= -1 do
  begin
  sum := sum + numb;
  count := count+1;
  readon(numb); writeon(numb);
  end;
 if count = 0 then write("empty group")
  else write("count ",count,"sum ",sum,"average ",sum/count);
  iocontrol(2);
  end
 end
%data
1 3 2 5 4 6 7 -1
8 7 6 7 6 5 4 5 4 3 -1
-1
8 7 6 6 6 5 5 5 6 5 44 33 22 7 6 7 6 7 6 8 787 88
```

Chapter 2: The Assembler

The Assembler supported on VPS is the IBM OS/VS-VM/370 Assembler. The assembler language supported by this assembler is documented in the IBM publication "OS/VS-DOS/VS-VM/370 Assembler Language" (GC33-4010). Information on its use can be found in the IBM publication "OS/VS-VM/370 Assembler Programmer's Guide" (GC33-4021).

Unlike the compilers described in this manual, the Assembler has been modified to run in the native VPS environment (the system programs that create an "OS/VS" environment are not necessary to assemble a program). However, the object code generated is identical to that produced by the unmodified Assembler. Programs may be assembled which are intended for use on VPS or VS and in fact, programs written for VS may be run on VPS in the "OS/VS" environment (see below).

Invoking the Assembler

The VS Assembler is invoked by placing the following control statement before the first assembler source statement of a program:

```
/LOAD VSASM,PARM='parm1,parm2,...'
```

Normally, this is done by placing the /LOAD statement first in a source file with the VPS editor when the source file is created. Information on the parameters which may be passed to the Assembler using the PARM= keyword can be found in the section "Specifying Assembler Options".

In reality, not only does the /LOAD statement call the VS Assembler, but it also invokes a corresponding VPS internal procedure. A VPS internal procedure consists of a set of VPS control statements needed to assemble most source programs. These control statements may be overridden at execution time by programmer-supplied statements. The VS Assembler procedure invoked using /LOAD VSASM at the terminal is shown below:

```
/FILE UNIT=(DISK,NAME=SYSUT1),SPACE=(TRK,(30,5))
/FILE UNIT=(DISK,NAME=SYSUT2),SPACE=(TRK,(30,5))
/FILE UNIT=(DISK,NAME=SYSUT3(,SPACE=(TRK,(30,5))
/FILE UNIT=(TERMOUT,NAME=SYSPRINT)
/FILE UNIT=(LIBIN,NAME=SYSIN),DSNAME=<input stream>
/FILE UNIT=(LIBOUT,NAME=SYSPUNCH),DSNAME=*2
/FILE UNIT=(TERMOUT,NAME=SYSTERM)
/LOAD VSASM
```

Procedure Files

The default /FILE statements in the VPS VSASM procedure are described below:

File	Use
SYSUT1, SYSUT2, SYSUT3	Defined as temporary work files. Used by the Assembler as scratch files during assembly.
SYSPRINT	Defined as the terminal when assembling at the terminal or as the high-speed printer when assembling on the batch. Used by the Assembler as the output listing file.
SYSIN	Defined as the input stream. Used as the input to the Assembler.
SYSPUNCH	Defined as the *2 file when executing at the terminal or as the card punch when assembling on the batch. Used by the Assembler as the output object file (if DECK appears in the parameter list - see below).
SYSTERM	Defined as the terminal when assembling at the terminal. Used by the Assembler for diagnostic messages and statistics.

Specifying Assembler Options

As shown above, Assembler options, or parameters, may be passed to the VS Assembler using the PARM= keyword of the /LOAD statement. If parameters are passed to the Assembler they must be enclosed in single quotes and separated with commas. The following is a subset of the compiler options accepted by the VS Assembler. A full description of all the options may be found in the Assembler Programmer's Guide. Note that default options are underlined.

Option	Description
LIST NOLIST	Controls the printing of an output listing on SYSPRINT. NOLIST is the default on the terminal, LIST is the default on the batch.
ALIGN NOALIGN	Controls the alignment in the object module produced and the checking of alignment violations. For example, if NOALIGN is specified F-type constants will not necessarily appear on fullword boundaries.
ALOGIC NOALOGIC	Controls the listing of conditional assembly statements in open code.
ESD NOESD	Controls the listing of the external symbol dictionary.
DECK NODECK	Controls the writing of an object module to SYSPUNCH.

OBJ NOOBJ	Controls the writing of the object module to to SYSGO. Note that a /FILE statement must be placed in the job stream defining SYSGO if OBJ is specified.
RENT NORENT	Controls the checking of reentrancy violations in the input source.
RLD NORLD	Controls the listing of the relocation dictionary.
XREF(FULL) XREF(SHORT) NOXREF	Controls the listing of the cross reference. A FULL listing includes symbols that are not referenced.
MACLIB=(VPSMAC, MACLIB)	Specifies the VPS library indexes to be searched for MACRO definitions. VPSMAC contains the VPS MACROs described in "VPS System Facilities for Assembler Programmers". MACLIB contains the VS MACRO library described in various IBM manuals. The order specified is the order searched.

Assembling - An Example

The following example shows a library file ready for assembly at the terminal. The listing file is to be printed on the high-speed printer by overriding SYSPRINT. A deck is requested and will be written to the *2 file. VPSMAC, MACLIB, and USERLIB will be searched for MACRO definitions.

```
/FILE UNIT=(PRINTER,NAME=SYSPRINT)
/LOAD VSASM,PARM='DECK,LIST,MACLIB=(VPSMAC,MACLIB,USERLIB)'
TEST     CSECT
         BALR 12,0
         USING *,12
         ST   14,SAVE14
         SLR  3,3
READ     DS   0H
         QGET RECORD,RB=INRB
         LA   3,1(,3)
         .
         .
         .
```

Executing Assembler Programs

After object code is written to the temporary *2 file (by specifying DECK in the parameter list), it may be saved in a permanent VPS library file using the /SAVE or /RSAV commands (see the VPS Handbook). Object code saved in this manner can be executed in either the native VPS environment or in the "OS/VS" environment with FORTRAN, PL/1, or COBOL source programs or by itself.

If an assembler program uses system facilities and/or MACROs described in the manual "VPS System Facilities for Assembler

Programmers" the program may be executed in the native VPS environment. If an assembler program is written as a subroutine for a source program in a higher level language and/or if it uses system facilities or MACROs documented in IBM publications it must be executed in the "OS/VS" environment.

To execute an assembler program in native VPS, the following control statement must precede the object module(s) to be executed:

```
/LOAD LOADER
```

This control statement invokes the VPS Loader which loads and executes the object found in the job stream. The VPS Loader is described in the VPS Utilities manual.

If the program is to read data from the input stream (SYSIN, unit 5), a /DATA control statement must separate the object from the data. For a job run at the terminal the following default /FILE statements will be used:

```
/FILE UNIT=(5,LIBIN,NAME=SYSIN),DSNAME=<input stream>
/FILE UNIT=(6,TERMOUT,NAME=SYSPRINT),RECFM=FA
/FILE UNIT=(9,TERMIN)
/FILE UNIT=(10,TERMOUT,NAME=SYSSAVE),DSNAME=*2
```

If any other files are to be defined or default files overridden, appropriate /FILE statements must be placed in the job stream before the /LOAD LOADER statement.

For assembler programs to be run with a higher level source program, see the appropriate chapter in this manual for information pertaining to the inclusion of object in the job stream. For assembler programs to be run in the "OS/VS" environment without higher level source, either Chapter 5, describing FORTRAN, or Chapter 8, describing PL/1, should be consulted for information on executing straight object. Except for the differences in default execution /FILE statements, either /LOAD statement will provide the same "OS/VS" environment.

Assembler Program Execution Example

In this example an assembler main program object has been saved in the VPS library file AMAIN and its assembler subprogram object has been saved in the file ASUB. Data is to be read from the input stream from the file ADATA and a work file, UR.PLG123.OUTFILE, is to be created. The following file will execute the program in the VPS native environment:

```
/FILE UNIT=(DISK,NAME=OUTPUT),DSNAME=UR.PLG123.OUTFILE,RECFM=FB,
/ LRECL=80,BLKSIZE=4160,DISP=(NEW,KEEP,DELETE),SPACE=(TRK,(1,1))
/LOAD LOADER
/INC AMAIN,ASUB
/DATA
/INC ADATA
```

Chapter 3: BASIC

The BASIC language currently supported on VPS is the IBM CALL/360-OS BASIC, Version 1.1. The language is fully documented in the IBM publication "CALL/360-OS BASIC Language Reference Manual" (GH20-0699). This compiler has been modified to run under VPS. The following sections describe the language elements implemented in this version of BASIC, its use on VPS, and some helpful debugging hints.

CALL BASIC Language Elements

Constants

CALL BASIC supports the following types of constants:

Numeric constants - numbers are stored as double precision floating point values. This allows up to 16 decimal digits of precision in a numeric constant. However, under CALL BASIC the maximum integer that can be read or printed contains 15 digits and the maximum real number that can be printed without FORMAT control (the PRINT USING statement described later in this chapter) has 10 digits. Fifteen digit real numbers may be printed with the PRINT USING statement.

Literal constants - literal values such as character strings may be represented as constants by enclosing them in either single (') or double (") quotation marks. A literal constant may contain a maximum of 18 characters, including blank spaces.

Internal constants - CALL BASIC supports the following 3 internal constants:

```
&pi   (the value of pi)        3.141592653589793
&e    (the value of e)         2.718281828459045
&sqr2 (the square root of 2)   1.414213562373095
```

Variables

The following types of variables are supported:

Numeric variables - may be assigned a value of up to 16 decimal digits of precision. Numeric variables are represented by a letter of the alphabet, optionally followed by a digit in the range from 0 to 9. The symbols @, $, and # may also be used as "letters" to form numeric variable names. Examples of numeric variables are:

```
a = 5.3
c1 = 1234.5678
@ = 10
```

Character string variables - are used to store literal data. Under CALL BASIC such a variable is represented by a letter followed

by a dollar sign ($). A character string variable may contain up to 18 characters. If uninitialized, it will contain 18 blanks. Examples of character string variables are:

```
a$ = 'this is a string'
f$ = 'some would call me a literal'
$$ = 'i am one also'
```

Arrays - in CALL BASIC may be one dimensional vectors or two dimensional and matrices may be explicitly defined by use of the DIM statement or implicitly defined by use of a subscript value. The latter will have either 10 or 100 elements depending upon whether one (a 10 element array) or two (a 10 X 10 element array) subscripts are used. Only non-zero positive integers may be used as subscripts.

Numeric arrays are represented by a single letter followed by one or two subscripts enclosed in parentheses. Examples are:

```
a(10)
b(4,8)
#(2)
```

Character string arrays are designated by a single letter followed by a dollar sign ($), followed by a single subscript enclosed in parentheses. Each element may contain up to 18 characters. Examples are:

```
a$(10)
j$(4)
```

A numeric array may have a maximum of 3583 elements and a character array may have a maximum of 1592 elements. The total amount of space available for the storage of arrays is 28,668 bytes.

Operators

CALL BASIC supports the following operators for the manipulation of constants and variables.

+	addition
-	subtraction
*	multiplication
/	division
**	exponentiation
<	less than
<=	less than or equal
>	greater than
>=	greater than or equal
=	equal
<>	not equal

BASIC Statements

The following statements are implemented in CALL BASIC:

DATA	MAT PRINT
DEF	MAT READ
DIM	NEXT
END	PAUSE
FOR	PRINT
GOSUB	PRINT USING
GOTO (computed)	READ
GOTO (conditional)	REM
IF	RESET
IMAGE	RESTORE
INPUT	RETURN
LET	STOP

BASIC Functions

The following functions are available in CALL BASIC:

SIN(X)	sine of x radians
COS(X)	cosine of x radians
TAN(X)	tangent of x radians
COT(X)	cotangent of x radians
CSC(X)	cosecant of x radians
ASN(X)	arcsine of x radians
ACS(X)	arc-cosine of x radians
ATN(X)	arctangent of x radians
HSN(X)	hyperbolic sine of x radians
HCS(X)	hyperbolic cosine of x radians
HTN(X)	hyperbolic tangent of x radians
DEG(X)	convert x from radians to degrees
RAD(X)	convert x from degrees to radians
EXP(X)	e to the x power
ABS(X)	absolute value of x
LOG(X)	log x to the base e
LTW(X)	log x to the base 2
LGT(X)	log x to the base 10
SQR(X)	positive square root of x
RND(X)	generate a random number between 0 and 1
INT(X)	integral part of x
SGN(X)	sign of x (-1, 0, +1)

The following array functions are also supported:

CON	set all elements to 1
IDN	set matrix to the identity matrix
INV	invert matrix
TRN	transpose matrix
ZER	set all elements to zero

Additional Considerations

The following items should be noted concerning the use of CALL BASIC:

1) The use of the word LET in assignment statements is optional.

2) All BASIC statements must have unique statement numbers from 1 to 9999. The statement numbers must begin in column 1. Generally, statement number increments of 5 or 10 are used thus permitting future insertion of lines. The statements will be executed in the order in which they appear.

3) The maximum number of statements in a CALL BASIC program is approximately 800. The maximum program space available in CALL BASIC is approximately 60,000 bytes of which no more than 28,668 bytes can be used for array storage.

4) The form of the computed GOTO statement in CALL BASIC is:

   GOTO {line number, line number,...} ON {variable name}

   as opposed to the conventional BASIC form shown below:

   ON {variable name} GOTO {line number, line number,...}

5) The PRINT USING statement has the following characteristics:

   Numeric overflow results in the printing of a set of asterisks, one for each character position in the target format.

   The floating dollar sign ($), useful in financial applications, is not supported. If there is space between a fxed dollar sign and the first digit in a given result, it will be filled with blanks.

   The only supported form of the PRINT USING statement is shown below:

   PRINT USING {lnum} ,var,var,...

   Where lnum is the line number of the format statement and var is either a variable name or value to be printed. The format cannot be placed in the PRINT USING statement.

Invoking BASIC

CALL BASIC is invoked by placing the following control statement before the first source statement of a BASIC program:

```
/LOAD BASIC
```

Normally, this is done by placing the /LOAD statement first in a source file with the VPS editor when the source file is created.

BASIC Input/Output

There are two methods of entering data for BASIC program execution. The first involves the use of the INPUT source statement. If a BASIC program is executing at the terminal the INPUT statement will cause the program to wait and request input from the terminal. If the BASIC program is executing on the batch a VPS /DATA statement must be placed after the last source statement in the job stream, followed by the data in free form. The following shows the standard job setup for a BASIC program which is to be run on the batch and uses the INPUT statement:

```
/LOAD BASIC
    basic source statements
/DATA
    program data
```

The batch input form may be used with the INPUT statement when executing at the terminal by specifying the following option on the /LOAD statement:

```
/LOAD BASIC,TRANS=(9->5)
```

This form of the TRANS keyword indicates that VPS should read the input stream following the /DATA statement for data rather than the terminal.

The second method of inputing data to an executing program is through the use of the READ, DATA, and RESTORE statements. In this method, all data needed for execution of the program is incorporated as source statements within the program before execution. The READ statement assigns values to variables which are found on one or more DATA statements in the program in the order they are found. The DATA statements may be placed anywhere in the program. Generally, they are placed immediately following the READ statement or after the END statement. Note that the VPS /DATA statement should not be used in this case.

The RESTORE statement causes the next READ statement to return to the first value in the first DATA statement and begin the sequence again. The following shows an example of a BASIC program using the READ, DATA, and RESTORE statements:

```
/load basic
10 read a,b,c
20 print a;b;c
30 print
40 restore
50 read d,e,f
60 print d;e;f
70 print
80 read g,h,i
90 print g;h;i
95 data 4,9,2,3,5,7,8,2,1
99 end
```

If this program existed in the VPS library file SHOW the following is an example of its execution:

```
*GO
show

4       9       2

4       9       2

3       5       7
*GO
```

Errors

If either source or execution errors are detected using CALL BASIC an error message is printed. The error message will contain the statement number and a self explanatory message describing the error.

BASIC Example

Assume that the following BASIC program exists in the VPS library file MORTG:

```
/load basic
10 print  ´mortage repayment schedule ´
20 print ´ ´
30 print ´enter principal, rate of interest, and years´
40 input p,r,t
50 let r1=r/100
55 let z=1
60 let t1=t*12
70 let a=p*((r/12)/(1-(1+r1/12)**(-t1)))
75 let a1=a/100
80 print using 81,a1
81 :monthly principal and interest $####.##
81 print ´ ´
90  for i=1 to  t1
100 let i1=(r1/12)*p
110 let p1=a1-i1
120 let y1=y1+i1
130 if p < a1 then  150
140 go to 160
150 let p1=p
160 let p=p-p1
165 let a1=i1+p1
167 let y2=y2+p1
170 let c=c+1
180 if c = 12 then 300
190 go to 260
260 next i
270 go to 500
300 print using 301,z,y1,p
301 :year ## totals...int = $####.##  bal = $######.##
302 let y6=y6+y1
304 let y7=y7+y2
310 let y1=0
320 let y2=0
330 let c=0
335 let z=z+1
340 go to 260
500 print ´ ´
505 print ´total payments´
510 print using 511,y6
511 :    interest = $#####.##
520 print using 521,y7
521 :    principal = $######.##
530 let y6=0
540 let y7=0
550 let y1=0
560 let y2=0
565 print ´ ´
570 print ´run new mortgage schedule?´
580 print ´enter yes  or  no´
590 input k$
600 if k$=´yes´ then 20
999 end
```

The following shows a sample execution of MORTG at the terminal:

```
*GO
mortg

ENTER PRINCIPAL, INTEREST RATE, AND YEARS
1000,6,10
MONTHLY PRINCIPAL AND INTEREST IS $  11.10

YEAR  1 TOTALS...INT = $  57.95  BAL = $   924.73
YEAR  2 TOTALS...INT = $  53.31  BAL = $   844.81
YEAR  3 TOTALS...INT = $  48.38  BAL = $   759.97
YEAR  4 TOTALS...INT = $  43.15  BAL = $   669.89
YEAR  5 TOTALS...INT = $  37.59  BAL = $   574.26
YEAR  6 TOTALS...INT = $  31.69  BAL = $   472.73
YEAR  7 TOTALS...INT = $  25.43  BAL = $   364.94
YEAR  8 TOTALS...INT = $  18.78  BAL = $   250.49
YEAR  9 TOTALS...INT = $  11.72  BAL = $   128.99
YEAR 10 TOTALS...INT = $   4.23  BAL = $     0.00

TOTAL PAYMENTS
      INTEREST = $   332.25
      PRINCIPAL = $  1000.00

RUN NEW MORTGAGE SCHEDULE?
ENTER YES  OR  NO
no
*GO
```

Chapter 4: COBOL

The COBOL available under VPS is IBM OS/VS COBOL, an implementation of the American National Standard COBOL, X3.23-1974, the standard approved by ANSI, the American National Standards Institute. VPS COBOL jobs run in a simulated IBM OS/VS environment provided by support software. Thus the IBM documentation for the compiler is in large part applicable to its use on VPS, and this chapter is only a supplement to indicate VPS control statement usage and to qualify support for certain COBOL features.

IBM Manuals

There are two pertinent IBM manuals:

"IBM VS COBOL for OS/VS" (GC26-3857)

"IBM OS/VS COBOL Compiler and Library Programmer's Guide" (SC28-6483)

The former is the compiler Language Reference Manual; it documents both the 1974 COBOL standards and IBM extensions to them, including language from the older 1968 standards still supported for compatibility. This chapter provides the information on VPS control statements needed to execute COBOL programs. Detailed descriptions of all VPS control statements can be found in the VPS Handbook. The IBM COBOL Programmer's Guide contains descriptions of several compiler features such as Advanced Symbolic Debugging, Optimized Object Code, and the Lister feature to reformat source code. Also note that the compiler error messages are contained in the Programmer's Guide and not in a separate manual as for other languages.

Invoking the Compiler

The COBOL compiler is invoked by placing the following statement in the input stream before any language source statements:

```
/LOAD COBOLVS
```

Normally this is done with the VPS editor when a COBOL source file is created. When the file is subsequently executed, a program is loaded that controls three distinct phases:

1) compiling of the COBOL source language
2) loading the compiler-generated code into storage for execution
3) actually executing this code.

To accomplish the three steps, certain files are required and are supplied by default by VPS upon recognition of the /LOAD COBOLVS. These control statements, an internal VPS procedure, are listed below.

