CHAPTER 18

Program EPF Calling Sequence

The main entrypoint of a program EPF is invoked by the command environment with a standard calling sequence. This calling sequence consists of five arguments:

1. The command line supplied by the invoker

2. The command status set by the invoked program to indicate its level of success to the invoker

3. Information on the command processing state supplied by the invoker

4. A flag indicating whether the invoker desires a return value — that is, whether the invoker is treating the invoked program as a command function

5. A pointer set by the invoked program to point to the returned value structure

The complete calling sequence is illustrated near the end of this chapter; however, very few programs need to use all of the information and arguments provided by the command environment. In fact, most programs need accept only two or fewer arguments.

The invoker is the EPF$INVK subroutine, which is called either directly by user programs, by the EPF$RUN subroutine, or by the CP$ subroutine. EPF$RUN is, itself, called directly by user programs. CP$ is also callable by user programs, and is called by PRIMOS to execute a command.

TYPES OF CALLING SEQUENCES

There are five types of program EPF calling sequences, with various levels of complexity. They are:

1. The program calling sequence, which takes no command line and which returns no information

2. The command calling sequence, which accepts a command line and which returns a severity code

3. The command function calling sequence, which accepts a command line and which returns both a severity code and a pointer to the returned function value

4. The detailed command calling sequence, which is an extended form of the command calling sequence in that it also accepts detailed command processing information

5. The complete calling sequence, which combines the command function calling sequence with the detailed command calling sequence

The remainder of this chapter describes each of the above calling sequences.

Except for the program calling sequence, the EPF$INVK subroutine treats all program EPFs the same in that it passes all five arguments to the main subroutine of a program EPF. For the program calling sequence, the EPF$INVK subroutine detects that the main subroutine of the program EPF it will invoke accepts no arguments — it does this by examining the main subroutine's ECB — and it therefore invokes the main subroutine with no arguments.

The only differences between the calling sequences is how many arguments the main subroutine has been designed to accept. If it accepts fewer than five arguments, then the extra arguments passed to it are ignored by the main subroutine. (The PCL instruction, which performs procedure calls on Prime systems, handles this situation properly.) In fact, a main subroutine may accept five arguments but choose to ignore some or all of them.

The different calling sequences are therefore described only to simplify the construction of a main program. You should decide what kind of program you are writing by looking at the descriptions of the functionality each calling sequence provides throughout the rest of this chapter, and then choose the calling sequence that best suits your program.

PROGRAM CALLING SEQUENCE

The program calling sequence is the simplest calling sequence because it accepts no arguments. Any command line passed to such a program is ignored; no severity code is returned, so a severity code of 0 is assumed by the invoker; if the program is invoked as a command function, no pointer to the returned value is returned.

The calling sequence is not illustrated, because it consists of no input or output arguments.

A program whose main subroutine accepts no arguments may use the SETRC$ subroutine, described in the Subroutines Reference Guide, to return a severity code, even though it does not accept the severity code argument in its main subroutine. This feature is provided to allow the conversion to an EPF of an existing static-mode program that uses SETRC$ to be as easy as possible.

COMMAND CALLING SEQUENCE

The command calling sequence is used for programs that accept command line arguments and options and that return a severity code.

Arguments in the Command Calling Sequence

The command calling sequence is the simplest calling sequence that accepts arguments. It accepts two arguments:

1. The command line, an input-only argument

2. The severity code, an output-only argument

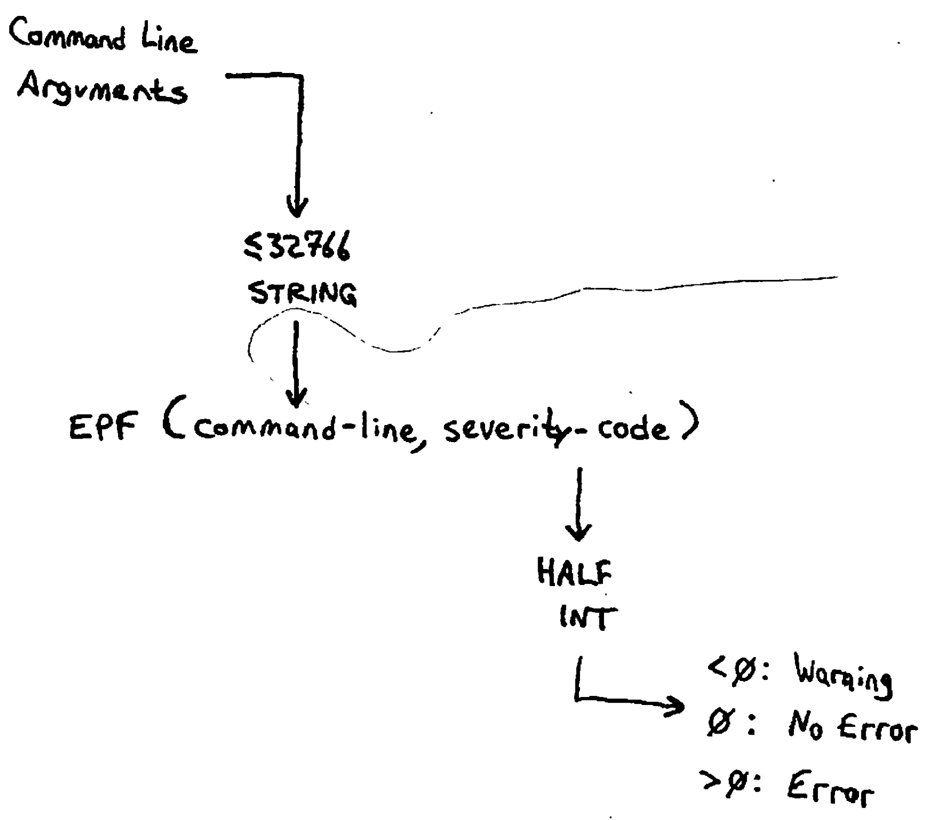
If a program that accepts only these two arguments is invoked as a command function, no pointer to the returned value is returned.

Figure 18-1 illustrates the command calling sequence, where EPF is the main subroutine of the program EPF.

Command Line: The length of the command line can be a maximum of 32,766 characters. Your program may limit the length to any value it chooses. Practical limits depend on the source of the command line. For example, the limit on the length of a command line entered by an interactive user or from a command input file is 160 characters; whereas the limit on the length of a command line in a CPL program is 1024 characters.

If your program is passed a command line longer than it can handle, it should use the error code E$TRCL as both a severity code and as an error code to ERRPR$ to indicate that the command line has been

Command Calling Sequence



Command Calling Sequence
Figure 18-1

truncated. If your program aborts due to this condition, then a truncated command line is an error; therefore, your program should return E$TRCL, a positive value, as the severity code. If your program continues processing, but uses a truncated form of the command line, your program should return -E$TRCL, a negative value, as the severity code (unless a positive error code is required for other reasons) to indicate a warning condition.

In PL1/G, you can use the LENGTH built-in function to check whether the length of the command line is greater than your program supports, even if you have declared the command line to be the maximum size your program supports. In FORTRAN and other languages, you can compare the first halfword of the command line argument, which is the actual length of the command line, to the maximum length your program supports.

If your program does not accept a null command line, it should use the E$NCOM to indicate that it has been passed a null command line. In addition, you may wish to have your program display usage information when passed a null command line; this is what many Prime-supplied programs, such as SPOOL and JOB, do with a null command line. Even if your program does display usage information, it should still return E$NCOM, a positive value, as the severity code to indicate an error.

Other error codes your program may wish to return as either positive values (to indicate errors) or as negative values (to indicate warnings), and which your program may also wish to use when calling ERRPR$ to display warning messages, are:

Error Code	Used For
E$BPAR	Invalid numeric arguments, arguments where a number was expected but some other argument was supplied.
E$BNAM	Invalid file system objectname arguments.
E$NMLG	Overly long names, such as a file system objectname that is more than 32 characters long.
E$ITRE	Invalid pathnames.
E$CMND	Invalid command formats, such as use of an option when no options are allowed, or use of command line arguments when no command line arguments are allowed.
E$BARG	Invalid arguments, such as use of an unrecognized option, or use of a name or number when an option was expected.
E$IVCM	Invalid usage of a command, such as a combination of options and arguments that is not permitted or that does not make sense.

E$MISA	Missing arguments, such as when a number, name, or option that is required is not provided on the command line.

All standard PRIMOS error codes, including those shown above, are listed along with their numeric equivalents, messages, and descriptions, in Appendix B.