```
  /FILE UNIT=(DISK,NAME=LINKLIB),DISP=SHR,DSN=<compiler load library>
  /FILE UNIT=(DUMMY,NAME=SYSPRINT)
  /FILE UNIT=(TERMOUT,NAME=SYSTERM)
  /FILE UNIT=(DISK,NAME=SYSLIN),SPACE=(REC,(1000,1000)),
  /     RECFM=FB,LRECL=80,BLKSIZE=1600
  /FILE UNIT=(LIBOUT,NAME=SYSPUNCH),DSN=*2
  /FILE UNIT=(DISK,NAME=SYSUT1),SPACE=(BLK,(25,4)),
  /     RECFM=U,BLKSIZE=19069,LRECL=19069
  /FILE UNIT=(DISK,NAME=SYSUT2),SPACE=(BLK,(10,4)),
  /     RECFM=F,BLKSIZE=19069,LRECL=19069
  /FILE UNIT=(DISK,NAME=SYSUT3),SPACE=(BLK,(10,4)),
  /     RECFM=F,BLKSIZE=19069,LRECL=19069
  /FILE UNIT=(DISK,NAME=SYSUT4),SPACE=(BLK,(10,4)),
  /     RECFM=F,BLKSIZE=19069,LRECL=19069
  /FILE UNIT=(DISK,NAME=SYSUT5),SPACE=(BLK,(54,27)),
  /     RECFM=F,LRECL=512,BLKSIZE=512
  /FILE UNIT=(LIBIN,NAME=SYSIN),DSN=<input stream>
-----
  /FILE UNIT=(DISK,NAME=LINKLIB),DISP=SHR,DSN=<COBOL execution
                                                time routines>
  /FILE UNIT=(DISK,NAME=SORTLIB),DISP=SHR,DSN=<SORT utility library>
  /FILE UNIT=(DISK,NAME=SYSUT5)  same SYSUT5 as for the compile step
  /FILE UNIT=(TERMOUT,NAME=SYSPRINT)
  /FILE UNIT=(LIBOUT,NAME=SYSPUNCH),DSN=*2
  /FILE UNIT=(TERMOUT,NAME=SYSOUX)
  /FILE UNIT=(TERMOUT,NAME=SYSOUT)
  /FILE UNIT=(TERMOUT,NAME=SYSDBOUT)
  /FILE UNIT=(TERMOUT,NAME=SYSDTERM)
  /FILE UNIT=(LIBIN,NAME=SYSIN),DSN=<input stream>

  /LOAD COBOLVS
```

The statements above the dashed line define files required by the compile step, while the statements below the dashed line define files available to the executing COBOL program.

Compiler Files

The files used during compilation are described below; they are listed by their ddnames, the NAME= subparameter of the UNIT parameter.

File	Use
LINKLIB	Load module library from which compiler modules are fetched.
SYSPRINT	Output listing file containing the compiler options in effect, error messages, and other listings if requested. SYSPRINT is initially dummied out for the compile step when compiling on the terminal, however, any list-producing compiler option (see below the section describing compiler options on the /PARM statement), will cause SYSPRINT to be reset to its previous status, a TERMOUT unit if not otherwise overridden. When compiling on the batch

	SYSPRINT for the compile step is always defaulted to the high speed line printer.
SYSTERM	Output message file containing a compiler progress message and error messages. SYSTERM is used when the TERM compiler option is set, the default when compiling on the terminal.
SYSLIN	Output object file - a temporary work file on which the compiler writes the object code that will be loaded by the loader. Producing this object is controlled by the LOAD/NOLOAD compile option, which defaults to LOAD.
SYSPUNCH	Output object file controlled by the DECK/NODECK option which defaults to NODECK. SYSPUNCH is associated with the *2 file when a compilation is done on the terminal and the card punch when the compilation is done on the batch.
SYSUT1-SYSUT4	Work files used for scratch space by the compiler during its processing.
SYSUT5	Work file used by the compiler when the symbolic debug option SYMDMP is requested - see the debugging section of this chapter (page 37).
SYSIN	Input source and/or object - the input stream from the first COBOL source line to the VPS /DATA statement. See the typical job setup example on page 23.

Execution Time Files

Ten files are defined in the internal VPS procedure for the executing COBOL program. They are:

File	Use
LINKLIB	Load module library which contains execution-time COBOL library subroutines.
SORTLIB	Load library containing the system sort utility modules. This is used only when the sort verb is coded.
SYSUT5	The same compiler utility file as described above. Information from the compile step is used to provide the formatted dump output requested by SYMDMP.
SYSPRINT	Output printout routed to the terminal if the program is executing at the terminal or to the high speed line printer if the program is executing on the batch. To use this predefined file, the ddname

SYSPRINT must be the external file name in the ASSIGN clause of a SELECT statement.

SYSPUNCH — Output associated with the programmer's *2 file on the terminal or the card punch on the batch. This file is used if the UPON SYSPUNCH clause is coded explicitly with the DISPLAY verb, or implied by the mnemonic-name option (see the IBM Language Reference Manual for DISPLAY). The logical record length is fixed at 80 characters, however, the program-id is inserted in positions 73 through 80 by COBOL.

SYSOUX — Output routed as SYSPRINT above. This file is used for DISPLAY when no UPON clause is coded, or when UPON SYSOUT is coded with the DISPLAY, or when UPON mnemonic-name implies SYSOUT (see the IBM Language Reference Manual for DISPLAY). Also the EXHIBIT and READY TRACE verbs produce output on this file. Line width defaults to 121 characters but can be overridden with a user-supplied /FILE statement.

The ddname is a local modification made only to VPS COBOL.

SYSOUT — Printout routed as SYSPRINT. When an internal sort is performed, the sort utility writes its messages to SYSOUT.

SYSDBOUT — Output routed as SYSPRINT. This file is required when any of the symbolic debugging options is specified.

SYSDTERM — Output routed as SYSPRINT. Dynamic dumps are written to SYSDTERM when the PARM SYMDMP is used and when an execution-time control file SYSDBG is provided. See the debugging section of this chapter (page 37).

SYSIN — The input stream after the /DATA control statement. SYSIN will be read when the programmer codes an ACCEPT verb without the FROM option or with FROM and a mnemonic-name implying SYSIN (see the IBM Language Reference Manual on ACCEPT). In this case the logical record size is assumed to be 80 characters.

Also the programmer may specify SYSIN explicitly as the external file name in the ASSIGN clause of a SELECT statement, in which case the logical record size is determined from the 01 size(s) for that file.

The control statements in the COBOL procedure may be overridden or other files defined by supplying /FILE statements before the /LOAD COBOLVS. This partial input stream is a typical COBOL job setup.

```
/FILE UNIT=(DISK,NAME=DONOR),DSN=U.PLG.DONOR.NAMES,DISP=SHR
/FILE UNIT=(PRINTER,NAME=LABLS),CLASS=J,OUTLIM=20000
/LOAD COBOLVS
        IDENTIFICATION DIVISION.
                    .
                    .
                    .
        ENVIRONMENT DIVISION.
                    .
                    .
                    .
        INPUT-OUTPUT SECTION.
        FILE-CONTROL.
            SELECT MASTR ASSIGN S-DONOR.
            SELECT FORMS-OUT ASSIGN S-LABLS.
            SELECT ERRORS-OUT ASSIGN S-SYSPRINT.
            SELECT PARMS-IN ASSIGN S-SYSIN.
                    .
                    .

PRINT LABELS FOR Q-Z
PRINT ERRORS FOR MISSING ADDRESS
```

User files DONOR and LABLS are defined for the application program to be executed as the third step. The /LOAD COBOLVS invokes the VPS internal procedure for COBOL. During compilation, the compiler reads from its SYSIN and obtains the source lines following the /LOAD until /DATA. The VPS loader loads the compiler-produced code for execution and finally the COBOL program is executed. Its SYSIN file, established by the SELECT of the FD PARMS-IN, includes the two lines following the /DATA. The system names in the ASSIGN clause illustrate required information only. For compatibility with older COBOL's other fields such as device type are accepted and treated as comments.

Specifying Compiler Options

As alluded to in the discussion of compiler files, certain processing and listings may be requested or suppressed. This is done by specifying the appropriate keywords on a /PARM statement inserted after the /LOAD COBOLVS but before the program source lines.

```
/PARM option,option...
```

Only one /PARM statement should be used; continuation is not supported. All options are documented in the IBM COBOL Programmer's Guide, but a few are mentioned here, with the defaults underlined.

Option	Description
SOURCE NOSOURCE	controls the printing of the COBOL source module. The default is NOSOURCE on the terminal but SOURCE on the batch.
DECK NODECK	indicates whether the object module is to be written to the compiler's SYSPUNCH file.
LOAD NOLOAD	indicates whether the object module is to be written on SYSLIN.
SXREF NOSXREF	indicates whether or not to produce a sorted cross-reference listing for data-names and procedure names.
CSYNTAX NOCSYNTAX	suppresses object code generation if E-level or higher errors are found, that is, in the case of such errors, the compiler will only scan the source for syntax. The parameter SYNTAX causes this to happen unconditionally.
LANGLVL(1) LANGLVL(2)	indicates which level of ANS COBOL, 1968 or 1974, to use when evaluating those few elements whose meanings changed between the two standards. This parameter has no effect when there is no conflict.
STATE NOSTATE	indicates whether to print the source statement line number of the verb being executed when an abend occurs, such as a data exception.
FLOW[=nn] NOFLOW	indicates whether to print the statement numbers of the last nn paragraphs executed prior to an abnormal termination. While FLOW must be specified at compile time, the number of trace items can be altered at execution time. Also it can be defaulted to 99.
SYMDMP NOSYMDMP	causes the data areas of the executing program to be formatted and dumped when an abend occurs. By supplying a set of control statements under the ddname SYSDBG, snapshot dumps during execution can be generated. See the COBOL Programmer's Guide on Symbolic Debugging.
BATCH NOBATCH	indicates that multiple source programs are to be compiled in a single execution of the compiler. Each program must be preceded by a CBL statement. See the section on calling and called programs in this chapter (page 26).

CLIST NOCLIST	controls printing the condensed listing, that is, the location of the first generated instruction for each verb, the verb, and its source line number.
TERM NOTERM	controls the printing of a progress message and error messages on the SYSTERM file. Currently, on the terminal TERM is the default and SYSPRINT is dummied out during compilation. Any list-producing option, however, resets SYSPRINT'S dummy status. On the batch, NOTERM is the default.

Specifying Loader Options

The VPS /JOB statement may specify options for the loader to use as it processes object in order to build an executable module. A full description of these options appears in Appendix A. Most importantly, a load module map can be requested (MAP) and the load and execute steps can be suppressed (NOGO). The MAP output can be used in conjunction with that from /PARM CLIST to tie an execution program interrupt address back to the offending source statement line. (Note that the powerful debugging feature of the compiler requested by /PARM STATE automates this process). The NOGO loader option is used to compile only, most often with /PARM DECK in preparation to saving the compiler-generated object for later loading and execution. See the section on calling and called programs (page 26).

In addition, the /JOB statement can be used to override the default list of libraries to be searched at load time for unresolved external references. Most often, however, this default list is adequate. It includes these libraries, which are searched in order:

SYS.COBLIB	the IBM supplied COBOL subroutines (these are documented in the IBM COBOL Programmer's Guide).
SYS.VPSLIB	the Computing Center subroutine library (documented on page 34).

The format of /JOB is

```
/JOB option,option...
```

Continuation is not supported but multiple /JOB statements are. The /JOB must follow the /LOAD COBOLVS but precede any COBOL source lines as shown in this example.

```
/LOAD COBOLVS
/JOB MAP
/PARM CLIST,SOURCE,CSYNTAX,SXREF,STATE
       IDENTIFICATION DIVISION.
         .
         .
         .
```

The order of the /JOB and /PARM statements is not restricted.

Calling and Called Programs

Compiling many COBOL source programs in one execution of the compiler is possible by setting the BATCH parameter. Note that this is incompatible with SYMDMP. A CBL statement separates the different source programs and optionally changes the listings desired. The main program, the calling module, should be the first one compiled as shown in the next example.

```
/LOAD COBOLVS
/PARM BATCH,SOURCE
/JOB MAP
  CBL SXREF
       COBOL source for main program
  CBL
       COBOL source for subprogram
/DATA
       test data
```

Alternatively, a program made up of independently compiled modules may be executed partly or wholly from object. Subprograms must be compiled first and their object saved in the VPS library. If the main program is to be compiled and executed, the subprogram object must be included at that time.

The process is shown in this example. Assume the following file has been setup for a COBOL subprogram. It should be executed first.

```
/LOAD COBOLVS
/PARM DECK
/JOB NOGO
       source statements for the subprogram
```

As discussed previously, DECK causes the compiler to write the object module on its SYSPUNCH file, which on the terminal is the user's *2 file. /JOB NOGO insures that no load or execute will be attempted for the subprogram, as it is not an independent element. After the subprogram has been compiled successfully, the object code in *2 should be saved in a permanent file by issuing the /RSAV command.

```
/RSAV objfile
```

This will save (or replace) the VPS library file named "objfile" with the contents of *2. "Objfile" must be a name different from the subprogram source file name, or the source will be lost. Obviously if the subprogram is changed, the above steps must be repeated.

The calling-called program unit may be tested by running the following file.

```
/LOAD COBOLVS
/PARM STATE
/JOB MAP
       COBOL source for main program
/INC objfile
/DATA
data ...
```

"Objfile" is the name chosen on the /RSAV command. Many object decks may be saved as above and then combined when the main program is run by using a /INCLUDE of the form:

```
/INC objfile1,objfile2,objfile3...
```

It is also possible to save a main program object separately and avoid compilation altogether at execution time. Job setup is the same as for the compile, load, and execute of the main program, only no source appears; multiple object modules are included. For a program that is rarely modified, one may consider creating a VPS load module and avoiding some of the loader processing as well. Appendix B contains information on creating load modules from object and executing these load modules in the VPS "OS/VS" environment.

/PARM for User Execution

If a COBOL program is coded to accept an execution-time parameter (see the IBM Language Reference Manual), the value can be passed on a VPS /PARM statement. The COBOL source must be delimited by a /DATA statement ,even if no execution-time SYSIN data exists, which must then be followed by the /PARM.

```
/LOAD COBOLVS
/PARM SOURCE
       IDENTIFICATION DIVISION.
            .
            .
            .
       LINKAGE SECTION.
       01  PRM.
           05  P-LENGTH  PIC  9(04)  USAGE COMP.
           05  P-VAL.
               10  P-FUNC        PIC  X(06).
               10  P-SUBFUNC     PIC  X(06).
       PROCEDURE DIVISION USING PRM.
            .
            .
            .
/DATA
/PARM 'print full
data for program SYSIN
```

Note that some of the COBOL run-time routines accept parameters also. As documented in the COBOL Programmer's Guide, these are specified after a slash within the PARM. See the IBM manual for further details.

COBOL I/O Considerations

Support or lack of support for the different input/output options of COBOL is summarized in the following chart. All of the input/output elements recognized by the compiler are listed, including those language extensions to 1974 COBOL incorporated from IBM OS Full American National Standard COBOL (and documented in Appendix A of the current Language Reference Manual).

I/O SUPPORTED				NOT SUPPORTED
FILE ORGANIZATION	RECORDING MODE	ACCESS MODE	OPEN	
S SEQUENTIAL	F FIXED BLOCKED OR UNBLOCKED V VARIABLE BLOCKED OR UNBLOCKED U UNDEFINED	SEQUENTIAL	INPUT OUTPUT	RECORDING MODE S (SPANNED) OPEN I-O OPEN INPUT REVERSED
D,W DIRECT	F FIXED UNBLOCKED V VARIABLE UNBLOCKED U UNDEFINED	SEQUENTIAL RANDOM	INPUT OUTPUT I-O	RECORDING MODE S
R RELATIVE	F FIXED UNBLOCKED	SEQUENTIAL RANDOM	INPUT OUTPUT I-O	
I INDEXED				NO SUPPORT
AS VSAM SEQUENTIAL				NO SUPPORT
VSAM INDEXED OR RELATIVE				NO SUPPORT

Physical Sequential Files

Sequential files must be read or written in order, that is, if record 126 is to be accessed then records 1 through 125 must be accessed first. On VPS, sequential files may be associated with any valid unit type: library files, disk work files, tapes, virtual printer or punch files, terminal input or output, and the input stream. As the input/output chart indicates spanned records are not supported; records in disk sequential files cannot be rewritten; and tapes cannot be read backwards. See the full sample program at the end of this chapter (page 39) for examples of sequential files in VPS COBOL.

The parameters on the /FILE statement that may pertain to a sequential file are summarized below. See the VPS Handbook for a full description of them.

Parameter	When Required
UNIT=(type,NAME=aaaaaaaa[,COND])	always required
DSN=dddddddd	required for permanent disk files
DISP=(OLD,KEEP ,KEEP) SHR,DELETE,DELETE NEW MOD	required if not the default
BLKSIZE=n	required for new files when BLOCK CONTAINS 0 RECORDS is specified
LRECL=n	same as for BLKSIZE
LEVELS=n	optional for library input units
SPACE=(REC,(n,m)) BLK TRK CYL	required for new disk work files
DEN=n	required for tape output with BLP
LABEL=(n,NL) BLP	required for tape only
VOL=SER=vvvvvv	required for tape only
CLASS=c	optional, for virtual unit record devices
OUTLIM=n	same as for CLASS
COPIES=n	same as for CLASS

Direct Files

A record in a direct file may be accessed individually without processing all previous records. This requires some unique identifier, a key, that can be associated with each record. Direct files must exist on disk, and on VPS they must be work files. The record identifier is manipulated according to an algorithm to yield a relative track number within predefined file limits, 0 to n tracks.

The ACTUAL KEY data-name specifies an area containing the calculated track identifier and the unique record key. In the COBOL Programmer's Guide a common randomizing technique to transform a key into a relative track number is discussed. Depending on the range of key values the algorithm may yield more records with the same relative track address than can be held on one track. As per OS, VPS provides the option to try succeeding track(s) until an empty slot is found. This is called extended search and can be specified using the LIMCT=n parameter of the VPS /FILE statement (where n is the number of tracks to be searched).

When reading the COBOL Programmer's Guide on direct files, note that the disk format of spanned records is not supported by VPS. Also note that multivolume direct files are not supported.

The /FILE parameters that may apply to a file with direct organization are:

Parameter	When Required
UNIT=(DISK,NAME=aaaaaaaa[,COND])	always required
DSN=dddddddd	required for permanent files
DISP=(OLD,KEEP ,KEEP) SHR,DELETE,DELETE NEW	required if not the default
SPACE=(REC,(n,m)) BLK TRK CYL	required for new direct files (secondary allocation is meaningful only during file create)
DSORG=DA	required except when access is sequential for an input
LIMCT=n	number of tracks beyond the supplied relative track to search. This should be identical on the file statements for the create program as well as for subsequent access programs.

OPT=FORMAT does not apply to direct files created with COBOL as COBOL insures that the file will be formatted. See the IBM Programmer's Guide for details on file create and the type of

formatting performed for the different record types, fixed or variable and undefined.

In the following example, a direct file of error messages is created from card input containing an error code internal to the application and the corresponding text message that a user would see on a final error report. The calculation of relative track number is quite simple, because the structure of the record key happens to be simple and because the number of possible messages is small. The programmer must determine track capacity for the particular disk device to be used from an IBM Reference Summary card; currently the Computing Center has 3350's and the corresponding card is GX20-1983. Fifty-five 80 character records with keys will fit on one 3350 track and so the sample allocation of 10 tracks will hold a maximum of 550 records.

```
/file unit=(disk,name=emsg),dsn=u.plg.emsg.direct,
/  disp=(new,keep,delete),
/  space=(trk,10),dsorg=da,limct=10
/load cobolvs
/parm source,csyntax
/job map
       identification division.
       program-id.  bs001075.
       remarks.   load error message text file.
       environment division.
       configuration section.
         source-computer.  ibm-370.
         object-computer.  ibm-370.
       input-output section.
       file-control.
           select card-input assign s-sysin.
           select error-msgs assign d-emsg
               access mode is random
               actual key is act-key.
       data division.
       file section.
       fd  card-input
           label records standard.
       01  error-rec-in.
           05  error-cd-in.
               10  error-sub-systm pic  x(01).
               10  error-cd-dig    pic  x(02).
           05  error-text-in       pic  x(77).
       fd  error-msgs
           label records standard.
       01  e-msg-rec       pic  x(80).
       working-storage section.
       01  misc.
           05  rec-in      pic  9(07)  usage comp-3  value zeroes.
           05  work-sub-systm     pic  9(01).
       01  act-key.
           05  trk-id      pic s9(05)  usage comp.
           05  rec-id.
               10  sub-systm-id pic  x(01).
               10  error-cd     pic  9(02).
```

```
       procedure division.
       m-001-init.
           move zeroes to trk-id.
           move zeroes to rec-id.
           open output error-msgs
               input card-input.
       m-005-read.
           read card-input at end go to m-050-eoj.
      *
      *    input may not be in order, but there will
      *    be no duplicates
      *
           add 1 to rec-in.
           display error-rec-in.
           move error-cd-in of card-input to
                rec-id      in act-key.
           move error-sub-systm in card-input to
                 work-sub-systm.
           move work-sub-systm to trk-id.
           if work-sub-systm not numeric
              then subtract 1 from trk-id.
      *
      *    sub-systm-id may be alpha, but strip zone
      *    bits and use for relative track number -
      *    all letters, numbers will then be in the
      *    0 thru 9 range
      *
           move error-rec-in to e-msg-rec.
           write e-msg-rec invalid key
               display 'invalid key on '
                   trk-id ' ' rec-id
               stop run.
           go to m-005-read.
       m-050-eoj.
           close error-msgs card-input.
           display rec-in ' messages loaded'.
           stop run.
```

Relative Record Files

Relative files can also be accessed randomly, however, there is no unique record key. The access is by record number within a file, 0 to n records. As for direct, relative files must be VPS work files. The NOMINAL key clause provides the relative record number of the record to be read or written, and system routines calculate the location on disk.

The parameters of the /FILE statement that pertain are nearly those for direct files. They are:

Parameter	When Required
UNIT=(DISK,NAME=aaaaaaaa[,COND])	always required

DSN=ddddddd	required for permanent files
DISP=(OLD,KEEP ,KEEP) SHR,DELETE,DELETE NEW	required if not the default
SPACE=(REC,(n,m)) BLK TRK CYL	required for new relative files (secondary allocation is meaningful only during file create)
DSORG=DA	required except when access is sequential for an input

A modification of the specifications for the error message file in the previous section would suit it to relative organization. If the error codes were assigned only numeric values, this sample program would create a relative file. The 8 tracks (of 3350) allocated will hold 72 80-character records per track and thus the file capacity is 576 records. The IBM 3350 Direct Access Storage Reference Summary (GX20-1983) indicates number of records per track for files without keys.

```
/file unit=(disk,name=emsg),dsn=u.plg.emsg.relative,
/  disp=(new,keep,delete),
/  dsorg=da,space=(trk,8)
/load cobolvs
/parm source,csyntax
/job map
       identification division.
       program-id.  bs001075.
       remarks.   load error message text file.
       environment division.
       configuration section.
         source-computer.  ibm-370.
         object-computer.  ibm-370.
       input-output section.
       file-control.
           select card-input assign s-sysin.
           select error-msgs assign r-emsg
               access mode is sequential
               nominal key is nom-key.
       data division.
       file section.
       fd  card-input
           label records standard.
       01  error-rec-in.
           05  error-cd-in    pic  9(03).
           05  error-text-in  pic  x(77).
       fd  error-msgs
           label records standard.
       01  e-msg-rec.
           05  error-cd-ou  pic  9(03).
           05  r-error-cd  redefines  error-cd-ou.
               10  dumm-cd  pic  x(01).
               10  filler   pic  x(02).
```

```
        05  error-text-ou pic x(77).
    working-storage section.
    01  misc.
        05  rec-in      pic  9(07)  usage comp-3  value zeroes.
        05  save-er-cd  pic  x(03)  value zeroes.
    01  nom-key         pic s9(08)  usage comp.
    procedure division.
    m-001-init.
        move zeroes to nom-key.
        open output error-msgs
            input card-input.
    m-005-read.
        read card-input at end go to m-050-eoj.
        if error-cd-in not numeric
            then display 'invalid error code '
                error-cd-in
            stop run.
    *
    *   input has been presorted
    *
        if error-cd-in not greater than save-er-cd
            then display 'sequence error '
                error-cd-in ' last input ' save-er-cd
            stop run.
        add 1 to rec-in.
        move error-cd-in to save-er-cd nom-key.
        display error-rec-in.
        move error-rec-in to e-msg-rec.
        write e-msg-rec invalid key
            display 'invalid key on '
                nom-key
            stop run.
        go to m-005-read.
    m-050-eoj.
        close error-msgs card-input.
        display rec-in ' messages loaded'.
        stop run.
```

Computing Center Supplied Subroutines

The Computing Center maintains a common subroutine library, SYS.VPSLIB. Routines that may be useful to the COBOL programmer are described below by function.

1) To obtain the current execution time of a job -

```
CALL 'EXTIME' USING data-name-1.
```

"data-name-1" is a full word binary number that will receive the current execution time value in units of one hundredths of a second.

```
01  data-name-1     PIC 9(09) USAGE COMP.
```

2) To specify a VPS library file name to be read from a LIBIN unit -

```
CALL 'OPEN' USING
      index-name
      file-name
      dd-name
      ret-code
      levels.
```

The arguments are defined as follows:

```
01  index-name    PIC X(8).
01  file-name     PIC X(8).
01  dd-name       PIC X(8).
01  ret-code      PIC 9(9) USAGE COMP.
01  levels        PIC 9(9) USAGE COMP.
```

"index-name" contains the library index name. If the field is blank the default is USERLIB.

"file-name" contains the library file name.

"dd-name" contains the COBOL external file name as specified in the SELECT statement. (This corresponds to the NAME= subparameter on the /FILE statement.)

"ret-code" contains the sum of the return codes that the subroutine is to ignore. If "ret-code" has a value of zero or if it is omitted and an error occurs, or if an error occurs which is not specified in "ret-code", a message will be printed and the job abnormally terminated. Otherwise, "ret-code" will contain the actual return code. The return codes are:

```
  0 - successful open
  2 - invalid unit number (type not LIBIN)
  4 - invalid file name
  8 - file not found
 16 - index not found
 32 - access to file disallowed
 64 - unable to enqueue
128 - library volume not mounted
256 - permanent I/O error
512 - storage unavailable for processing
```

Serious error conditions that will always cause abnormal termination are:

```
1 - no corresponding /FILE statement
3 - number of parameters in call in error
5 - argument list invalid
```

"levels" contains the resolution level for any /INCLUDE statement occurring in the file. If omitted the default for that file will be used. (See the VPS Handbook for a complete description of the resolution level.)

To use this subroutine the programmer must supply a corresponding /FILE statement of the format:

```
/FILE UNIT=(LIBIN,NAME=ddname)
```

where ddname is the external file name in the SELECT statement in the COBOL program.

3) To save or replace a VPS library file with the contents of a LIBOUT unit, or to rewind a LIBOUT unit -

```
CALL 'CLOSE' USING
      index-name
      file-name
      dd-name
      type-code
      ret-code.
```

The arguments are defined as follows:

```
01  index-name    PIC X(8).
01  file-name     PIC X(8).
01  dd-name       PIC X(8).
01  type-code     PIC X(1).
01  ret-code      PIC 9(9) USAGE COMP.
```

"index-name" contains the library index name. If the field is blank the default is USERLIB.

"file-name" contains the library file name.

"dd-name" contains the COBOL external file name as specified in the SELECT statement. (This corresponds to the NAME= subparameter on the /FILE statement.)

"type-code" contains one of the following characters indicating the service to be performed.

S - Save the information as a new VPS library file, then clear and rewind the LIBOUT unit.

R - Replace an existing VPS library file with the information, then clear and rewind the LIBOUT unit.

D - Clear and rewind the associated LIBOUT unit without saving the contents.

"ret-code" contains the sum of the return codes that the subroutine is to ignore. If "ret-code" has a value of zero or if it is omitted and an error occurs, or if an error occurs which is not specified in "ret-code", a message will be printed and the job abnormally terminated. Otherwise, "ret-code" will contain the actual return code. The return codes are:

0 - successful close
2 - invalid unit number (type not LIBOUT)
4 - invalid file name
8 - null output file
16 - index not found
32 - write access to file disallowed
64 - name already in use
128 - library space not available
256 - permanent I/O error
512 - storage unavailable for processing
1024- no space in index

Serious error conditions that will always cause abnormal termination are:

1 - no corresponding /FILE statement
3 - number of parameters in call in error
5 - argument list invalid
7 - type parameter invalid or missing

To use this subroutine the programmer must supply a corresponding /FILE statement of the format:

```
/FILE UNIT=(LIBOUT,NAME=ddname)
```

where ddname is the external file name in the SELECT statement in the COBOL program.

Debugging

Enhanced debugging facilities are available as OS/VS COBOL options. As documented in the section on the /PARM statement (page 24) STATE, FLOW, and SYMDMP can be activated. All of the required files are included in the VPS internal procedure for COBOL. STATE causes run-time routines to display the line number of the COBOL statement in control at the time of an abnormal termination. FLOW will display a trace of the statement numbers of the last nn paragraphs executed before an abend (where nn is 1 to 99). With the COBOL procedure file statements only, that is, with no SYSDBG defined, SYMDMP triggers a formatted dump of COBOL data areas upon abnormal termination. This includes working-storage variables, file control information, and buffers. In addition by providing the appropriate debug control statements in SYSDBG, periodic checkpoint dumps can be taken to the SYSDTERM file. See the IBM COBOL Programmer's Guide for more about this feature.

Certain source language debugging statements are also available: EXHIBIT, DISPLAY, and READY TRACE verbs and the ON conditional statement. EXHIBIT and DISPLAY are used to show the values of data-names during execution. READY TRACE shows program flow by listing the source statement line numbers of paragraphs that are invoked. Since this output could be exceedingly large, RESET TRACE is provided to terminate the tracing function. Also, using the ON statement qualifies the diagnostic verbs so as not to generate too

much output. For example to display some of the values of the variable PATIENT-BALANCE the programmer should code:

```
ON 5 AND EVERY 5 UNTIL 100 EXHIBIT NAMED PATIENT-BALANCE.
```

See the IBM COBOL Language Reference Manual for the exact formats of these debugging statements. The default file for DISPLAY as well as for EXHIBIT and READY TRACE output has the ddname SYSOUX and is a TERMOUT unit if not overridden by the job stream.

Support Qualifications

For a summary of support of input/output elements see the chart on page 28. In addition, the following facilities are unavailable to the VPS COBOL programmer:

segmentation feature
error declaratives
label declaratives

There are no compile-time incompatibilities between VPS and OS/VS COBOL, that is, in all cases the same source program can be compiled on either system to yield identical object code. The COPY and BASIS compiler directives are supported for compatibility with other installations, even though those same functions are served by the VPS /INCLUDE statement. Execution on VPS will be terminated for I/O facilities not supported and the segmentation feature. Error and label declaratives will simply be ignored as execution continues. All other features of the IBM OS/VS COBOL compiler not specified are available in VPS COBOL. This includes the internal sort and the report writer features. The full sample program at the end of this chapter contains a sort and the job control statements necessary to run it.

There are a few miscellaneous pitfalls that, while they are not the result of lack of support by VPS, are mentioned here because of their frequency of occurrence. Be aware that WRITE BEFORE ADVANCING generates machine code carriage control characters, and so, as documented in the VPS Handbook, will work only on the batch. Also note that it is an option (ADV), and the default at this installation, for the compiler to reserve the first character of a print buffer for the carriage control character; thus it is not part of the programmer's 01-level record description. Another compiler default is the use of apostrophes to delineate literals. This can be changed to quotes by saying QUOTE on a /PARM statement. Also, the IBM sort utility currently in use at the Academic Computing Center is OS SORT/MERGE. As stated in the COBOL Programmer's Guide, linkage to this sort utility from OS/VS COBOL programs sometimes results in a harmless sort message IGH067I INVALID EXEC OR ATTACH PARAMETER.

Sample Program

```
/file unit=(disk,name=chrg),dsn=u.plg.db,disp=(new,keep),
/  space=(rec,(50,50)),blksize=4480,lrecl=160
/file unit=(disk,name=pay),dsn=u.plg.cr,disp=(new,keep),
/  space=(rec,(50,50)),blksize=4480,lrecl=160
/file unit=(disk,name=sortwk01),space=(trk,5)
/file unit=(disk,name=sortwk02),space=(trk,5)
/file unit=(disk,name=sortwk03),space=(trk,5)
/load cobolvs,nsegs=8
/parm csyntax,state
       identification division.
       program-id.  bs001005.
       remarks.   split input by transaction type into 2 files
                  -  charges, charge credits, adjustment debits in one
                  and payments, adjustment credits in the other -
                  unidentified transactions are put in the former
                  with the appropriate error code.
       environment division.
       configuration section.
         source-computer.  ibm-370.
         object-computer.  ibm-370.
       input-output section.
       file-control.
           select card-input assign s-sysin.
           select print-out assign s-sysprint.
           select sort-f1 assign s-sortwk01.
           select chrg-file assign s-chrg.
           select pay-file assign s-pay.
       data division.
       file section.
       fd  card-input
           label records standard.
       01  trans-in          pic  x(80).
       fd  print-out
           label records standard.
       01  print-line.
           05  c-control    pic  x(001).
           05  data-line    pic  x(132).
       sd  sort-f1.
       01  trans-rec.
           05  trans-type   pic  x(02).
               88  charge   value '01'.
               88  adj-db   value '03'.
               88  chg-cr   value '05'.
               88  pay      value '07'.
               88  adj-cr   value '09'.
           05  trans-date.
               10  yy       pic  x(02).
               10  mm       pic  x(02).
               10  dd       pic  x(02).
           05  trans-acct   pic  x(04).
           05  filler       pic  x(60).
           05  trans-amt    pic  9(06)v99.
       fd  chrg-file
```

```
    label records are standard
    block contains 0 records.
01  chrg-out.
    05  c-rec        pic  x(80).
    05  filler       pic  x(77).
    05  error-cd     pic  x(03).
fd  pay-file
    label records are standard
    block contains 0 records.
01  pay-out.
    05  p-rec        pic  x(80).
    05  filler       pic  x(80).
working-storage section.
01  ws-con.
    05  error-con    pic  x(03)  value 'b99'.
01  hd-line.
    05  filler       pic  x(01)  value  '1'.
    05  c-date       pic  x(08).
    05  filler       pic  x(48)  value  spaces.
    05  filler       pic  x(20)
                     value  'input control totals'.
01  sum-line.
    05  filler       pic  x(01)  value '-'.
    05  filler       pic  x(05)  value 'input'.
    05  pr-rec-in    pic  z(07).
    05  filler       pic  x(10)  value spaces.
    05  filler       pic  x(07)  value 'charges'.
    05  pr-rec-c     pic  z(07).
    05  filler       pic  x(10)  value spaces.
    05  filler       pic  x(08)  value 'payments'.
    05  pr-rec-p     pic  z(07).
    05  filler       pic  x(10)  value spaces.
    05  filler       pic  x(09)  value 'undefined'.
    05  pr-rec-u     pic  z(07).
01  kounters         usage comp-3.
    05  rec-in       pic  9(07)  value zero.
    05  rec-out-c    pic  9(07)  value zero.
    05  rec-out-p    pic  9(07)  value zero.
    05  rec-out-u    pic  9(07)  value zero.
procedure division.
main-line section.
m-01.
    sort sort-f1
        ascending key trans-acct trans-date
        descending key trans-type
        using card-input
        output procedure is test-recs.
m-99.
    if sort-return equal zero
        then display 'sort successful'
        else display 'sort failed'.
    stop run.
test-recs section.
o-01.
    open output  pay-file chrg-file.
```

```
       o-05.
           move spaces to pay-out chrg-out.
           return sort-f1 record at end go to o-50.
           add 1 to rec-in.
           if adj-cr or pay
               then move trans-rec to p-rec
                   write pay-out
                   add 1 to rec-out-p
               else move trans-rec to c-rec
                   if charge or adj-db or chg-cr
                       then add 1 to rec-out-c
                           write chrg-out
                       else add 1 to rec-out-u
                           move error-con to error-cd
                           write chrg-out.
           go to o-05.
      *
       o-50.
           close pay-file chrg-file.
           open output print-out.
           move current-date to c-date.
           write print-line from hd-line
               after positioning c-control.
           move rec-out-c to pr-rec-c.
           move rec-out-p to pr-rec-p.
           move rec-out-u to pr-rec-u.
           move rec-in to pr-rec-in.
           write print-line from sum-line
               after positioning c-control.
           close print-out.
/data
test data
     .
     .
     .
```

Chapter 5: FORTRAN G1 and H Extended

Three FORTRAN compilers are supported under VPS. The IBM Program Products, FORTRAN G1 and FORTRAN H Extended, which are discussed in this chapter and WATFIV, written at the University of Waterloo. (FORTRAN G, an older version of FORTRAN G1 also available on VPS, is not documented in this manual since it is no longer supported by IBM.) WATFIV, discussed in Chapter 10, supports a superset of the FORTRAN language and is intended mainly for student use since it provides rapid compilation of small programs and excellent error message facilities.

The FORTRAN G1 and H Extended compilers provide full FORTRAN language support and have not been modified to run under VPS - rather, special VPS system support programs provide an "OS/VS" environment in which these compilers and their compiled programs run. VPS FORTRAN programs are, therefore, source and object compatible with OS/VS FORTRAN programs. Programs can be written and tested under VPS and run under OS/VS with only control card changes.

IBM Manuals

This chapter is not a replacement for the documentation available in IBM publications. It is a supplement to the documentation explaining necessary VPS control statement usage for these compilers. FORTRAN programmers are encouraged to use the following publications, which can be found in reference racks at the Computing Center.

The FORTRAN supported by these compilers is documented in -

"IBM System/360 and System/370 FORTRAN IV Language" (GC28-6515)

Information on compiler options, extended error handling, debugging specifications, etc. can be found in -

"IBM OS Code and Go FORTRAN and FORTRAN IV (G1) Programmer's Guide" (SC28-6853)

"IBM OS FORTRAN IV (H Extended) Compiler Programmer's Guide" (SC28-6852)

Error messages can be found in -

"IBM OS FORTRAN IV (H Extended) Compiler and Library (Mod II) Messages" (SC28-6865)

Standard IBM supplied subroutines are discussed in -

"IBM System/360 Operating System: FORTRAN IV Library - Mathematical and Service Subprograms" (GC28-6818)

"IBM System/360 OS FORTRAN IV Mathematical and Service Subprograms Supplement for Mod I and Mod II Libraries" (SC28-6864)

Compiler differences are discussed in -

"IBM FORTRAN Program Products for OS and the CMS Component of VM/370 General Information" (GC28-6884)

Choosing a Compiler

Both FORTRAN compilers, besides supporting the full language, provide list directed I/O (free-format input and output, for more information see the Language Manual) and extended error handling (see the Programmer's Guide). H Extended also provides two levels of object code optimization, extended precision arithmetic, and automatic function selection. However, G1 provides a powerful Debug facility and is purported to be a faster compiler.

Therefore, unless language differences prevent it, programs which require frequent modification or programs which are being developed and need the Debug facility should be compiled using FORTRAN G1. Large programs which require little modification, can be run from object or a load module, and can take advantage of optimization should be compiled with FORTRAN H Extended. Note that these are not hard and fast rules, and the programmer is encouraged to become familiar with both compilers and make individual decisions on choosing a compiler relative to the requirements of the program.

Invoking the Compilers

A compiler is invoked by placing a VPS /LOAD control statement (see the VPS Handbook for a full description of /LOAD) before the first FORTRAN source statement in a job stream. Normally, this is done by placing the /LOAD statement first in a source file with the VPS editor when the source file is created. The format of the /LOAD control statement used to invoke the FORTRAN G1 compiler is:

```
/LOAD FORTG1
```

The format of the /LOAD statement used to invoke the FORTRAN H Extended compiler is:

```
/LOAD FORTX
```

In reality, not only does the /LOAD statement call the correct compiler, but it also invokes a corresponding VPS internal procedure. A VPS procedure is a set of VPS control statements needed to run average source programs. These control statements may be overridden at execution time by programmer-supplied statements. The FORTRAN G1 procedure invoked using /LOAD FORTG1 at the terminal is shown below:

```
/FILE UNIT=(DISK,NAME=LINKLIB),DSNAME=<fortran library>
/FILE UNIT=(DUMMY,NAME=SYSPRINT)
/FILE UNIT=(LIBIN,NAME=SYSIN),DSNAME=<input stream>
/FILE UNIT=(DISK,NAME=SYSLIN),SPACE=(REC,(1000,1000)),RECFM=FB,
/ LRECL=80,BLKSIZE=4560
/FILE UNIT=(LIBOUT,NAME=SYSPUNCH),DSNAME=*2
/FILE UNIT=(TERMOUT,NAME=SYSTERM)
-----
/FILE UNIT=(TERMOUT,NAME=FT06F001)
/FILE UNIT=(LIBIN,NAME=FT05F001),DSNAME=<input stream>
/FILE UNIT=(LIBOUT,NAME=FT07F001),DSNAME=*2
/FILE UNIT=(TERMIN,NAME=FT09F001)
/LOAD FORTG1
```

The FORTRAN H Extended terminal procedure is shown below:

```
/FILE UNIT=(DISK,NAME=LINKLIB),DSNAME=<fortran library>
/FILE UNIT=(DUMMY,NAME=SYSPRINT)
/FILE UNIT=(LIBIN,NAME=SYSIN),DSNAME=<input stream>
/FILE UNIT=(DISK,NAME=SYSLIN),SPACE=(REC,(1000,1000)),RECFM=FB,
/ LRECL=80,BLKSIZE=4560
/FILE UNIT=(DISK,NAME=SYSUT1),SPACE=(BLK,(3,3)),RECFM=F,
/ LRECL=3465,BLKSIZE=3465
/FILE UNIT=(DISK,NAME=SYSUT2),SPACE=(BLK,(10,10)),RECFM=F,
/ LRECL=5048,BLKSIZE=5048
/FILE UNIT=(LIBOUT,NAME=SYSPUNCH),DSNAME=*2
/FILE UNIT=(TERMOUT,NAME=SYSTERM)
-----
/FILE UNIT=(TERMOUT,NAME=FT06F001)
/FILE UNIT=(LIBIN,NAME=FT05F001),DSNAME=<input stream>
/FILE UNIT=(LIBOUT,NAME=FT07F001),DSNAME=*2
/FILE UNIT=(TERMIN,NAME=FT09F001)
/LOAD FORTX,NSEGS=4
```

In both procedures /FILE statements appearing above the dashed line define files used by the compiler. /FILE statements appearing below the dashed line define standard files used by an executing FORTRAN program.

Compiler Files

Except for the SYSUT1 and SYSUT2 files used by FORTRAN H Extended, both compilers have the same file requirements as listed below:

File	Use
LINKLIB	FORTRAN Compiler Library - defined as the work file containing the particular FORTRAN compiler.

SYSPRINT	Output listing file - defined as DUMMY if compiling at the terminal (unless a list-producing option is specified on the /PARM statement - see below) or as the high speed printer if compiling on the batch.
SYSIN	Input source and/or object file - defined as the input stream from the first source statement to the /DATA statement (see below).
SYSLIN	Output object file - defined as a temporary work file used to hold the object code to be loaded by the loader for execution.
SYSPUNCH	Output object file - defined as the *2 file if co piling at the terminal or the card punch if compiling on the batch allowing object code to be saved for future use (used only if DECK appears on the /PARM statement).
SYSTERM	Output message file - defined as the terminal (used only if the compilation occurs at the terminal).

FORTRAN H Extended only:

SYSUT1	Temporary work file used if either FORMAT or TERMINAL appears on the /PARM statement.
SYSUT2	Temporary work file used if XREF appears on the /PARM statement.

Execution Time Files

Both procedures provide /FILE statements for standard FORTRAN units 5, 6, 7, and 9. FORTRAN units, or reference numbers, refer to files defined on /FILE statements. The format of the NAME= subparameter of the UNIT= parameter is as follows:

```
NAME=FTxxF001
```

Where 'xx' is the FORTRAN unit number. Therefore, FORTRAN unit 6 is FT06F001, FORTRAN unit 14 is FT14F001, etc.

Because of the unique nature of VPS execution facilities the default unit 5 must be discussed separately. If the default /JOB statement is used (see below), after the FORTRAN compiler is finished compiling the source program, the VPS loader loads the object code from SYSLIN into memory and the program is executed. Unit 5 (the /FILE statement having NAME=FT05F001 in the UNIT= parameter) is defined as any records following the /DATA control statement. The /DATA statement is used to separate source statements in the input stream (which are read by the compiler using SYSIN) from data to be read at program execution time using unit 5. A standard FORTRAN compilation and execution has the following structure:

```
/LOAD FORTG1

    FORTRAN source statements

/DATA

    data
```

The default execution time /FILE statements in both the FORTRAN G1 and H Extended procedures are shown below:

File	Use
FT05F001	FORTRAN unit 5 - defined as the input stream after the /DATA statement.
FT06F001	FORTRAN unit 6 - defined as terminal output if executing at the terminal or the high speed printer if executing on the batch.
FT07F001	FORTRAN unit 7 - defined as the *2 file if executing at the terminal or as the card punch if executing on the batch.
FT09F001	FORTRAN unit 9 - defined as terminal input if executing at the terminal or the input stream if executing on the batch.

These files may be overridden or other files defined by placing appropriate /FILE statements before the /LOAD statement in the job stream. In the following example, a programmer wishes to read data from a tape file using unit 12 and from standard unit 5:

```
/file unit=(tape,name=ft12f001),vol=ser=mastdt,disp=old,
/ label=(1,nl),recfm=fb,lrecl=120,blksize=1200
/load fortx
      tot1=0
      tot2=0
      tot3=0
  10  read(5,20,end=40) a,b,c
  20  format(3f5.0)
      read(12,30) aa,bb,cc
  30  format(3f5.2)
      tot1=a+aa
      tot2=b+bb
      tot3=c+cc
      go to 10
  40  write(6,50) tot1,tot2,tot3
  50  format('the totals are:',1x,3f6.2)
      stop
      end
/data
  66.  43.  87.
  49. 192. 205.
```

Specifying Compiler Options (/PARM)

The VPS /PARM control statement may be used to pass compiler options to the respective FORTRAN compiler. The /PARM statement must be placed in the job stream after the /LOAD statement and before the first FORTRAN source statement (note that /PARM and /JOB may appear in either order). The format of the /PARM statement is shown below:

```
/PARM option,option,...
```

Where "option" is a valid compiler option. If more than one option is specified, they must be separated by commas. Multiple /PARM statements are not supported.

The following is a subset of the compiler options accepted by the FORTRAN G1 and H Extended compilers. A full description of all the options for each compiler can be found in the respective IBM Programmer's Guide. Note that the default options are underlined.

Option	Description
SOURCE NOSOURCE	Controls the printing of a FORTRAN source listing on SYSPRINT.
DECK NODECK	Controls the writing of the object code to SYSPUNCH.
MAP NOMAP	Controls the printing of the storage and statement label map on SYSPRINT.
LIST NOLIST	Controls the printing of the assembler language listing of the compiled program on SYSPRINT.
EBCDIC BCD	EBCDIC (the default) specifies that the source program is in EBCDIC. BCD specifies that it is in BCD.

FORTRAN G1 only -

LOAD NOLOAD	Controls the writing of the object code to SYSLIN for use by the loader.

FORTRAN H Extended only -

OBJECT NOOBJECT	Controls the writing of the object code to SYSLIN for use by the loader.
XREF NOXREF	Controls the printing of the label and variable cross reference on SYSPRINT.

OPT(0) OPT(1) OPT(2) NOOPT	Controls the optimization level to be in force during compilation. NOOPT (the default) indicates that no optimization is to be performed and is equivalent to OPT(0). OPT(1) specifies that each source module is to be treated as a single program loop and that it is to be optimized with regard to register allocation and branching. OPT(2) specifies that each source module is to be treated as a collection of program loops and each loop is to be optimized with regard to register allocation, branching, common expression elimination, and replacement of redundant computations.

Specifying Loader Options (/JOB)

The VPS /JOB control statement may be used to specify loader options for compiled programs which are to be loaded for execution. (A full description of the /JOB statement can be found in Appendix A.) The /JOB statement can be used to control the printing of a load map, unresolved external references, entry address, etc. It can be used to terminate a job after compilation but before execution (NOGO). It can also be used to override the default list of libraries to be searched at load time for unresolved external references (such as mathematical and graphics subroutines existing in VPS subroutine libraries). The default FORTRAN library search list is -

SYS.FORTLIBX - the IBM supplied support and mathematical subroutine library

SYS.TEKLIB - the TEKTRONIX graphics subroutine package

SYS.VPSLIB - the Computing Center subroutine library

The libraries are searched in the order specified.

The /JOB statement must be placed in the job stream after the /LOAD statement but before the first FORTRAN source statement. In the following example the /JOB statement is used to terminate a job immediately after compiling a FORTRAN program:

```
/LOAD FORTG1
/JOB NOGO
     FORTRAN source statements
```

Note that though continuation is not supported for the /JOB statement, multiple /JOB statements are permissible.

The Strange Behavior of SYSPRINT

Normally, when executing a job on the terminal, the SYSPRINT file is defined as TERMOUT. For FORTRAN compilations at terminals the SYSPRINT file is dummied out to bypass massive amounts of output being printed at the terminal during compilation. However, there are circumstances where the programmer may wish to direct SYSPRINT to a location other

than the terminal (a library file, the high speed printer, a work file, etc.). To allow this, VPS will not dummy SYSPRINT if a list-producing option appears on a /PARM statement in the job stream. A list-producing option is defined as any compiler option which will cause printing to occur on SYSPRINT. In the following example the FORTRAN H Extended compiler is used to compile a program where the source and cross reference listing is to be printed on the high speed printer, the object code is to be written to the *2 file, and job execution is to be terminated after compilation - all from the terminal:

```
/FILE UNIT=(PRINTER,NAME=SYSPRINT)
/LOAD FORTX
/JOB NOGO
/PARM SOURCE,XREF,DECK
      FORTRAN source statements
```

Note that for batch jobs SYSPRINT always defaults to the high speed printer.

Memory Requirements for Compilation

In the FORTRAN H Extended procedure shown on page 45 the NSEGS= parameter on the /LOAD statement specifies a value of 4 segments (a segment is 64K bytes). Unless the NSEGS= parameter is overridden, VPS will allot 256K of memory to a FORTRAN H Extended job execution. This allows for a compilation of approximately 800 FORTRAN source statements. For larger compilations, the programmer should increase the NSEGS= parameter on the /LOAD statement. In the following example a 1000 statement FORTRAN program is compiled using 7 segments:

```
/LOAD FORTX,NSEGS=7
      FORTRAN source statements
```

Since the NSEGS= parameter does not appear on the FORTRAN G1 procedure the VPS default of 2 segments (128K) will be used. This should be enough memory to compile a 600 statement program. If more memory is needed, the NSEGS= parameter should be specified on the /LOAD statement in the manner shown above.

FORTRAN Object Files

After object code is written to the temporary *2 file (by specifying DECK on the /PARM statement), it may be saved in a permanent VPS library file using either the /SAVE or /RSAV commands (see the VPS Handbook). Object code saved in this manner can be included in the input stream for execution either with FORTRAN source modules or by itself to be loaded and executed in the VPS FORTRAN "OS/VS" environment. (Note that object code from the two compilers can be intermixed in a job.) If object code is to be used in a FORTRAN execution with FORTRAN source modules it must be placed after the last FORTRAN source statement and before the /DATA statement. If no FORTRAN source statements are present, the object code must be placed after the /LOAD statement (or /JOB statement, if present) and before the /DATA statement.

In the following example a FORTRAN main program calls a subroutine CALC whose object exists in the file CALCOBJ. The VPS /INCLUDE control statement (see the VPS Handbook) is used to include the object after the main program in the input stream. The object code will be used by the loader to resolve any references to CALC in the main program.

```
/LOAD FORTG1
      FORTRAN source statements
/INC CALCOBJ
/DATA
      data
```

In cases where a FORTRAN program needs little modification, running from object code can save enormous amounts of time since the compilation is bypassed. In these cases, any /FILE statements and a /LOAD statement should be placed as the first line(s) of the object file using the VPS editor. If subroutine object, not already existing in the file, is needed; a /INCLUDE statement may be placed after the last object record to include other object in the input stream. Finally, if data exists in another library file which is to be read using standard unit 5, a /DATA statement should be inserted as the last record of the file.

In the following example, a main program whose object is in the library file MASTER is edited to include other object from files SUBAO, SUBBO, and SUBCO. The edited version of MASTER is shown below:

```
/LOAD FORTX
   object
/INC SUBAO,SUBBO,SUBCO
/DATA
```

If MASTER were to be run using the file DATA99 as the input data file, the following /EXEC command would be typed at *GO:

```
*GO
/exec master,data99
```

Efficiency in running seldom modified FORTRAN programs can be increased further by using a load module for execution instead of object code. Appendix B contains information on creating load modules from object and executing these load modules in the VPS "OS/VS" environment.

FORTRAN Input/Output Considerations

VPS supports full function input/output for FORTRAN program execution as described in the IBM FORTRAN Language Manual and Programmer's Guides. (Note, however, that though VPS supports asynchronous I/O as described for FORTRAN H Extended, using it will not make program execution more efficient since all VPS I/O is synchronous with execution.) FORTRAN supports two types of files for input/output - sequential and direct. The following sections describe these files and their use under VPS.

Sequential Files

Sequential files are files which must be read or written in sequential order. If record 126 is needed from a sequential file, records 1 through 125 must be read first. Standard units 5, 6, 7, and 9 are defined as sequential files. On VPS sequential files may exist as library files, disk work files, tape files, the input stream, etc. However, FORTRAN unformatted input/output is not supported on VPS library files. The example on page 47 shows a sequential tape file being used as input to a FORTRAN program. The IBM Programmer's Guides give extensive information on the definition and use of sequential files.

Direct Files

Direct files are files in which records may be accessed individually and in any order. That is, record 100 may be accessed without accessing the first 99. Records in a direct file can be processed using only the FORTRAN direct-access I/O statements but may be processed in a sequential fashion through the use of the associated variable in the direct-access statements. Direct files on VPS must be disk work files and must be created as fixed-unblocked files (i.e. RECFM=F must appear on the /FILE statement). DSORG=DA must also appear on the /FILE statement when the file is created. (Note that OPT=FORMAT as documented in the VPS Handbook should not be specified when creating a direct file with FORTRAN G1 or H Extended since these compilers will automatically format a NEW direct file.)

In the following example a direct file is created from a sequential tape file. Each record corresponds to each day of the year and contains telephone company data on the use of telephone information in the greater Boston area for that day. The first field contains the total number of information calls followed by 50 fields containing the number of information calls to each of the 50 states. Note that since the first field has a format specification of I10 and the 50 other fields have format specifications of I8 the block size for the direct file is 410 bytes (10 + 50*8).

```
/file unit=(tape,name=ftl1f001),vol=ser=mabell,
/ label=(1,nl),recfm=fb,lrecl=410,blksize=8200,disp=old
/file unit=(disk,name=ftl2f001),dsname=ur.mabl12.mabell.data,
/ disp=(new,keep),recfm=f,lrecl=410,blksize=410,dsorg=da,
/ space=(blk,365)
/load fortgl
      define file 12(365,410,e,nxrec)
      dimension istate(50)
      do 100 i=1,365
      read(11,10) itot,istate
      write(12'i,10) itot,istate
100   continue
10    format(i10,50i8)
      stop
      end
```

In the next example a control record is read from the input stream

indicating a date range for which statistics are to be generated from the direct file created above. Only a small portion of the program is shown - finding the average total calls for the period.

```
/file unit=(disk,name=ft12f001),dsname=ur.mab112.mabell.data,
/ disp=shr
/load fortg1
      define file 12(365,410,e,nxrec)
      dimension istate(50)
      .
      .
      .
      read(5,10) ibdate,iedate
10    format(2i3)
      itcalls=0
      do 30 i=ibdate,iedate
      read(12'i,20) itot,istate
20    format(i10,50i8)
      itcalls=itcalls+itot
30    continue
      iacalls=itcalls/(iedate-ibdate+1)
      write(6,40) ibdate,iedate,iacalls
40    format('period: ',i3,' - ',i3,' average calls: ',i10)
      .
      .
      .
/data
004015
```

For further information on direct files see the IBM FORTRAN Language Manual and the respective Programmer's Guide.

Computing Center Supplied Subroutines

As discussed on page 49 the Computing Center provides a number of subroutines to supplement those available from IBM. The following describes the more useful of these:

1) To obtain the current time of day -

```
CALL TIME(ihrs,imins,isecs,ihunds)
```

Where "ihrs" is an integer variable which will be set to the current hour, "imins" is an integer variable which will be set to the current minute, "isecs" is an integer variable which will be set to the current second, and "ihunds" is an integer variable which will be set to the current hundredth of a second.

2) To obtain the current execution time of a job -

```
CALL EXTIME(i)
```

Where "i" is an integer variable which will be set to the current execution time in hundredths of a second units.

3) To obtain the current date -

```
CALL DATE(date)
```

Where "date" is a real*8 variable which will be set to the character date in the form - ddmmmyy - and may be printed using an A8 format.

4) To specify a VPS library file name to be read from a LIBIN unit -

```
CALL OPEN(index,file,unit,ret,level)
```

Where -

"index" is a real*8 variable containing the eight character index name. If omitted or blank, the default of USERLIB will be used.

"file" is a real*8 variable containing the eight character file name.

"unit" is an integer variable containing the FORTRAN unit number which will be used to read the file.

"level" is an integer variable containing the resolution level for any /INCLUDE statement which may appear in the file. If omitted the default for that file will be used. (See the VPS Handbook for a complete description of the resolution level.)

"ret" is an integer variable containing the sum of the return codes which the subroutine is to ignore. If "ret" has a value of 0 or if it is omitted and an error occurs, or if an error occurs which is not specified in "ret", a message will be printed indicating the error and the job will be terminated. Otherwise, the value of "ret" is replaced with the actual return code and control is returned to the calling program. The return codes are:

```
  0 - successful open
  2 - invalid unit number
  4 - invalid file name
  8 - file not found
 16 - index not found
 32 - access to file disallowed
 64 - unable to enqueue
128 - library volume not mounted
256 - permanent I/O error
512 - storage unavailable for processing
```

Note that to use this subroutine correctly, the programmer must supply a corresponding /FILE statement having the following format:

```
/FILE UNIT=(LIBIN,NAME=FTxxF001)
```

Where "xx" is the FORTRAN unit number to be used to read the library file.

5) To save or replace a VPS library file with the contents of a LIBOUT unit -

```
CALL CLOSE(index,file,unit,type,ret)
```

Where -

"index" is a real*8 variable containing the eight character index name. If omitted or blank, the default of USERLIB will be used.

"file" is a real*8 variable containing the eight character file name.

"unit" is an integer variable containing the FORTRAN unit number which was used when the information was written.

"type" is an integer variable whose high order byte is one of the following characters:

S - Save the information as a new VPS library file then clear and rewind the LIBOUT unit.

R - Replace an existing VPS library file with the information then clear and rewind the LIBOUT unit.

D - Clear and rewind the associated LIBOUT unit wihtout saving the contents.

"ret" is an integer variable containing the sum of the return codes which the subroutine is to ignore. If "ret" has a value of 0 or is omitted and an error occurs, or if an error occurs which is not specified in "ret", a message will be printed and the job terminated. Otherwise, the value of "ret" is replaced with the actual return code and control is returned to the calling program. The return codes are:

```
   0 - successful close
   2 - unit number invalid
   4 - invalid file name
   8 - null file
  16 - index not found
  32 - write access disallowed
  64 - name already in use
 128 - insufficient library space available
 256 - permanent I/O error
 512 - storage unavailable for processing
1024 - no space in index
```

Note that to use this subroutine correctly, the programmer must supply a corresponding /FILE statement having the following format:

```
/FILE UNIT=(LIBOUT,NAME=FTxxF001)
```

Where "xx" is the FORTRAN unit number used to write the information.

In the following FORTRAN program a library file (specified by the

programmer using a read from FORTRAN unit 9) is copied to a LIBOUT unit. The programmer is then prompted for a file name under which the information is to be saved and the type of save. If the file already exists (CLOSE return code 64), the programmer is prompted for another name.