Severity Code: Your program should set the severity code to an appropriate value before returning from its main subroutine. As indicated, the meaning of a severity code depends on whether it is negative, zero, or positive. The magnitude of the severity code is not defined by PRIMOS; however, your program should have documentation that describes the different severity codes it may return and what they mean. Typically, standard PRIMOS error codes, listed in Appendix B, are used for severity codes; to indicate warning conditions, the negated values of standard PRIMOS error codes are often used.

COMMAND FUNCTION CALLING SEQUENCE

The command function calling sequence is used when the program expects to be invoked as a command function. It may or may not expect command line arguments and options, and it may or may not return a severity code. Such a program returns a pointer to a structure that contains the returned value, a text string, that can be substituted for the invocation of the program as a command function on a command line.

The steps a command function performs are:

1. Accept five arguments in the main entrypoint calling sequence

2. Determine the string value to be returned to the calling program

3. Allocate memory for the string value to be returned

4. Copy the string value into the allocated memory

5. Store the pointer to the allocated memory into the pointer passed in the calling sequence of the main entrypoint

6. Return to the calling program

Step 1, accepting five arguments in the main entrypoint, is described below in the section entitled Arguments in the Command Function Calling Sequence. Step 2, determining the value to be returned, depends on the purpose of your program. Steps 3 and 4 are typically combined into one step by calling the ALS$RA subroutine, described below in the section entitled The ALS$RA Subroutine. Alternatively, they may be performed separately by calling the ALC$RA subroutine, described below in the section entitled The ALC$RA Subroutine, and then by copying the string value afterwards. Typically, only programs written in PL1/G and PMA perform Steps 3 and 4 separately.

Step 5 is often performed implicitly during Step 3 if ALS$RA or ALC$RA is passed the same variable accepted in the calling sequence of the main entrypoint; otherwise, your command function must explicitly set the <u>rtn-fcn-ptr</u> variable passed to it in the calling sequence of the main entrypoint so that it points to the structure allocated by ALS$RA or ALC$RA.

Step 6 is performed in the same way as for other types of programs. Your program should set the returned severity code to an appropriate value before returning.

After the next three sections, a section entitled <u>Sample Command Functions</u> presents two simple sample command functions.

Arguments in the Command Function Calling Sequence

The main subroutine of a command function accepts five arguments:

1. The command line, an input-only argument
2. The severity code, an output-only argument
3. An input-only argument that may be ignored by most command functions
4. The invocation form bit, an input-only argument
5. The returned value pointer, an output-only argument

Figure 18-2 illustrates the command calling sequence, where EPF is the main subroutine of the program EPF.