```
/file unit=(libin,name=ft24f001)
/file unit=(libout,name=ft25f001)
/load fortg1
      dimension array(20)
      real*8 file
      real*4 ret,type
      write(6,10)
10    format(' enter file name to be copied ')
      read(9,20) file
20    format(a8)
      call open(0,file,24,0,0)
30    continue
      read(24,40,end=50) array
40    format(20a4)
      write(25,40) array
      go to 30
50    continue
      write(6,60)
60    format(' enter new file name ')
      read(9,20) file
      write(6,70)
70    format(' enter s for save or r for replace ')
      read(9,80) type
80    format(a1)
      ret=64
      call close(0,file,25,type,ret)
      if (ret.eq.0)go to 100
      write(6,90)
90    format(' name already in use ')
      go to 50
100   stop
      end
```

Debugging

The most useful of the FORTRAN debugging tools is the debug facility of FORTRAN G1. The debug facility provides for tracing the flow within a program, tracing the flow between programs, displaying the values of variables and arrays, and checking the validity of subscripts. This is accomplished using debug "packets" which are composed of special debug facility statements and FORTRAN statements. Debug packets are placed after the source statements and before the END statement in the FORTRAN source module to be debugged. Appendix E of the IBM Language Manual gives a complete description of this facility.

The WATFIV compiler provides some other useful alternatives. Even if a program cannot be run under WATFIV, errors may be found by WATFIV during compilation that are not detected by either of the IBM compilers. For programs that can be run under WATFIV, the interactive

debugging and tracing facility is a very powerful tool. Descriptions of the job setup for a WATFIV compilation, execution, and use of the interactive debugging facility can be found in Chapter 10.

Experienced FORTRAN programmers who wish to debug from the assembler object code (the LIST compiler option produces the listing on SYSPRINT) should use the MAP option of the /JOB statement (see Appendix A) to find the locations of the FORTRAN main program and subroutines at execution time. By subtracting the nearest lower routine address from the last 6 digits of the PSW in the FORTRAN error message, the location of the failure in the object listing can be found.

For example, assume that the main program begins at 6718 (from the /JOB MAP statement) and the following error message is produced at execution time -

```
IH0209I IBCOM - PROG INT - DIVIDE CHECK OLD PSW IS 03F50009A20068A4
```

The failing instruction is at 18C hexadecimal (68A4 - 6718) in the object listing.

Chapter 6: PASCAL

The PASCAL compiler available under VPS is PASCAL 8000 for the IBM 360/370 series of computers developed at the Australian Atomic Energy Commission. This PASCAL compiler is a modified version of the PASCAL 8000 compiler written at the University of Tokyo for the HITAC 8800/8700 computer and, except for some extensions, processes the same language described in the "PASCAL User Manual and Report", by N. Wirth and K. Jensen (Springer-Verlag, 1975). A handout describing the full PASCAL 8000 language is available at the Computing Center.

PASCAL 8000 was written for the IBM OS/VS operating systems. Special VPS system support programs provide an "OS/VS" environment in which the PASCAL compiler and the user program run. PASCAL has not been modified to run under VPS.

Invoking PASCAL

The PASCAL compiler is invoked by placing the following control statement before the first PASCAL source statement of a program:

```
/LOAD PASCAL
```

Normally, this is done by placing the /LOAD statement first in a source file with the VPS editor when the source file is created.

In reality, not only does the /LOAD statement call the PASCAL compiler, but it also invokes a corresponding VPS internal procedure. A VPS internal procedure is a set of VPS control statements needed to run average source programs. These control statements may be overridden at execution time by programmer-supplied statements. The PASCAL procedure invoked using /LOAD PASCAL at the terminal is shown below:

```
/FILE UNIT=(DISK,NAME=LINKLIB),DSNAME=<pascal library>
/FILE UNIT=(DISK,NAME=SYSGO),SPACE=(TRK,(5,5)),RECFM=FB,LRECL=80,
/ BLKSIZE=4560
/FILE UNIT=(TERMOUT,NAME=SYSPRINT)
/FILE UNIT=(LIBIN,NAME=SYSIN),DSNAME=<input stream>
/LOAD PASCAL,NSEGS=4
```

Procedure Files

The default compilation and execution /FILE statements in the VPS PASCAL procedure are described below:

File	Use
LINKLIB	Defined as the work file containing the PASCAL compiler.

SYSGO	Defined as the object output file for compilation. See "Saving Compiler Object Code" later in this chapter for more information.
SYSPRINT	Defined as the terminal when executing at the terminal or as the high-speed printer when executing on the batch. Used by the PASCAL compiler and user programs as the output listing file.
SYSIN	Defined as the input stream - during compilation, from the first source statement to the /DATA statement; during execution, records after the /DATA statement (see below).

These files may be overridden or other files defined by placing appropriate /FILE statements before the /LOAD statement in the job stream.

Separating Program Statements from Data

As shown above, PASCAL 8000 and, possibly, the user program may read from the file SYSIN. The /DATA statement is used to separate source statements in the input stream from any data which is to be read by the program from SYSIN. A standard PASCAL job stream has the following structure:

```
/LOAD PASCAL

  PASCAL source statements

/DATA

  data
```

Files Used for Program Execution

External files are referred to by providing /FILE statements which use the PASCAL 'filename' in the NAME= subfield of the UNIT= parameter, except for the standard filenames INPUT and OUTPUT which use SYSIN and SYSPRINT. As noted above, /FILE statements for SYSIN and SYSPRINT are pre-defined for program execution in the VPS PASCAL procedure. However, the filenames INPUT and/or OUTPUT must still appear in the program heading if they are used.

Filenames may be declared, like any other variables in the program, and they may be passed as reference (VAR) parameters to procedures. External files may be used as "output" type files after a REWRITE operation and as "input" after a reset operation. The "extended" operations READ, READLN, WRITE, and WRITELN are allowed on all user-defined files. For each user-defined file declared in a PASCAL program a corresponding /FILE statement must also be placed before the /LOAD PASCAL statement in the job stream. In the following example a disk work file is used first as an "output" file and then as an "input" file:

```
/file unit=(disk,name=tempf),recfm=fb,lrecl=80,blksize=4560
/load pascal
program files(output,tempf);
  var i:char;
      tempf:file of char;
   begin
      i:='z';
      rewrite(tempf);
      write(tempf,i);
      writeln(tempf);
      reset(tempf);
      readln(tempf,i);
      write(output,' the value of i is:',i);
   .
   .
   .
```

I/O Notes

1. READ(...); is equivalent to READ(INPUT,...); and WRITE(...); is equivalent to WRITE(OUTPUT,...);.

2. For output files, line buffers are flushed only after the buffer is full or a WRITELN operation. This implies that if a program terminates without doing a WRITELN on output files, the last output line generated by the program may not actually be written.

3. Before a user-defined file can be used for writing, it must be preceded by a REWRITE operation. Before a user-defined file can be used for reading, it must be preceded by a RESET operation.

4. Assignment to, or comparison of, file variables is not allowed in the language. However, these operations may be performed on file buffers.

5. PASCAL 8000 local files are not supported.

Memory Requirements

In the PASCAL procedure shown on page 59 the NSEGS= parameter on the /LOAD statement specifies a value of 4 segments (a segment is 64K bytes). Unless the NSEGS= parameter is overridden, VPS will allot 256K of memory to a PASCAL execution. This allows for a compilation of approximately 3000 statements. For larger compilations, the programmer should increase the NSEGS= parameter on the /LOAD statement.

Compiler Options

Compiler options are specified within a PASCAL comment statement. The comment statement must appear as the first source statement of a program. The first character after the comment declaration " (* " must be a dollar sign "$", indcating that compiler options follow. The format of this special form of the comment statement appears below:

```
(*$option,option,... comments *)
```

Each option consists of a one-character option code followed by either a "+", indicating the option is to be turned on, or a "-", indicating the option is to be turned off. The following is a list of the options accepted by PASCAL and their description. Note that default options are under-lined.

Option	Description
L+ L-	Controls the printing of the PASCAL source program on SYSPRINT.
T+ T-	Controls the generation of code to provide run time checking for: assignment of values to variables of type <subrange>; ensuring that array index operations are within the bounds of the array specified; and ensuring that case statement selection falls within the realm of one of the case tags.
P+ p-	Controls the generation of the code necessary to generate a traceback and full dump of local variables if an execution error occurs.
N+ N-	Controls the generation of the code necessary to generate a traceback without a full dump if an execution error occurs.
S+ S-	Controls the flagging of constructs which are not 'Standard PASCAL'.

The following example shows a portion of a VPS terminal session where a PASCAL program is created using the VPS editor and then executed. The program reads in 3 numbers and finds their sum.

```
*GO
/edit
INPUT
/load pascal
 program sum (input,output);
  var i,j,total: integer;
   begin
    total := 0;
    for i := 1 to 3 do
      begin
       readln(j);
       total := total + j
      end;
    writeln(' the total is: ',total:8)
 end.
/data
1
4
3

EDIT
save paststst
*SAVED
*GO
/exec paststst

  PASCAL 8000/1.2     AAEC (1ST FEB 78)

  0630 --    PROGRAM SUM (INPUT,OUTPUT);
  0644 --     VAR I,J,TOTAL: INTEGER;
  0000 0-      BEGIN
  0012 --       TOTAL := 0;
  0018 --       FOR I := 1 TO 3 DO
  0020 1-         BEGIN
  002C --          READLN(J);
  0040 --          TOTAL := TOTAL + J
  0040 -1         END;
  005C --       WRITELN(' THE TOTAL IS: ',TOTAL:8)
  0086 -0    END.

*AAEC PASCAL COMPILATION CONCLUDED *

*NO ERRORS DETECTED IN PASCAL PROGRAM *

LOADER

PROGRAM SIZE - 3518 (HEX), 13,592 (DECIMAL).


THE TOTAL IS:        8

*GO
```

The four hexadecimal digits on the left in the listing indicate the relative addresses of variables, data, and code, wherever appropriate. While variables are being declared with the VAR construct, the hex address will reflect the relative offsets from the start of the stack for the procedure being compiled.

The next two indicators are known as nest level indicators, and reflect the static block structure of a procedure. The left indicator is incremented whenever a BEGIN, LOOP, REPEAT, or CASE is encountered. On termination of these structures, with an END or UNTIL, the right indicator is printed, and the static level counter decremented. This scheme makes it very convenient to match BEGIN - END pairs. A correctly composed procedure should start with a zero left indicator and terminate with a zero right indicator.

Where appropriate a character will follow the nest indicators reflecting the static procedure levels. The character will be updated for each nest level ("A" for level 2, "B" for level 3, etc.) and printed next to the heading and the BEGIN and END associated with that procedure.

Listing Control

There are several options available to the programmer to control the output listing format. These are indicated to the compiler by a '$' character in column 1 of a source input record, immediately followed by the option keyword. These control statements may appear between any two PASCAL source statements. The options available are:

Option	Description
$TITLE <title>	Replaces the title currently printed (if at all) with <title> and then skips to a new page. The title is printed at the top of each page, until a new $TITLE is encountered, or an $UNTITLE is found. <title> is limited to 40 characters.
$EJECT	Causes the next line of the listing to appear on a new page (unless that line is $UNTITLE).
$SPACE n	Causes n lines to be skipped in the listing.
$UNTITLE	Compiler-generated page-skipping and titling is suppressed.

Saving Compiler Object Code

The PASCAL 8000 compiler produces during compilation a saveable object module which can be used in later executions. This facility can be used to save the output of the PASCAL compiler for later program executions and avoid the overhead of recompilation each time the program is run. To save the object module produced by the compiler the following job setup should be used:

```
/FILE UNIT=(LIBOUT,NAME=SYSGO),DSNAME=*2
/LOAD PASCAL
/JOB NOGO

      PASCAL source program
```

The execution of this job will cause the compiler to compile the PASCAL program and the object module to be placed in the temporary *2 file. (Note that the object can be saved in a permanent library file using the /RSAV command - see The VPS Handbook.) The program itself will not be executed.

Saved object can be executed either by itself or with a PASCAL source program that references it. In the following example a PASCAL procedure has been compiled and its object saved in the VPS library file PROCDA. The example shows the use of that object with a PASCAL source program.

```
/LOAD PASCAL

      PASCAL source statements

/INC PROCDA
/DATA

      data
```

Chapter 7: PL/C

PL/C, which was developed at Cornell University, is a subset of the PL/I-F language. Information concerning PL/C is provided in the manual "The User's Guide to PL/C, The Cornell Compiler for PL/I" published by the Computer Science Department of Cornell University. The PL/C compiler is designed to be used by students in an introductory programming course. (The full PL/I language is provided by the PL/I Optimizing Compiler, documented in Chapter 8) PL/C features rapid compilation speeds and excellent source and object program diagnostics. One of the unique facilities of PL/C is that it attempts to "fix up" source program errors so that a source program containing errors will execute, often producing correct results. This feature permits the user to debug a PL/C program with relatively few runs on the computer.

PL/C Language Facilities

In general, PL/C is designed to be a compatible subset of PL/I-F. Thus PL/C programs, with minor exceptions, may be run under the IBM PL/I Optimizing Compiler, PLIOPT, with only control card changes. The following is a description of those PL/I-F features that are not supported by PL/C.

1. Files - PL/C supports STREAM and RECORD I/O sequential files with the limitation that list processing is not implemented. PL/C does not support direct-access files. Thus the following statements are not supported.

```
DELETE
LOCATE
REWRITE
UNLOCK
```

The following attributes are not supported.

```
BACKWARDS
BUFFERED
EXCLUSIVE
IRREDUCIBLE
KEYED
PACKED
REDUCIBLE
UNBUFFERED
UPDATE
```

The following options of the OPEN statement are not supported.

```
TRANSIENT
BUFFERED
UNBUFFERED
UPDATE
KEYED
```

EXCLUSIVE
BACKWARDS

The KEY and PENDING conditions are not supported.

2. Storage - PL/C does not support controlled or based storage. List processing features are not supported. Thus the ALLOCATE and FREE statements, the AREA, BASED, CONTROLLED, OFFSET and POINTER attributes, and the AREA condition are not supported.

3. PL/C does not support multi-tasking.

4. PL/C does not support the Optimizing Compiler compile time facilities.

5. PL/C does not support the 48 character set option.

6. PL/C does not support the DISPLAY statement.

7. Sterling and picture data-types are not supported.

8. The DEFINED and LIKE attributes are not supported.

Invoking PL/C Under VPS

PL/C may be run under the VPS batch or from VPS terminals. The PL/C compiler is invoked by use of the following control statement which must be placed before the first PL/C statement in a program:

```
/LOAD PLC
```

Using the default VPS memory allocation (2 segments, or 128K) a program of approximately 200 statements may be executed. For larger programs, the NSEGS= parameter of the /LOAD statement should be used (see the VPS Handbook).

PL/C Control Statements

PL/C recognizes the following control statements:

```
*PL/C     <options>
```

This statement is optional and is used to specify PL/C compiler options. It immediately follows the /LOAD PLC statement.

```
*PROCESS  <options>
```

This statement is required between external procedures and may also be used to redefine compiler options.

```
*OPTIONS  <options>
```

This statement is optional and is used to redefine compiler options. It may appear between any 2 PL/C source statements.

```
*DATA
```

This statement is used to separate source statements from program data in the input stream. It is not required if data is not present.

As with the /LOAD PLC statement, each control statement must begin in column 1. The statement sequence for most PL/C jobs is as follows:

```
/LOAD PLC
*PL/C option,option,...

 (PL/C source statements)

*DATA

 (Data, if any)
```

Note that earlier versions of the PL/C compiler recognized *PLC rather than *PL/C and /DATA instead of *DATA. These control statements are no longer valid.

PL/C Compiler Options

The following options may be specified on either the *PL/C, *PROCESS, or *OPTIONS control statements. This is only a partial list of the compiler options available. Full documentation on all the PL/C options may be found in "The User's Guide to PL/C". Where applicable, an underline indicates the default option on VPS.

Option	Description
SOURCE NOSOURCE	Controls the printing of a source listing. The default on the terminal is NOSOURCE. The default on the batch is SOURCE. SOURCE may be abbreviated as S.
ATR NOATR	Controls the printing of an attribute listing. ATR may be abbreviated as A.
OPLIST NOOPLIST	Controls the printing of the options in effect. OPLIST may be abbreviated as O.
XREF NOXREF	Controls the printing of a cross-reference listing. XREF may be abbreviated as X.
DUMP NODUMP	Controls the printing of a post-mortem status dump. DUMP may be abbreviated as D. input
DUMPE NODUMPE	Controls the printing of a post-mortem status dump only if an error was encountered during execution. DUMPE may be abbreviated as DE.
DUMPT NODUMPT	Controls the printing of a post-mortem status dump only if execution was terminated by an error. DUMPT may be abbreviated DT.

PL/C File Names Under VPS

Under the VPS PL/C implementation, standard names are provided to reference files. These file names may be specified in PL/C programs by using the DECLARE statement.

The following is a list of the default file names for PL/C programs under VPS. For each file name, the VPS default file type and logical record length are presented. For a detailed description of file types see the VPS Handbook.

File Name	VPS File Type	LRECL
SYSIN	input stream	80
SYSPRINT	terminal output (on terminal) line printer (on batch)	121
CONSOLE	terminal input (on terminal) input stream (on batch)	80
SYSPUNCH	*2 file (on terminal) card punch (on batch)	80

Input/Output Notes

1. If the user wishes to use a file other than those specified above, the file must be declared in the PL/C program using the DECLARE statement and defined for VPS using a /FILE control statement. /FILE statements must be placed before the /LOAD PLC statement in the source file. A detailed description of the /FILE control statement can be found in the VPS Handbook.

In the following example a temporary file is used to store intermediate information by a PL/C program. Note that the file name specified in the DECLARE statement must also appear in the NAME= subparameter of the UNIT= parameter of the /FILE statement defining the file.

```
/file unit=(disk,name=temp),recfm=f,lrecl=100,blksize=100
/load plc
 .
 .
 .
declare temp file;
declare rec char(100);
 .
 .
 .
write file (temp) from (rec);
 .
 .
 .
```

2. If a file is used with PL/C, a CLOSE statement will rewind the file. A CLOSED file may be re-processed starting with the first record if a subsequent OPEN statement is issued. If a file is opened for output and then CLOSED, an end of file mark will be written before the file is rewound.

3. An end of file condition may be generated for input read from the terminal (file CONSOLE) by entering the VPS command /END.

4. Any prompting messages to be printed before reading data interactively from CONSOLE must be forced to print by use of a PUT SKIP statement with no data. For example, the following sequence will cause the message ENTER DATA to be printed prior to reading the terminal keyboard.

```
put skip list('enter data'); put skip;
get file(console) list (a,b,c);
```

In the above example, the second PUT SKIP statement is required to force the message ENTER DATA to print before the GET FILE statement is executed.

Sample PL/C Run

The following example shows the execution of a PL/C program from a VPS terminal. The program reads two values interactively from the terminal keyboard, adds them together, and prints the sum. The program has a "bug" which PL/C fixes so that the program runs correctly.