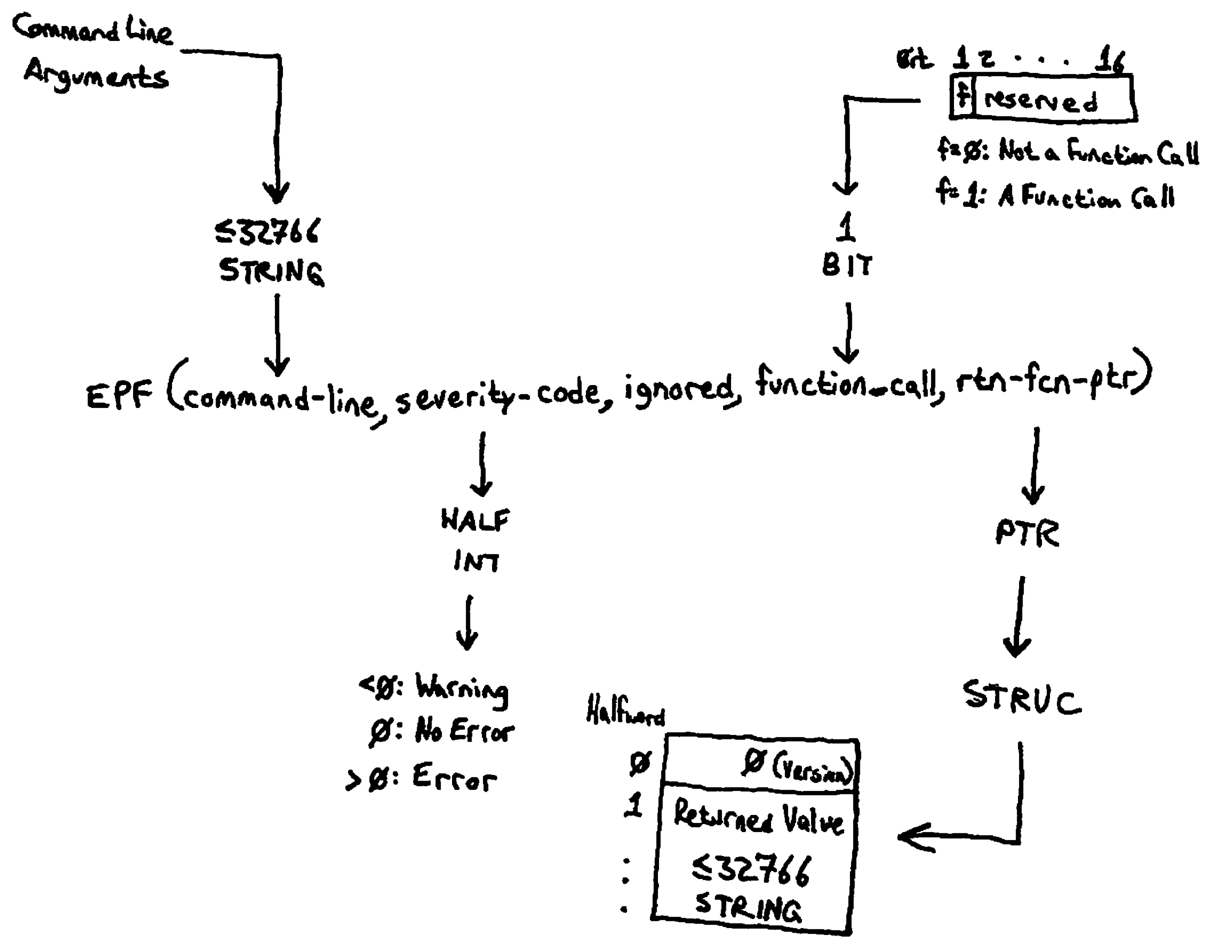
<u>Command Line</u>: See the section earlier in this chapter entitled <u>COMMAND CALLING SEQUENCE</u> for information on the command line. That information applies to command functions as well.

<u>Severity Code</u>: See the section earlier in this chapter entitled <u>COMMAND CALLING SEQUENCE</u> for information on the severity code. That information applies to command functions as well.

<u>Ignored</u>: The information passed to a program in the third argument may be ignored by most command functions. It is described in the next section, entitled <u>DETAILED COMMAND CALLING SEQUENCE</u>.

<u>Invocation Form</u>: The form of program invocation is a bit that

Command Function Calling Sequence



Command Function Calling Sequence
Figure 18-2

indicates whether the program is being invoked as a command function or as a normal command. When set (1), function-call indicates that the invoker expects the program to set rtn-fcn-ptr to point to a structure containing the returned value of the function. When reset (0), function-call indicates that the invoker does not expect the program to set rtn-fcn-ptr at all, and that in fact the invoker may not have supplied the rtn-fcn-ptr argument.

Caution

Under no circumstances should your program set rtn-fcn-ptr when function-call is reset (0), nor should your program allocate storage for the returned value. When function-call is reset (0), the fifth argument, rtn-fcn-ptr, may not be passed to your program, and any attempt that your program makes to set it may therefore result in a POINTER_FAULT$ error condition being signaled. If the fifth argument is passed, but function-call is reset (0), then your program may succeed at setting rtn-fcn-ptr, but the invoking program will not expect it to point to the returned structure, and will therefore not deallocate the memory used by the structure.

Returned Value Pointer: If your program has been invoked with the function-call bit of the calling sequence set (1), then the invoking program expects your program to return a pointer to a structure that contains the returned value. The returned value is a text string 0-32766 characters in length. The structure contains a version number (currently 0) as a HALF INT value and the returned value as a <=32766 STRING value.

Your program must return a pointer returned by one of the two allocation subroutines described below, ALS$RA or ALC$RA, in rtn-fcn-ptr. The calling program will use the FRE$RA subroutine, described in Chapter 19, to free the storage allocated by ALS$RA or ALC$RA, by passing to FRE$RA the pointer your program returns in rtn-fcn-ptr.