```
*GO
/edit
INPUT
/load plc
 adder: proc options(main);
 /* program to add two numbers */
  on endfile (console) stop;
 loop: put skip list('enter a and b');put skip;
       get file(console) list (a,b);
       c=a+b
       put list ('the sum is ',c);
       go to loop;
       end;

EDIT
save plctest
*SAVED
*GO
/exec plctest

7    1           1                C=A+B
 ERROR IN STMT     7  MISSING SEMI-COLON (SY08)
      FOR STMT     7 PL/C USERS  C=A+B;


ERRORS/WARNINGS DETECTED DURING CODE GENERATION:

   WARNING: NO FILE SPECIFIED. SYSIN/SYSPRINT ASSUMED. (CG0C)


ENTER A AND B
1 3
THE SUM IS      4.00000E+00
ENTER A AND B
5.67 8.90
THE SUM IS      1.45700E+01
ENTER A AND B
/end


IN STMT     3 PROGRAM IS STOPPED.
*GO
```

Chapter 8: PL/I

Two PL/I compilers are supported under VPS. PL/C, a student oriented compiler, provides a subset of the PL/I language and is documented in Chapter 7. This chapter documents the use of the IBM PL/I Optimizing Compiler under VPS.

The PL/I Optimizer provides full PL/I language support and has not been modified to run under VPS - rather, special VPS system support programs provide an "OS/VS" environment in which this compiler and its compiled programs run. VPS PL/I programs are, therefore, source and object compatible with OS/VS PL/I programs. Programs can be written and tested under VPS and run under OS/VS with only control statement changes.

IBM Manuals

This chapter is not a replacement for the documentation available in IBM publications. It is a supplement to the documentation explaining necessary VPS control statement usage for this compiler. PL/I programmers are encouraged to use the following publications which can be found in reference racks at the Computing Center.

The PL/I language supported by this compiler is documented in -

"IBM OS PL/I Checkout and Optimizing Compilers: Language Reference Manual" (GC33-0009)

Information on compiler options, defining data sets, etc. can be found in -

"IBM OS PL/I Optimizing Compiler: Programmer's Guide" (GC33-0008)

Error messages can be found in -

"IBM OS PL/I Optimizing Compiler: Messages (GC33-0027)

Invoking the PL/I Optimizing Compiler

The compiler is invoked by placing a VPS /LOAD control statement (see the VPS Handbook for a full description of /LOAD) before the first PL/I source statement in a job stream. Normally, this is done by placing the /LOAD statement first in a source file with the VPS editor when the source file is created. The format of the /LOAD control statement used to invoke the PL/I Optimizer is:

```
/LOAD PLIOPT
```

In reality, not only does the /LOAD statement load the PL/I compiler, but it also invokes a corresponding VPS internal procedure. A VPS internal procedure is a set of VPS control statements needed to run average source programs. These control statements may be overridden at execution time by programmer-supplied statements. The

PL/I procedure invoked using /LOAD PLIOPT at the terminal is shown below:

```
  /FILE UNIT=(DISK,NAME=LINKLIB),DSNAME=<pl/i library>,DISP=SHR
  /FILE UNIT=(DUMMY,NAME=SYSPRINT)
  /FILE UNIT=(LIBIN,NAME=SYSIN),DSNAME=<input stream>
  /FILE UNIT=(DISK,NAME=SYSLIN),SPACE=(REC,(1000,1000)),RECFM=FB,
  / LRECL=80,BLKSIZE=4560
  /FILE UNIT=(DISK,NAME=SYSUT1),SPACE=(BLK,(10,10)),RECFM=F,
  / BLKSIZE=1024
  /FILE UNIT=(LIBOUT,NAME=SYSPUNCH),DSNAME=*2
  /FILE UNIT=(TERMOUT,NAME=SYSTERM)
-----
  /FILE UNIT=(DISK,NAME=LINKLIB),DSNAME=<pl/i library>
  /FILE UNIT=(DISK,NAME=SORTLIB),DSNAME=<SORT utility lib>,DISP=SHR
  /FILE UNIT=(TERMOUT,NAME=SYSPRINT)
  /FILE UNIT=(LIBIN,NAME=SYSIN),DSNAME=<input stream>
  /FILE UNIT=(LIBOUT,NAME=SYSPUNCH),DSNAME=*2
  /FILE UNIT=(TERMIN,NAME=CONSOLE)
  /LOAD PLIOPT
```

The /FILE statements appearing above the dashed line define files used by the compiler. Those appearing below the dashed line define standard files used by an executing PL/I program.

Compiler Files

The file requirements for the PL/I Optimizing Compiler are listed below:

File	Use
LINKLIB	PL/I library - defined as the work file containing the PL/I Optimizing Compiler.
SYSPRINT	Output listing file - defined as DUMMY if compiling at the terminal (unless a list-producing option is specified on the /PARM statement - see below) or as the high speed printer if compiling on the batch.
SYSIN	Input source and/or object file - defined as the input stream from the first source statement to the /DATA statement (see below).
SYSLIN	Output object file - defined as a temporary work file used to hold the object code to be loaded by the loader for execution.
SYSUT1	Temporary work file - used by the compiler during compilation.

SYSPUNCH	Output object file - defined as the *2 file if compiling at the terminal or the card punch if compiling on the batch allowing object code to be saved for future use (used only if DECK appears on the /PARM statement).

Execution Time Files

As shown above the VPS PLIOPT procedure provides default /FILE statements for the standard PL/I data sets - SYSIN, SYSPRINT, SYSPUNCH, and CONSOLE. The input stream is defined as the file SYSIN at both compilation and execution time. If the default /JOB statement is used (see below), after the Optimizer is finished compiling the source program, the VPS loader loads the object code from SYSLIN into memory and the program is executed. SYSIN is then defined as any records following a /DATA control statement. The /DATA statement is used to separate source statements in the input stream (which are read by the compiler using SYSIN) from data to be read at program execution time using SYSIN. A standard PL/I compilation and execution has the following structure:

```
/LOAD PLIOPT

   PL/I source statements

/DATA

   data
```

The default execution time /FILE statements for the PL/I Optimizer are shown below:

File	Use
LINKLIB	PL/I library - defined as the work file containing the PL/I Transient Library routines.
SORTLIB	SORT/MERGE library - defined as the work file containing the SORT/MERGE routines used only if an internal sort is requested.
SYSPRINT	Output file - defined as the terminal if executing at the terminal or the high speed printer if executing on the batch.
SYSIN	Input file - defined as the input stream after the /DATA control statement.
SYSPUNCH	Output file - defined as the *2 file if executing at the terminal or as the card punch if executing on the batch.

CONSOLE	Input file - defined as the terminal if executing at the terminal or as the input stream after the /DATA statement if executing on the batch.

These files may be overridden or other files defined by placing appropriate /FILE statements before the /LOAD statement in the job stream.

Specifying Compiler Options

The VPS /PARM control statement may be used to pass compiler options to the PL/I Optimizing Compiler. The /PARM statement must be placed in the job stream after the /LOAD statement and before the first PL/I source statement (note that /PARM and /JOB may appear in either order). The format of the /PARM statement is shown below:

```
/PARM option,option,...
```

Where "option" is a valid compiler option. If more than one option is specified, they must be separated by commas. Multiple /PARM statements are not supported.

The following is a subset of the compiler options accepted by the Optimizing Compiler. A full description of all the options can be found in the IBM Programmer's Guide. Note that the default options are underlined.

Option	Description
SOURCE NOSOURCE	Controls the printing of a PL/I source listing on SYSPRINT. NOSOURCE is the default on the terminal. SOURCE is the default on the batch. (Abbr. - S|NS)
DECK NODECK	Controls the writing of the object code to SYSPUNCH. (Abbr. - D|ND)
MAP NOMAP	Controls the printing of tables showing the organization of static storage. (Abbr. - M|NM)
ATTR NOATTR	Controls the printing of the listing of source program identifiers and their attributes. (Abbr. - A|NA)
COMPILE NOCOMPILE	Controls whether the source program is to be compiled (COMPILE) or just syntax checked (NOCOMPILE). (Abbr. - C|NC)
XREF NOXREF	Controls the printing of the identifier cross reference. (Abbr. - X|NX)
LIST NOLIST	Controls the printing of the assembler language listing of the compiled program on SYSPRINT.

MARGINS(m,n,c) MARGINS(2,100,1)	Specifies the locations in each input record that contain PL/I source statements. Where 'm' is the column number of the leftmost character to be processed, 'n' is the column number of the rightmost character to be processed, and 'c' (which is outside the range 'm' to 'n') is the location of the printer control character for the source listing. (Abbr. - MAR(m,n,c))
OBJECT NOOBJECT	Controls the writing of the object code to SYSLIN for use by the loader. (Abbr. - OBJ|NOBJ)
OPT(TIME) OPT(0) OPT(2) NOPT	Controls the optimization level to be in force during compilation. NOPT specifies fast compilation speed, but inhibits optimization for faster execution and reduced main storage requirements (OPT(0) is equivalent to NOPT). OPT(TIME) specifies that the compiler is to optimize the machine instructions for faster execution and a reduction in main storage usage (OPT(2) is equivalent to OPT(TIME)).
TERM TERM(op,op,...) NOTERM	Applicable only when compiling at the terminal and can specify compiler listing options which are to be printed on the terminal independent of the SYSPRINT data set. If TERM is specified with no options, dianostic and informational messages are printed at the terminal. The following options can be specified in the option list: AGGREGATE, ATTRIBUTES, ESD, INSOURCE, LIST, MAP, OPTIONS, SOURCE, STORAGE, and XREF.

Specifying Loader Options

The VPS /JOB control statement may be used to specify loader options for compiled programs which are to be loaded for execution. (A full description of the /JOB statement can be found in Appendix A.) The /JOB statement can be used to control the printing of a load map, unresolved external references, entry address, etc. It can be used to terminate a job after compilation but before execution (NOGO). It can also be used to override the default list of libraries to be searched at load time for unresolved external references (such as mathematical and graphics subroutines existing in VPS subroutine libraries). The default PL/I library search list is -

SYS.PLIXLIB - the IBM supplied support and mathematical subroutine library

SYS.TEKLIB - the TEKTRONIX graphics subroutine package

SYS.VPSLIB - the Computing Center subroutine library

The libraries are searched in the order specified.

The /JOB statement must be placed in the job stream after the /LOAD statement but before the first PL/I source statement. In the following example the /JOB statement is used to terminate a job immediately after compiling a PL/I program:

```
/LOAD PLIOPT
/JOB NOGO
     PL/I source statements
```

Note that though continuation is not supported for the /JOB statement, multiple /JOB statements are permissible.

The Strange Behavior of SYSPRINT

Normally, when executing a job at the terminal, the SYSPRINT file is defined as TERMOUT. For PL/I compilations at terminals the SYSPRINT file is dummied out to bypass massive amounts of output being printed at the terminal during compilation. However, there are circumstances where the programmer may wish to direct SYSPRINT to a location other than the terminal (a library file, the high speed printer, a work file, etc.). To allow this, VPS will not dummy SYSPRINT if a list-producing option appears on a /PARM statement in the job stream. A list-producing option is defined as any compiler option which will cause printing to occur on SYSPRINT. In the following example a program is compiled where the source and cross-reference listing is to be printed on the high speed printer, the object code is to be written to the *2 file, and job execution is to be terminated after compilation - all from the terminal:

```
/FILE UNIT=(PRINTER,NAME=SYSPRINT)
/LOAD PLIOPT
/JOB NOGO
/PARM SOURCE,XREF,DECK
     PL/I source statements
```

Note that if the default SYSPRINT file is used, a list-producing option is specified, and the TERM parameter with option(s) is specified, double messages will appear on the terminal creating a very confusing output listing.

Memory Requirements for Compilation

The PL/I Optimizing Compiler requires 60K bytes of storage for compilation and therefore, runs quite nicely in the default VPS region of 2 segments (128K). This is more than enough room for small and medium size PL/I programs. PL/I will automatically use the spill file (SYSUT1) for additional auxiliary storage if it is needed during compilation. In many cases compilation times can be reduced by increasing the main storage allocation for large compilations and decreasing the use of the spill file. To increase the main storage allocation for compilation the NSEGS= parameter of the /LOAD statement should be used (see the VPS Handbook for a complete description of the NSEGS= parameter). In the following example the number of segments used for compilation is increased to 4 (256K bytes):

```
/LOAD PLIOPT,NSEGS=4
      PL/I source statements
```

PL/I Object Files

After object code is written to the temporary *2 file (by specifying DECK on the /PARM statement), it may be saved in a permanent VPS library file using either the /SAVE or /RSAV commands (see the VPS Handbook). Object code saved in this manner can be included in the input stream for execution either with other PL/I source modules or by itself to be loaded and executed in the VPS PL/I "OS/VS" environment. If object code is to be used in a PL/I execution with PL/I source modules it must be placed after the last PL/I source statement and before the /DATA statement. If no PL/I source statements are present, the object code must be placed after the /LOAD statement (or /JOB or /PARM statement, if present) and before the /DATA statement.

In the following example a PL/I main program calls an external procedure DCALC whose object exists in the file DCALCO. The VPS /INCLUDE control statement (see the VPS Handbook) is used to include the object after the main program in the input stream. The object code will be used by the loader to resolve any references to DCALC in the main program.

```
/LOAD PLIOPT
      PL/I source statements
/INC DCALCO
/DATA
      data
```

In cases where a PL/I program needs little modification, running from object code can save enormous amounts of time since the compilation is bypassed (of course, if source is modified the program must be recompiled). In these cases, any /FILE statements and a /LOAD statement may be placed as the first line(s) of the object file using the VPS editor. If subroutine object, not already existing in the file, is needed; a /INCLUDE statement may be placed after the last object record to include other object in the input stream. Finally, if data exists in another library file which is to be read from SYSIN, a /DATA statement should be inserted as the last record of the file.

In the following example, a main program whose object is in the library file MMAST is edited to include other object from files MSUBAO, MSUBBO, and MSUBCO. The edited version of MMAST is shown below:

```
/LOAD PLIOPT
   object
/INC MSUBAO,MSUBBO,MSUBCO
/DATA
```

If MMAST were to be run using the file MDATA2 as the input data file, the following /EXEC command would be typed at *GO:

```
*GO
/exec mmast,mdata2
```

Efficiency in running seldom modified PL/I programs can be increased further by using a load module for execution instead of object code. Appendix B contains information on creating load modules from object and executing these load modules in the VPS "OS/VS" environment.

PL/I Optimizer Features Not Supported on VPS

The full PL/I language may be compiled using the Optimizer on VPS, however, certain restrictions apply to program execution. The following is a list of those restrictions:

1. All pre-processor options are supported including %INCLUDE. However, nested %INCLUDEs are not allowed. The VPS /INCLUDE statement may be used as a replacement for nested %INCLUDEs (see the VPS Handbook).

2. Multitasking is not supported.

3. File organizations CONSECUTIVE and all REGIONAL are supported on VPS with minor exceptions (see "PL/I Files" below). INDEXED, TELEPROCESSING, and VSAM are not supported.

4. The H option (hexadecimal dump) of the PL/I dump facility is not suppoted.

Note that use of the IBM Sort/Merge Package in PL/I as documented in the IBM PL/I Programmer's Guide is supported for execution under VPS.

PL/I Files

VPS supports the two major PL/I file types - CONSECUTIVE and REGIONAL. (Note that the file type CONSECUTIVE is known generally as sequential. However, since the IBM PL/I language group made the brilliant decision to use the keyword SEQUENTIAL as a type of access in the DECLARE FILE statement, everyone is confused. Therefore, for this discussion we will use the PL/I keyword CONSECUTIVE to refer to files which are known as "sequential" in all other chapters of this manual.) The following sections describe these files and their use on VPS.

If the user wishes to use a file other than those specified in "Execution Time Files" above, the file must be declared in the PL/I program using the DECLARE statement and defined for VPS using a /FILE control statement. /FILE statements must be placed before the /LOAD PLIOPT statement in the source file. A detailed description of the /FILE control statement can be found in the VPS Handbook.

In the following example a temporary file is used to store intermediate information by a PL/I program. Note that the file name specified in the DECLARE statement must also appear in the NAME= subparameter of the UNIT= parameter of the /FILE statement defining the file.

```
/file unit=(disk,name=temp),recfm=f,lrecl=100,blksize=100
/load pliopt
 .
 .
 .
declare temp file record;
declare rec char(100);
 .
 .
 .
write file (temp) from (rec);
 .
 .
 .
```

Consecutive Files

Consecutive files are files which must be read or written in sequential order. If record 239 is to be read from a consecutive file, records 1 through 238 must be read first. VPS supports stream-oriented and record-oriented transmission for consecutive files.

All consecutive file access types and access attributes are supported under VPS with the exception of the following:

1. The BACKWARDS attribute in not supported.

2. The UPDATE access attribute is not supported unless the UNBUFFERED attribute is specified (which requires the file to be unblocked and also a work file).

3. Spanned records are not supported unless the UNBUFFERED attribute is specified (which again requires the file to be unblocked and also a work file).

As described in the IBM Programmer's Guide DCB attributes (i.e. BLKSIZE, RECFM, LRECL, etc.) must be specified either in the ENVIRONMENT attribute of the DECLARE statement for the file or on the /FILE statement (the VPS equivalent of the IBM DD statement) defining the file. On VPS consecutive files may exist as library files, disk work files, tape files, terminal input, the input stream, etc. In the following example, a consecutive file is read from tape using the file name TAPEIN and from the input stream as SYSIN:

```
/file unit=(tape,name=tapein),vol=ser=mastdt,disp=old,
/ label=(1,nl),recfm=fb,lrecl=120,blksize=1200
/load pliopt
 tcalc: proc options(main);
  dcl tapein file record input,
      taperec char(120),
      eof bit(1) init('0'b);
  on endfile (sysin) eof='1'b;
  tot1=0;  tot2=0;  tot3=0;
  rloop: do while (¬eof);
    get edit (a,b,c) (col(1),3(f(5)));
    read file (tapein) into (taperec);
    get string (taperec) edit (aa,bb,cc) (3(f(5)));
    tot1=tot1+a+aa;
    tot2=tot2+b+bb;
    tot3=tot3+c+cc;
    end rloop;
  put edit (' the totals are:',tot1,tot2,tot3) (a,3(x(1),f(8,2)));
  stop;
 end tcalc;
/data
  66.  43.  87.
  49. 129. 205.
.
.
.
```

Consecutive File Notes

1. If a file is used with PL/I, a CLOSE statement will rewind the file. A CLOSED file may be re-processed starting with the first record if a subsequent OPEN statement is issued. If a file is opened for output and then CLOSED, an end of file mark will be written before the file is rewound.

2. An end of file condition may be generated for input read from the terminal (file CONSOLE) by entering the VPS command /END.

3. Any prompting messages to be printed before reading data interactively from CONSOLE must be forced to print by use of a PUT SKIP statement with no data. For example, the following sequence will cause the message ENTER DATA to be printed prior to reading the terminal keyboard.

```
put skip list('enter data'); put skip;
get file(console) list (a,b,c);
```

In the above example, the second PUT SKIP statement is required to force the message ENTER DATA to print before the GET FILE statement is executed.

Regional Files

Regional files are files in which records may be accessed individually and in any order. That is, record 416 may be accessed without accessing the first 415. The records within the data set can be organized in one of three ways: REGIONAL(1), where each region is a record and each record is identified by its region number (note that this organization is identical to FORTRAN direct access files); REGIONAL(2), which is similar to REGIONAL(1) except that a key is recorded with each record; and REGIONAL(3), which is similar to REGIONAL(2) except that each region corresponds to one track of the direct access device and is not a record position (depending on the record length, a region may contain one or more records). The IBM Language Reference Manual and the Programmer's Guide contain extensive information on the creation and use of regional files.

Regional files on VPS must be disk work files and DSORG=DA must be specified on the /FILE statement when the data set is created. (Note that OPT=FORMAT as documented in the VPS Handbook should not be specified when creating a regional file with the PL/I Optimizing Compiler since the compiler will automatically format a regional file opened with the access attribute OUTPUT.) All rules governing DCB attributes as described in the IBM Programmer's Guide are applicable when using regional files on VPS.

In the following example a REGIONAL(3) data set is created containing what might be the inventory of a library. The key of each record is 4 characters in length (KEYLEN=4) and assigned to the variable NUMBER. The region number is calculated from the key and assigned to the variable REGION.

```
/file unit=(disk,name=stock),dsn=ur.plg100.stock,disp=(new,keep),
/ recfm=v,blksize=118,dsorg=da,keylen=4,space=(trk,(30))
/load pliopt
 crr3: proc options(main);
  dcl stock file record keyed env(regional(3)),
      1 card,
        2 number char(4),
        2 author char(25) var,
        2 title char(50) var,
        2 qty1 fixed dec(3),
      1 book ctl,
        2 (13,14) fixed dec(3),
        2 qty2 fixed dec(3),
        2 descn char(x),
      (11,12,x) fixed dec(3),
      inter fixed dec(5), region char(8),
      eof bit(1) init('0'b);
  on endfile (sysin) eof='1'b;
  open file(stock) sequential output;
  next: do while(¬eof);
    get file(sysin) list(card);
    11=length(author);  12=length(title);
    x=11+12;
    allocate book;
    13=11;  14=12;
    qty2=qty1;  descn=author||title;
    inter=(number-1000)/225;
    region=inter;
    write file(stock) from(book) keyfrom(number||region);
    end next;
  close file(stock);
  stop;
 end crr3;
/data
'1015' 'w.shakespeare' 'julius caesar' '5'
'1322' 'a.a.milne' 'winnie the pooh' '3'
'3078' 't.s.white' 'the once and future king' '4'
'3089' 'j.h.porter' 'the vps handbook' '32'
.
.
.
```

In the next example the file is updated directly. If the CODE read from SYSIN is 'A' the entry in the file is deleted, if CODE is 'I' the number of COPIES is subtracted from the quantity and the record is rewritten, or if the CODE is 'R' the number of COPIES is added to the quantity and the record rewritten.

```
/file unit=(disk,name=stock),dsn=ur.plg100.stock,disp=old
/load pliopt
 crr4: proc options(main);
  dcl stock file record keyed env(regional(3)),
      inrec char(110) var,
      1 book based(p),
        2 vlen fixed bin(15),
        2 (13,14) fixed dec(3),
        2 qty2 fixed dec(3),
        2 descn char(75),
      1 card,
        2 number char(4),
        2 copies fixed dec(3),
        2 code char(1),
      region char(8), inter fixed dec(5),
      eof bit(1) init('0'b);
  on endfile(sysin) eof='1'b;
  open file(stock) direct update;
  next: do while(¬eof);
     get file(sysin) list(card);
     inter=(number-1000)/225;
     region=inter;
     if code='a' then delete file(stock) key(number||region);
     else do;
        read file(stock) into(inrec) key(number||region);
        p=addr(inrec);
        if code='r' then qty2=qty2+copies;
        else qty2=qty2-copies;
        put edit (number,substr(descn,1,13),substr(descn,13+1,14),
           qty2) (col(1),x(2),3(a,x(1)),f(4));
        rewrite file(stock) from(inrec) key(number||region);
        end;
     end next;
   close file(stock);
   stop;
 end crr4;
/data
'5673' '4' 'i'
'1214' '1' 'r'
'3087' '2' 'a'
.
.
.
```

Computing Center Supplied Subroutines

As discussed on page 77 the Computing Center provides a number of subroutines to supplement those available from IBM. The following describes the more useful of these:

1) To specify a VPS library file name to be read from a LIBIN unit -

```
CALL OPEN(index,file,'ddname',ret,level)
```

Where -

"index" is a character(8) variable containing the eight character index name. If omitted or blank, the default of USERLIB will be used.

"file" is a character(8) variable containing the eight character file name.

"ddname" is the name, in single quotes, as specified in the PL/I declare statement for the output file. (This corresponds to the NAME= subparameter on the /FILE statement.)

"ret" is an integer variable containing the sum of the return codes which the subroutine is to ignore. If "ret" has a value of 0 or if it is omitted and an error occurs, or if an error occurs which is not specified in "ret", a message will be printed indicating the error and the job will be terminated. Otherwise, the value of "ret" is replaced with the actual return code and control is returned to the calling program. The return codes are:

```
  0 - successful open
  2 - invalid unit number (unit type not LIBIN)
  4 - invalid file name
  8 - file not found
 16 - index not found
 32 - access to file disallowed
 64 - unable to enqueue
128 - library volume not mounted
256 - permanent I/O error
512 - storage unavailable for processing
```

Unmaskable return codes are:

```
1 - no corresponding /FILE statement found
3 - number of parameters in call in error
5 - argument list invalid
```

"level" is an integer variable containing the resolution level for any /INCLUDE statement which may appear in the file. If omitted the default for that file will be used. (See the VPS Handbook for a complete description of the resolution level.)

Note that to use this subroutine correctly, the programmer must supply a corresponding /FILE statement having the following format:

```
/FILE UNIT=(LIBIN,NAME=ddname)
```

Where ddname is the name assigned to the input file in the PL/I program.

2) To save or replace a VPS library file with the contents of a LIBOUT unit -

```
CALL CLOSE(index,file,'ddname','type',ret)
```

Where -

"index" is a character(8) variable containing the eight character index name. If omitted or blank, the default of USERLIB will be used.

"file" is a character(8) variable containing the eight character file name.

"ddname" is the name, in single quotes, as specified in the PL/I declare statement for the output file. (This parameter corresponds to the NAME= subparameter of the /FILE statement.)

"type" is an integer variable whose high order byte is one of the following characters:

S - Save the information as a new VPS library file then clear and rewind the LIBOUT unit.

R - Replace an existing VPS library file with the information then clear and rewind the LIBOUT unit.

D - Clear and rewind the associated LIBOUT unit wihtout saving the contents.

"ret" is an integer variable containing the sum of the return codes which the subroutine is to ignore. If "ret" has a value of 0 or is omitted and an error occurs, or if an error occurs which is not specified in "ret", a message will be printed and the job terminated. Otherwise, the value of "ret" is replaced with the actual return code and control is returned to the calling program. The return codes are:

```
   0 - successful close
   2 - invalid unit number (unit type not LIBOUT)
   4 - invalid file name
   8 - null file
  16 - index not found
  32 - write access disallowed
  64 - name already in use
 128 - insufficient library space available
 256 - permanent I/O error
 512 - storage unavailable for processing
1024 - no space in index
```

Unmaskable return codes are:

```
1 - no corresponding /FILE statement found
3 - number of parameters in call in error
5 - argument list invalid
7 - type parameter missing or invalid
```

Note that to use this subroutine correctly, the programmer must supply a corresponding /FILE statement having the following format:

```
/FILE UNIT=(LIBOUT,NAME=ddname)
```

Where ddname is the name assigned to the output file in the PL/I program.

Notes on Efficient Debugging

Though the PL/I Optimizing Compiler provides one of the most powerful and flexible of the higher level languages, it can be notoriously expensive to run. Numerous compilations while debugging a program will take their toll in computing dollars and programmer frustration. However, by employing the following procedures the PL/I programmer can liberate her/himself from becoming a slave to the terminal.

1. No matter how good a programmer you think you are, or how small your program is, if you think you are not going to have syntax errors in the first compilation of a new program, you probably also still believe in the Tooth Fairy. Always specify NOCOMPILE (NC) on the /PARM statement when compiling a new program until no syntax errors are detected. Without NC the Optimizer will try to compile your program even though it may find syntax errors that would result in a program that cannot execute.

2. For medium to large programs which have been syntax checked, the main memory allocation should be increased to 4 segments (/LOAD PLIOPT,NSEGS=4). This will cut down on the use of the spill file and decrease compilation time.

3. When debugging programs which contain external procedures, after the external procedures have been debugged, they should be run from object rather than recompiling the entire program each time a source change has been made to a single procedure.

4. When debugging a program containing arrays, enabling the SUBSCRIPTRANGE condition will detect errors in subscript values. Similarly, programs using the SUBSTR built-in function can be checked for valid string assignments by enabling the STRINGRANGE condition. Conditions can be enabled by specifying the condition in a condition prefix. A condition prefix is a list of one or more condition names, enclosed in parentheses and separated by commas, and connected to a statement (or a statement label) by a colon. The condition prefix can be connected to a PROCEDURE statement and the conditions will be enabled for the entire procedure. In the following example STRINGRANGE is enabled for the procedure SCAN:

```
(stringrange):
scan: proc options(main);
.
.
.
end scan;
```

5. Liberal use of PUT statements can help to find errors in programs using based variables. Printing out some part of a structure addressed by the based variable is in most cases the only way of detecting whether the based variable is correct. If the structure contains no printable fields, define one for the purposes of debugging - it can always be deleted after the program is working.

6. Due to the way PL/I buffers its output records, programs which abnormally terminate (a PSW and register dump appears as the last thing in your program listing or your program is cancelled - either by you or VPS) do not include the last line of output produced by your program. For debugging purposes, using a PUT SKIP; statement after each PUT statement will print the buffer when it is produced. The "put skips" should be removed after the program is debugged. Note that all output is printed when the program terminates normally or an on-condition is raised.

7. PL/I provides a column on the lefthand side of the source listing indicating the statement nesting level. This information should be checked thoroughly for improper nesting in programs with multiple nesting levels. It is also a good idea to label BEGIN and DO statements and place that label in the appropriate END statement.

Chapter 9: SNOBOL

SNOBOL is a string manipulation language developed at Bell Telephone Laboratories in the early 1960s. The compiler currently available on VPS is SNOBOL4 Version 3.5 and the language it accepts is fully documented in "The SNOBOL4 Programming Language" (Prentice-Hall, 1971). SNOBOL has not been modified to run under VPS - rather, special VPS system support programs provide an "OS/VS" environment in which this compiler runs. The following sections describe how to use SNOBOL on VPS.

Invoking SNOBOL

The SNOBOL compiler is invoked by placing the following VPS control statement before the first SNOBOL source statement of a program:

```
/INCLUDE SNOBOL4
```

Normally, this is done by placing the /INCLUDE statement first in a source file with the VPS editor when the source file is created. Alternatively, the file name SNOBOL4 may appear first in a list of files to be executed by the /EXEC statement, as in:

```
*GO
/EXEC SNOBOL4,source file name,...
```

The file SNOBOL contains the necessary VPS control statements to invoke the SNOBOL compiler and set up the default VPS environment for program execution. For execution at the terminal the following default /FILE statements are used:

```
/FILE UNIT=(DISK,NAME=LINKLIB),DSNAME=<snobol library>
/FILE UNIT=(TERMOUT,NAME=FT06F001)
/FILE UNIT=(LIBIN,NAME=FT05F001),DSNAME=<input stream>
/FILE UNIT=(LIBOUT,NAME=FT07F001),DSNAME=*2
/FILE UNIT=(TERMIN,NAME=FT09F001)
```

These /FILE statements may be overridden or new files defined at execution time by programmer-supplied statements.

Default /FILE Statements

All input/output for SNOBOL is handled by FORTRAN I/O routines, which account for the unique names assigned to the NAME= subparameter of the UNIT= parameter of the /FILE statement. The format of this parameter is defined as follows:

```
NAME=FTxxF001
```

Where 'xx' is a FORTRAN unit number. Therefore, FORTRAN unit 6 is FT06F001, unit 14 is FT14F001, etc. The /FILE statements shown above are used by SNOBOL variables as shown below or can be programmer-defined using functions discussed later in this chapter.

The default /FILE statements and their associated SNOBOL variables are described below:

File	Use
LINKLIB	SNOBOL library - defined as the work file containing the SNOBOL compiler.
FT05F001	SNOBOL variable INPUT - defined as the input stream after the SNOBOL END statement.
FT06F001	SNOBOL variable OUTPUT - defined as terminal output if executing at the terminal or the high speed printer if executing on the batch.
FT07F001	SNOBOL variable PUNCH - defined as the *2 file if executing at the terminal or as the card punch if executing on the batch.

The following shows a SNOBOL source and data file ready for execution. The program (which deletes any '*' found in column 1) uses the input stream for input and writes the output to the *2 file. Note that the SNOBOL UNLIST option is used to suppress the source listing.

```
/inc snobol4
- unlist
        &trim = 1
main    in = input                  :f(end)
        in pos(0) '*' =
        punch = in                  :(main)
end
this is some test data...
* <- this asterisk will be deleted from this output record
if this program was run
* this one too ...
```

Input/Output Associations

Besides the default I/O variables available in SNOBOL, the programmer may define associations using the INPUT and OUTPUT functions. The forms of the functions are shown below:

```
INPUT(name,number,length)
OUTPUT(name,number,format)
```

Where 'name' is the variable name, 'number' is the FORTRAN unit number, 'length' is the record length, and 'format' is a FORTRAN format. For example, the default I/O variable associations correspond to the following:

```
INPUT('INPUT',5,80)
OUTPUT('OUTPUT',6,'(1x,132A1)')
OUTPUT('PUNCH',7,'(80A1)')
```

Whenever a non-defaulted unit number is used, a corresponding /FILE

statement must also appear in the job stream.

In the following example, the program shown earlier has been modified to write the output to the *2 file and a tape file.

```
/file unit=(tape,name=ft12f001),dsn=snoout,vol=ser=outtape,
/ recfm=fb,lrecl=80,blksize=8000,disp=old,label=(1,nl)
/inc snobol4
- unlist
          output('tape',12,'(80a1)')
          &trim = 1
main      in = input                    :f(end)
          in pos(0) '*' =
          punch = in
          tape = in                     :(main)
end
```

If the above example was saved as the VPS library file SNOSTRP, to execute it using the file TEXT as input, the following could be used:

```
*GO
/ex snostrp,text
```

Chapter 10: WATFIV

WATFIV, which was developed at the University of Waterloo, is a superset of the FORTRAN IV G language. The name WATFIV stands for Waterloo FORTRAN IV. WATFIV is designed for the rapid compilation of small FORTRAN programs and provides outstanding source and object program diagnostics.

WATFIV provides extensive diagnostic tools including an interactive debugging facility for debugging programs at the terminal. It has proved to be about ten times faster than FORTRAN G1 in terms of compilation and only 20 to 40 per cent slower than FORTRAN G1 in terms of execution. Thus the use of WATFIV is highly recommended whenever the user is debugging a FORTRAN program.

The Language Supported by WATFIV

Besides supporting the FORTRAN language accepted by the IBM FORTRAN compilers, WATFIV also accepts a number of extensions to that language. The most powerful of these is a set of structured programming statements designed to enhance the power of the standard FORTRAN. The specifications of WATFIV, including extensions to the FORTRAN language, are contained in the publication "FORTRAN IV with WATFOR and WATFIV" by Cress, Dirksen, and Graham (Prentice Hall, 1970) and as a handout available at the Computing Center.

Invoking WATFIV

WATFIV may be run under the VPS batch or from VPS terminals. The WATFIV compiler is invoked by use of the following control statement which must be placed before the first WATFIV statement in a job:

```
/LOAD WATFIV
```

This statement will invoke WATFIV with a memory allocation of 3 segments (192K) which is large enough for a program of approximately 150 statements. For larger programs, the NSEGS= parameter of the /LOAD statement should be used (see The VPS Handbook).

WATFIV Control Statements

WATFIV recognizes the following control statements:

```
$JOB <options>
```

This statement is required and is used to specify WATFIV compiler options. It must immediately precede the first WATFIV source statement of a program.

```
C$OPTIONS <options>
```

This statement is optional and can be used to redefine compiler options during compilation. It may appear between any 2 WATFIV source

statements.

$ENTRY

This statement is used to separate source statements from program data in the input stream. It is required even if no data appears in the input stream.

C$PROFON

This statement is used to activate the collection of PROFILER statistics (see 'Compiler Options' later in this chapter for further information). It may appear between any 2 WATFIV source statements.

C$PROFOFF

This statement is used to disable the collection of PROFILER statistics. It may appear between any 2 WATFIV source statements.

As with the /LOAD WATFIV statement, each control statement must begin in column 1. WATFIV allows multiple program executions per job as long as each program source is preceded by a $JOB statement and followed by a $ENTRY statement. The standard WATFIV job setup is shown below:

```
/LOAD WATFIV
$JOB option,option,...

    (WATFIV source statements)

$ENTRY

    (data, if any)

$JOB option,option,...

    (WATFIV source statements)

$ENTRY
  .
  .
  .
```

WATFIV Compiler Options

The following options may be specified on either the $JOB statement or the C$OPTIONS statement. Options must start on column 6 of the $JOB statement or column 11 of the C$OPTIONS statement. The options must be separated by commas and contain no embedded blanks. Where applicable, an underline indicates the default option on VPS.

Option	Description
LIST NOLIST	Controls the printing of a source listing. The default on the terminal is NOLIST. The default on the batch is LIST.
CHECK NOCHECK FREE	Controls the checking of attempted use of undefined variables at execution time. CHECK will cause execution to terminate if an undefined variable is used. NOCHECK will suppress checking. FREE checks for use of undefined variables but allows execution to continue.
WARN NOWARN	Controls the printing of all diagnostic messages of a severity less than a fatal error.
EXT NOEXT	Controls the printing of extension messages - indications of any WATFIV feature that may not work under an IBM FORTRAN compiler.
NARROW NONARROW	Controls the printing of PROFILER and XREF output in either 80 character lines (NARROW) or the standard 133 character lines (NONARROW).
DECIMAL	Causes REAL*4 values output under free-format to appear in a form that is more readable (F20.8) rather than the standard E-type format.
PROFC PROFP PROFA PROF NOPROF	Controls the collection of WATFIV PROFILER statistics output. PROFILER output will be generated for sections of a program between C$PROFON and C$PROFOFF statements. If C$PROFOFF is not encountered in a program and PROFC, PROFP, PROFA, or PROF is specified, statistics will be gathered from the C$PROFON statement on. PROFC specifies the printing of the profile count option. PROFP specifies the printing of the percentage histogram option. PROFA specifies the printing of the absolute histogram option. PROF specifies the printing of all of the above options.
XREF NOXREF XREFP	Controls the printing of the cross-reference. XREF produces a full cross-reference at one line per item. XREFP produces a more economical packed cross-reference.

WATFIV Input/Output Units

As was discussed above, to invoke WATFIV under VPS, a /LOAD WATFIV control statement must be placed before the first WATFIV statement of a job. In reality, not only does the /LOAD statement call the compiler, but it also invokes a VPS internal procedure. A VPS procedure is a set of VPS control statements needed to run average source programs. These control statements may be overridden at

execution time by programmer-supplied statements. The WATFIV procedure invoked using /LOAD WATFIV at the terminal is shown below:

```
/FILE UNIT=(TERMOUT,NAME=FT06F001)
/FILE UNIT=(LIBIN,NAME=FT05F001),DSNAME=<input stream>
/FILE UNIT=(LIBOUT,NAME=FT07F001),DSNAME=*2
/FILE UNIT=(TERMIN,NAME=FT09F001)
/LOAD WATFIV
```

This procedure provides /FILE statements for standard FORTRAN units 5, 6, 7, and 9. FORTRAN units, or reference numbers, refer to files defined on /FILE statements. The format of the NAME= subparameter of the UNIT= parameter is as follows:

```
NAME=FTxxF001
```

Where 'xx' is the FORTRAN unit number. Therefore, FORTRAN unit 6 is FT06F001, FORTRAN unit 14 is FT14F001, etc.

The default /FILE statements shown above in the VPS internal procedure are shown below:

File	Use
FT05F001	FORTRAN unit 5 - defined as the input stream after the $ENTRY statement.
FT06F001	FORTRAN unit 6 - defined as terminal output if executing at the terminal or the high speed printer if executing on the batch.
FT07F001	FORTRAN unit 7 - defined as the *2 file if executing at the terminal or as the card punch if executing on the batch.
FT09F001	FORTRAN unit 9 - defined as terminal input if executing at the terminal or the input stream if executing on the batch.

WATFIV I/O processing is compatible with the IBM FORTRAN G1 and H Extended compilers. Thus WATFIV supports such facilities as sequential and direct access work file processing in the same manner as the IBM FORTRANs. All VPS input/output units, as described in Chapter 43 of this manual, are supported by WATFIV.

WATFIV Sample Job

The following shows the creation of a WATFIV program using the VPS editor and the execution of the job at the terminal.

```
*GO
/edit
INPUT
/load watfiv
$job
      read(5,1) a,b
  1   format(2f4.0)
      c=a+b
      write(6,2) c
  2   format(' the sum is ',f4.0)
      stop
      end
$entry
   2   7

EDIT
save adder
*SAVED
*GO
/exec adder

          $JOB NOLIST


          $ENTRY


THE SUM IS 9.