Caution

If your program does not use ALS$RA or ALC$RA to determine the rtn-fcn-ptr pointer, instead using a pointer constructed by other means, then when the calling program calls FRE$RA with the returned pointer, a fatal error will occur.

The ALS$RA Subroutine

The ALS$RA subroutine allocates sufficient memory to hold the supplied string value, copies the string value into the allocated memory, and returns the pointer to the allocated memory for use by the program that invoked the command function. The calling sequence for ALS$RA is illustrated in Figure 18-3.

Your program passes the string value to be returned in value and its size, in characters, in value-size. ALS$RA allocates sufficient memory (at least (value-size+5)/2 halfwords) to hold the string value, sets the first halfword of the allocated memory to 0 to indicate a version 0 returned value structure, stores the length of the string in value-size into the second halfword of the allocated memory, copies the string in value into the allocated memory starting with the third halfword, and returns a pointer to the first halfword of the allocated memory in rtn-fcn-ptr.

After calling this subroutine, all your program need do is ensure that the pointer returned by ALS$RA is returned by the main entrypoint of your program to the calling program by storing it into the rtn-fcn-ptr argument of the main entrypoint of your program. Then, your program simply returns to its invoker. The invoking program is responsible for deallocating the memory allocated by ALS$RA.

The ALC$RA Subroutine

The ALC$RA subroutine is similar to the ALS$RA subroutine, except that it does not copy the string value into the allocated memory. It leaves this task to your program, the command function.

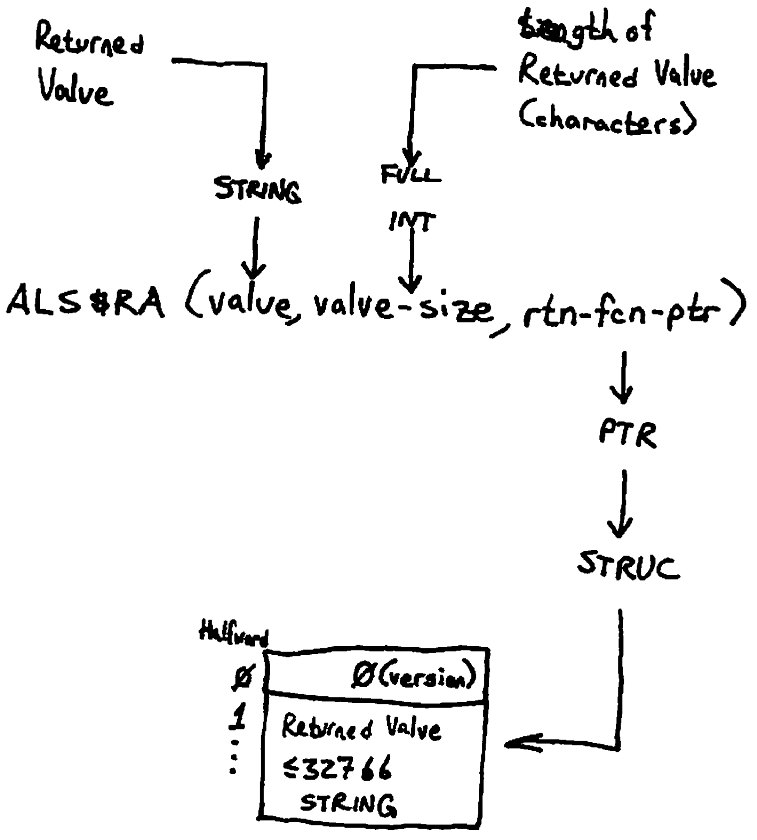
The ALC$RA subroutine allocates sufficient memory to hold a string value of the specified length and returns the pointer to the allocated memory for use by your program, the command function. The calling sequence for ALC$RA is illustrated in Figure 18-4.

Your program passes the number of halfwords to be allocated in halfwords. This value should be at least (value-size+5)/2, where value-size is the length of the string value to be returned. ALC$RA allocates the requested number of halfwords to hold the string value, and returns a pointer to the first halfword of the allocated memory in rtn-fcn-ptr.

After calling this subroutine, your program must set the first halfword of the allocated memory to 0 to indicate a version 0 returned value structure, set the second halfword of the allocated memory to the length of the string value in characters, then copy the string value into the allocated memory starting at the third halfword of the allocated memory. Because your program must use the rtn-fcn-ptr pointer to perform these tasks, only programs written in PL1/G and PMA typically use this interface.

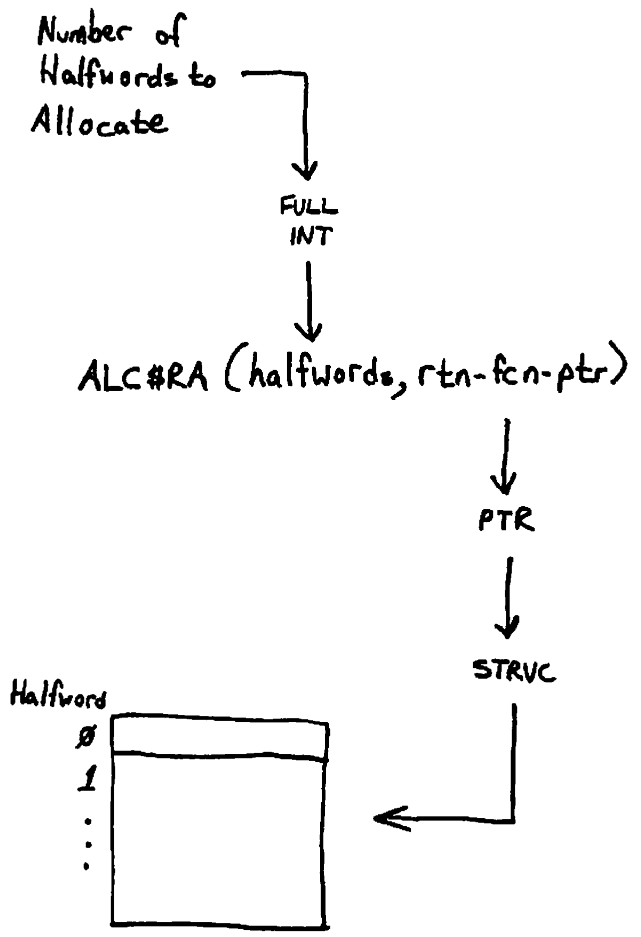
After copying the string value into the allocated memory, your program must ensure that the pointer returned by ALC$RA is returned by the main entrypoint of your program to the calling program by storing it into

Allocate and Set Retuned Function Value


Returned Value
~~Length~~ Length of Returned Value (characters)
STRING
FULL INT
ALS$RA (value, value-size, rtn-fcn-ptr)
PTR
STRUC
Halfword
Ø
1
Ø (version)
Returned Value
≤32766
STRING

The ALS$RA Subroutine
Figure 18-3

Allocate Space for Returned Function Value


Number of Halfwords to Allocate
FULL INT
ALC#RA (halfwords, rtn-fcn-ptr)
PTR
STRUC
Halfword
Ø
1
.
.
.

The ALC$RA Subroutine
Figure 18-4

the rtn-fcn-ptr argument of the main entrypoint of your program. Then, your program simply returns to its invoker. The invoking program is responsible for deallocating the memory allocated by ALC$RA.