STATEMENTS EXECUTED=   3

CORE USAGE  OBJECT CODE=  336 BYTES, ARRAY AREA= 0 BYTES,
            TOTAL AREA AVAILABLE= 54448 BYTES

DIAGNOSTICS NUMBER OF ERRORS=  0, NUMBER OF WARNINGS=  0,
            NUMBER OF EXTENSIONS=  0

COMPILE TIME= 0.03 SEC,EXECUTION TIME= 0.01 SEC,
            8.49.08  WEDNESDAY  3 OCT 79  WATFIV - JUN 1977 V1L6

          C$STOP
*GO
```

Interactive Debugging

WATFIV provides facilities for monitoring the execution of a WATFIV program interactively at a terminal. No additions or changes to a program are required in order to use this facility. A special command set is provided to allow the programmer to trace portions of a program, halt execution at various points, display or modify program variables, alter logic flow of the program and correct certain execution time errors interactively. Note that these commands do not change the program source, that must be done by the programmer with the VPS editor.

To invoke the WATFIV interactive debugging facilties place the following VPS control statement immediately after the /LOAD WATFIV statement:

```
/PARM XDEBUG
```

This will cause WATFIV to activate the interactive debugging facility immediately after compilation of the program but before execution. At this point, WATFIV will ask the user to enter one of the WATFIV Debug commands with the following prompt:

```
CMD:
```

Each time the debugging facilities are entered during execution of the program, this prompt will appear. The Debug commands which can be used are listed below.

Command	Description
TRACE lnum,range	Turns on statement execution tracing from line number 'lnum' for a range of 'range' lines. Only one tracing range can be in effect at one time.
OFF	Turns statement tracing off.
STOP lnum	Specifies a line number 'lnum' at which the program is to be stopped and WATFIV Debug entered. Only one stop can be in effect at a time. When program execution halts the stop is cleared.
RUN	Resume execution where the program stopped.
GOTO lnum	Resume execution at line numbered 'lnum'.
a ?	Display the contents of variable 'a', where 'a' may be a simple variable, array name, or array element.

a = value	Modify the contents of variable 'a', where 'a' may be a simple variable, array name, or array element.
EXIT	Terminates the debug session and returns to VPS.
STEP n	Execute 'n' statements before returning to Debug.

If an error is detected in execution while running under WATFIV Debug, the error message will be printed and the interactive debugging facility will prompt the programmer for Debug commands.

Interactive Debugging Sample Program

In the following example a WATFIV program has been created in the VPS library file WATPROG. The program contains an infinite loop. The program is listed and then executed using the WATFIV interactive debugging facilities.

```
*GO
/list watprog
/LOAD WATFIV
/PARM XDEBUG
$JOB LIST
      READ(5,1,END=99) A,B
   1  FORMAT(2F4.0)
   5  C=A+B
      WRITE(6,2) C
   2  FORMAT(' THE SUM IS ',F4.0)
      GO TO 5
  99  STOP
      END
$ENTRY
  2   7
*GO
```

The following is the debugging session for this program.

```
*GO
/exec watprog


          $JOB LIST
      1           READ(5,1,END=99) A,B
      2       1   FORMAT(2F4.0)
      3       5   C=A+B
      4           WRITE(6,2) C
      5       2   FORMAT(' THE SUM IS ',F4.0)
      6           GO TO 5
      7      99   STOP
      8           END


          $ENTRY


CMD:
trace 1,999
CMD:
stop 3
CMD:
run
LINE     1
STOPPED AT LINE    3
CMD:
b?
  0.7000000E 01
CMD:
step 1
(      4) CMD:
c?
  0.9000000E 01
CMD:
stop 3
CMD:
run
THE SUM IS   9.
STOPPED AT LINE    3
CMD:
a?
  02.000000E 01
CMD:
a=16
CMD:
b=4
CMD:
step 1
(      4) CMD:
c?
  0.2000000E 02
CMD:
```

```
stop 3
CMD:
run
THE SUM IS  20.
STOPPED AT LINE    3
CMD:
exit

STATEMENTS EXECUTED=  11

CORE USAGE  OBJECT CODE= 368 BYTES,ARRAY AREA= 0 BYTES,
            TOTAL AREA AVAILABLE= 54448 BYTES

DIAGNOSTICS NUMBER OF ERRORS= 0, NUMBER OF WARNINGS= 0,
            NUMBER OF EXTENSIONS= 0

COMPILE TIME= 0.04 SEC,EXECUTION TIME= 0.20 SEC,
            11.53.29   WEDNESDAY  3 OCT 79   WATFIV - JUN 1977 V1L6
```

Chapter 11: XPL

XPL is a compiler generator that was developed at Stanford University and the University of California at Santa Cruz between 1966 and 1969. The compiler generating language implemented in XPL is documented in "A Compiler Generator" (Prentice-Hall, 1970).

The compiler itself is an XPL program that translates compiler generation statements supplied by the programmer into an object form. The XPL object is then loaded and used to process input supplied also by the program. Both XPL and the XPL generated object run under a submonitor that provides an OS/VS interface. Special VPS system support programs provide an "OS/VS" environment in which the submonitor, XPL, and the translated object run. XPL and its submonitor have not been modified to run under VPS.

Invoking XPL

The XPL compiler is invoked by placing the following control statement before the first XPL source statement in an XPL program:

```
/LOAD XPL
```

Normally, this is done by placing the /LOAD statement first in a source file with the VPS editor when the source file is created.

In reality, not only does the /LOAD statement call the XPL compiler, but it also invokes a corresponding VPS internal procedure. A VPS internal procedure is a set of VPS control statements needed to run average source programs. These control statements may be overridden at execution time by programmer-supplied statements. The XPL procedure invoked using /LOAD XPL is shown below:

```
/FILE UNIT=(DISK,NAME=FILE1),SPACE=(TRK,(10,5)),RECFM=F,LRECL=18000,
/ BLKSIZE=18000
/FILE UNIT=(DISK,NAME=FILE2),SPACE=(TRK,(10,5)),RECFM=F,LRECL=18000,
/ BLKSIZE=18000
/FILE UNIT=(DISK,NAME=FILE3),SPACE=(TRK,(10,5)),RECFM=F,LRECL=18000,
/ BLKSIZE=18000
/FILE UNIT=(DISK,NAME=PROGRAM),DSNAME=UR.CCMM.XPL.XCOM.COMPILER,
/ DISP=SHR
/FILE UNIT=(LIBIN,NAME=INPUT2),DSNAME=XPLLIBRY,DISP=SHR
/FILE UNIT=(LIBIN,NAME=SYSIN),DSNAME=<input stream>
/FILE UNIT=(TERMOUT,NAME=SYSPRINT)
/FILE UNIT=(LIBOUT,NAME=SYSPUNCH),DSNAME=*2
/LOAD XPL,NSEGS=4
```

Note that the space allocations for FILE1, 2, and 3 and the memory allocation (NSEGS) are large enough to compile the XPL compiler itself (4200 statements).

Procedure Files

The /FILE statements in the XPL procedure are described below:

File	Use
FILE1	Defined as a temporary work file used to hold the output XPL object of the compiler which, in turn, is loaded by the submonitor and used to process the programmer-supplied input.
FILE2	Defined as a temporary work file and used as a scratch file by the XPL compiler.
FILE3	Defined as a temporary work file and used as a scratch file by the XPL compiler.
INPUT2	Defined as a library file containing the dynamic string compactification procedure.
PROGRAM	Defined as a permanent work file - UR.CCMM.XPL.XCOM.COMPILER - containing the XPL compiler unless ANALYZE is specified on the /JOB statement in which case it is - UR.CCMM.XPL.SYNTAX.ANALYZER - containing the XPL syntax analysis program (see below).
SYSIN	Defined as the input stream - during compilation, from the first XPL statement to the /DATA statement (see below); during during execution, records after the /DATA statement. Used when reading from INPUT0 or INPUT1.
SYSPRINT	Defined as the terminal when executing at the terminal or as the high-speed printer when executing on the batch. Used by XPL and the user program as the output listing files - OUTPUT0 or OUTPUT1.
SYSPUNCH	Defined as the *2 file when executing at the terminal or as the card punch when executing on the batch. Used by XPL programs as the output file OUTPUT2.

These files may be overridden or other files defined at execution time by placing the appropriate /FILE statement(s) before the /LOAD statement in the job stream.

Files are referenced in XPL programs using the pseudovariables

input(i) or output(i)

where: i = 0, 1, 2, 3

These pseudovariables correspond to the ddnames - INPUT0, INPUT1, ..., OUTPUT0, OUTPUT1, etc. However, as noted above, for some unknown reason the XPL submonitor translates the ddnames INPUT0 and INPUT1 to SYSIN, OUTPUT0 and OUTPUT1 to SYSPRINT, and OUTPUT2 to SYSPUNCH. Therefore, if the programmer wishes to override these files, instead of placing /FILE statements in the job stream with the ddnames INPUT0, INPUT1, etc.; the ddnames SYSIN, SYSPRINT, and SYSPUNCH must be used. Of course, if you wish to override INPUT2 or define INPUT3 or OUTPUT3 that ddname must be used. (At this time we would like to thank the authors of XPL for their contribution to the field of human engineering.)

Separating Source from Program Input

As shown above, the XPL compiler and, possibly, the user program may read from the file SYSIN. The /DATA statement is used to separate source statements in the input stream from any records which are to be read by the program from SYSIN. A standard XPL job stream has the following structure:

```
/LOAD XPL

    XPL source statements

/DATA

    program input
```

Other XPL Files

Two other files of interest to XPL programmers can be found in the VPS library. The file XPLBNF contains the BNF grammar for XPL expressed in a form suitable for input to the syntax analysis program. The file XPLSKELE contains the XPL source for the skeletal compiler. Part or all of these files may be incorporated into a programmer's source file using the VPS editor.

The /JOB Statement

The /JOB statement may be used to select the type of processing desired. If the /JOB statement is not used, XPL source statements will be compiled with the XPL compiler and the resulting XPL object will be loaded and executed. Using the /JOB statement the programmer may request compilation only (COMPILE), execution only (EXECUTE), or the XPL syntax analysis program (ANALYZE).

If used, the /JOB statement must follow the /LOAD XPL statement in the job stream. The XPL /JOB statement has the following format:

```
/JOB    COmpile
        EXecute
        ANalyze
```

Note that the keywords may be abbreviated down to the first two characters. If COMPILE is specified, the XPL compiler will be used to

compile the XPL source statements writing XPL object to FILE1. If EXECUTE is specified, input to the XPL (or user) compiler is assumed to follow the /JOB statement (see "Separating Compilation from Execution in Time" below for a complete explanation of the use of this option). If ANALYZE is specified, the XPL source statements will be analyzed with the XPL syntax analysis program.

Separating Compilation from Execution in Time

It is not necessary to recompile an XPL program each time the programmer wishes to execute a program if the XPL source has not been modified. In fact, it is highly advantageous not to recompile since compilation is costly and time consuming. To avoid recompiling, all the programmer need do is save the XPL object from an XPL compilation and then execute the object.

Because of the nature of XPL, saved object can only be kept in a work file. Before saving any XPL object on VPS the programmer must first obtain a work file allocation from the Computing Center. The XPL source program should then be compiled with the compile only option and a permanent work file created containing the XPL object by overriding FILE1. In the following example the XPL object is saved in the work file - ur.prg976.xplobj4:

```
/FILE UNIT=(DISK,NAME=FILE1),DSN=UR.PRG976.XPLOBJ4,DISP=(NEW,KEEP),
/ RECFM=F,LRECL=18000,BLKSIZE=18000
/LOAD XPL
/JOB COMPILE
  XPL source statements
```

After the compilation, the program may be executed any number of times by specifying the execute only option and overriding the PROGRAM file to use the saved XPL object. Note that in this case a /DATA statement should not be used and the program input should be placed immediately after the /JOB statement. In the following example, the object created in the above example is used in execution:

```
/FILE UNIT=(DISK,NAME=PROGRAM),DSN=UR.PRG976.XPLOBJ4,DISP=SHR
/LOAD XPL
/JOB EXECUTE
  program input
```

ABEND Codes Returned by the Submonitor

The submonitor will return the following user completion codes in the event of errors:

Code	Description
100	Error in opening PROGRAM, SYSPRINT, or SYSIN.
200	Unexpected end of file encountered reading PROGRAM.
300	I/O error while reading PROGRAM.

400	Not enough memory to load program.
500	Invalid request to submonitor.
800+i	I/O error on OUTPUTi.
900+i	Reference to OUTPUTi specifies a nonexistent file.
1000+i	I/O error on INPUTi.
1200+i	Program tried to read past end of file on INPUTi.
1400+i	Reference to INPUTi specifies a nonexistent file.
2000+i	I/O error on FILEi.
2200+i	Program tried to read past end of file on FILEi.
4000	XPL program requested termination by executing the statement - call exit;

Appendix A: The /JOB Statement

The /JOB statement may be used with FORTRAN, PL/I, PASCAL, or COBOL to pass parameters to the VPS system programs that provide the "OS/VS" environment in which these compilers and their compiled programs run. The /JOB statement must appear after the /LOAD statement and before the first source or object code record. Multiple /JOB statements may be placed in the job stream, but continuation is not supported. The format of the /JOB statement appears below.

```
/JOB  [GO]  [,LET]    [,MAP]    [,SEARCH]
      [NOGO] [,NOLET] [,NOMAP] [,NOSEARCH]

      [,PRINT]   [,SUMRY]   [,SYSLIB=( syslibname   ,...)]
      [,NOPRINT] [,NOSUMRY]            ddname

      [,MEMBER=lmname]   [,JOBLIB=(  ddname      ,...)]
                                     syslibname
```

[LOG - prints OS SVC's handled]

GO | NOGO
: Specifies whether the program should be executed after it has been compiled.

LET | NOLET
: Specifies whether a program should be executed if, during loading of the object, unresolved external references are detected.

MAP | NOMAP
: Specifies whether the Loader should produce a map of module locations and entry points.

PRINT | NOPRINT
: Specifies whether the Loader should print the informatory message giving the total length of the loaded program.

SEARCH | NOSEARCH
: Specifies whether the Loader should attempt to resolve external references from the subroutine libraries (see SYSLIB, below).

SUMRY | NOSUMRY
: Specifies whether the Loader should print the informatory message giving the entry point of the loaded program.

SYSLIB=

Specifies the subroutine library or libraries to be searched to resolve external references after the object module(s) have been processed. A single system library name or ddname (from the NAME= keyword of a /FILE statement) or a list of system library names and ddnames enclosed in parentheses and separated by commas may be specified. The libraries will be searched in the order specified. The default search list for each compiler can be found in the respective chapter of this manual. If a ddname is specified, a corresponding /FILE statement must appear in the job stream defining the subroutine library to be referenced.

MEMBER= (for load module execution only)

Specifies the name of the load module to be loaded into storage and executed.

JOBLIB= (for load module execution only)

Specifies the load module library or libraries to be searched for the load module name specified in the MEMBER= keyword and for any other load modules which may be loaded during execution. A ddname (from the NAME= keyword of a /FILE statement) or list of ddnames and/or system library names enclosed in parentheses and separated by commas may be specified. If a ddname is specified, a corresponding /FILE statement must appear in the job stream defining the load module library to be referenced.

Appendix B: Creating and Executing Load Modules

Unlike PLC, WATFIV, and ALGOLW (to name a few), the FORTRAN, PL/I, and COBOL compilers and the Assembler are designed to produce object modules which can be saved in the VPS library and executed at a later time, bypassing the compilation phase. In the chapters describing each of these compilers we have attempted to explain ways in which execution from object can dramatically improve the efficiency of running jobs.

This improvement can be taken a step further by creating load modules out of object for programs which are executed frequently and modified infrequently. When object is loaded into memory by the VPS Loader, among other things, external references must be resolved and, if necessary, compiler subroutines (in load module form) loaded. Each time object is executed this process must be performed. For load modules, this process takes place only once, when the load module is created using the VPS Linkage Editor.

It is not our intent to document the Linkage Editor which is fully described in the VPS Utilities Manual, rather we assume the reader is familiar with the Linkage Editor and in this section we will explain its use when creating load modules from FORTRAN, PL/I, and COBOL programs.

Creating Load Modules

There are two prerequisites which must be satisfied before a load module can be created on VPS. They are:

1) The program or programs must be compiled and object produced.

2) A work file allocation must be obtained from the Computing Center for the load module library. This is necessary since load modules can exist only in work files.

Assuming these tasks are accomplished, load modules may now be created and placed in the work file. It should be noted that multiple load modules may be placed in the same work file (if it is large enough) and each load module is wholly independent of the other modules in the work file. In other words, a work file may contain FORTRAN, PL/I, COBOL, etc. programs in load module form.

As documented in the VPS Utilities Manual, the Linkage Editor has a number of different options which may be passed to it on a /JOB statement. For example, when creating a new load module library, FORMAT must be specified on the /JOB statement. Another important option is the SYSLIB= parameter which defines the list of load module libraries that the Linkage Editor is to search to resolve external references. Below is the minimum list the programmer must specify for the indicated compiler:

The FORTRAN compilers -

```
/JOB SYSLIB=SYS.FORTLIBX
```

The PL/I compiler -

```
/JOB SYSLIB=SYS.PLIXLIB
```

The COBOL compiler -

```
/JOB SYSLIB=SYS.COBLIB
```

In the following example a load module library is created for a FORTRAN program, whose object is in the library file AMAIN, and two subprograms, whose object is in the file SUBS. The load module is given the name AMAINF.

```
/FILE UNIT=(3,DISK),DSNAME=UR.PLG100.LOADLIB,DISP=(NEW,KEEP),
/ SPACE=(TRK,6),DSORG=DA,OPT=FORMAT
/LOAD LKED
/JOB FORMAT MAXMEM=10 LIST MAP SYSLIB=SYS.FORTLIBX
/INC AMAIN,SUBS
 NAME AMAINF
```

In the next example another load module is placed in the same library. The PL/I program object is included from the *2 file and the Linkage Editor will search PLIXLIB and TEKLIB for external references. The load module is given the name PLOT4.

```
/FILE UNIT=(3,DISK),DSNAME=UR.PLG100.LOADLIB,DISP=OLD
/LOAD LKED
/JOB SYSLIB=(SYS.PLIXLIB,SYS.TEKLIB) LIST MAP
/INC *2
 NAME PLOT4
```

Executing Load Modules

After a load module is created with the Linkage Editor it can then be used in place of the object to execute the program. For Assembler load modules which do not require the "OS/VS" environment for execution, the /LOAD statement as documented in the VPS Handbook should be used. For FORTRAN, PL/I, and COBOL load modules and Assembler load modules which require the "OS/VS" environment the following /LOAD statement must be used:

```
/LOAD OSLOAD
```

OSLOAD will invoke the system programs that provide the "OS/VS" environment.

Following the /LOAD statement a /JOB statement must appear with the parameters MEMBER=, indicating the load module name to be executed, and JOBLIB=, indicating the /FILE statement which defines the load module library. Note that a /FILE statement must appear in the job stream defining the load module library. Both parameters are fully

documented in Appendix A. For PL/I or COBOL load modules the JOBLIB list must contain either SYS.PLIXLIB, for PL/I, or SYS.COBLIB, for COBOL, since these programs require run time routines from the appropriate library that are not linked into the load module (commonly known as transient routines). For PL/I or COBOL load modules which use the Sort/Merge Package, the JOBLIB list must contain SYS.SORTLIB.

Like FORTG1, FORTX, PLIOPT, etc., OSLOAD invokes a VPS internal procedure which defines a number of default /FILE statements available for execution. The /FILE statements defined in this procedure are the same as those documented for execution in the FORTRAN and PL/I chapters of this manual. If any /FILE statements are to be overridden or others defined, they must appear before the /LOAD OSLOAD statement in the job stream.

If data is to be read from the input stream. It should be placed immediately behind the /JOB statement (/DATA statement should not appear before the data).

In the following example the load module PLOT4 is executed out of the work file created in the previous examples. Data in the library file PLOTSIN is to be read from the input stream (SYSIN) and a tape file created (PLOT4OT).

```
/FILE UNIT=(DISK,NAME=LINKLIB),DSNAME=UR.PLG100.LOADLIB,DISP=SHR
/FILE UNIT=(TAPE,NAME=PLOT4OT),VOL=SER=PLGTPE,LABEL=(1,NL),
/ RECFM=FB,LRECL=120,BLKSIZE=12000,DISP=(NEW,KEEP)
/LOAD OSLOAD
/JOB MEMBER=PLOT4,JOBLIB=(LINKLIB,SYS.PLIXLIB)
/INC PLOTSIN
```