Sample Command Functions

The first sample program is a FORTRAN program that returns the usernumber of the user invoking the program.

```
      SUBROUTINE USRNUM(COMLIN,CODE,IGN,FUNC,RTNPTR)
      INTEGER*2 COMLIN(1),CODE,IGN,FUNC
      INTEGER*4 RTNPTR(2)
C
$INSERT SYSCOM>ERRD.INS.FTN
$INSERT SYSCOM>KEYS.INS.FTN
C
      INTEGER*2
     &   U,             /* User number; later, units digit of U.
     &   TIMARR(12),    /* TIMDAT array.
     &   STR(2),        /* String value containing user number.
     &   STRLEN,        /* Number of characters in STRLEN.
     &   H,            ,/* Hundreds digit of U.
     &   T              /* Tens digit of U.
C
C Make sure we have no command line.
C
      IF (COMLIN(1).EQ.0) GO TO 10
C
C Reject attempted use of command line.
C
      CODE=E$IVCM                /* Invalid command error.
      IF (AND(FUNC,:100000).EQ.0)  /* Invoked as command?
     &   CALL ERRPR$(K$IRTN,CODE,'No command line accepted',24,
     &   'USERNUMBER',10)
      RETURN                     /* Return to invoker.
C
10    CALL TIMDAT(TIMARR,12)     /* Get user number in TIMARR(12).
      U=TIMARR(12)               /* For ease of access.
      IF (U.GT.9) GO TO 20       /* More than one digit?
      STR(1)=LS(U,8)+'0 '        /* Convert to single-digit ASCII.
      STRLEN=1                   /* Set to 1 digit.
      GO TO 100
C
20    H=U/100                    /* Get hundreds digit.
      U=U-H*100                  /* Get last two digits.
      T=U/10                     /* Get tens digits.
      U=U-T*10                   /* Get last digit.
      IF (H.NE.0) GO TO 30       /* Need three digits?
      STR(1)=LS(T,8)+U+'00'      /* No, make two digits into ASCII.
      STRLEN=2                   /* Indicate two digits.
      GO TO 100
```

```
C
30     STR(1)=LS(H,8)+T+'00'    /* Make three digits into ASCII.
       STR(2)=LS(U,8)+'0 '
       STRLEN=3                 /* Indicate three digits.
C
100    IF (AND(FUNC,:100000).NE.0) GO TO 200
C
C Not a function call; display user number.
C
       CALL TNOUA('Your user number is ',20)
       CALL TNOUA(STR,STRLEN)
       CALL TNOU('.',1)
       GO TO 300
C
C A function call; allocate and store user number.
C
200    CALL ALS$RA(STR,INTL(STRLEN),RTNPTR)
C
C Return to invoker.
C
300    CODE=0                   /* Success!
       RETURN
C
       END
```

The next sample program, written in PL1/G, returns the username of the invoking user.

```
username: proc(comlin,code,ign,func,rtn_fcn_ptr);

dcl comlin char(32) var,  /* Must be null. */
    code fixed bin(15),   /* Severity code. */
    ign fixed bin(15),    /* Ignored. */
    func bit(1),          /* Set if function call. */
    rtn_fcn_ptr ptr;      /* Returned function value pointer. */

%include 'SYSCOM>ERRD.INS.PL1';
%include 'SYSCOM>KEYS.INS.PL1';

dcl unam char(32) var;    /* Trimmed username. */

dcl 1 timarr,
      2 ignore (12) fixed bin(15),  /* Ignore 12 halfwords. */
      2 user_name char(32);  /* The username. */

dcl 1 rtn_struc based(rtn_fcn_ptr),
      2 version fixed bin(15),
      2 value char(32) var;

dcl timdat entry(1,2 (12) fixed bin(15),2 char(32),fixed bin(15)),
    errpr$ entry(fixed bin(15),fixed bin(15),char(40),
                 fixed bin(15),char(8),fixed bin(15)),
        alc$ra entry(fixed bin(31),ptr),
        tnou entry(char(60),fixed bin(15)),
        tnoua entry(char(60),fixed bin(15));

    if comlin='' then
       do;  /* No command line. */
       call timdat(timarr,28);
       unam=trim(user_name,'11'b);
       if func then
          do;  /* Command function invocation. */
          call alc$ra(divide(length(unam)+5,2,15),rtn_fcn_ptr);
          rtn_struc.version=0;
          rtn_struc.value=unam;
          end;  /* if func */
       else
          do;  /* Command invocation. */
          call tnoua('Your user name is ',18);
          call tnoua((unam),length(unam));
          call tnou('.',1);
          end;
       code=0;  /* Success. */
       end;  /* if comlin='' */
    else
       do;  /* if comlin^='' */
       code=e$ivcm;
       if ^func then
          call errpr$(k$irtn,code,'No command line accepted',24,
                      'USERNAME',8);
       end;  /* if comlin^='' */

    end;  /* username: proc */
```

DETAILED COMMAND CALLING SEQUENCE

The detailed command calling sequence adds a third argument to the command calling sequence described earlier in this chapter. This third argument is a structure passed to the program EPF being invoked that includes the following information:

- The command name as entered by the user
- A pointer to CPL local variables, if appropriate
- Command preprocessing information

Typically, a program EPF uses only the portions of the structure that are applicable to the program. For example, if you wish your program to display the command name entered by the user, rather than the original name of your program in error messages, you could have the main entrypoint of your program use only the command name as entered by the user and ignore the remainder of the structure.

This remainder of this section describes the information passed in the third argument of the program EPF calling sequence.

Arguments in the Detailed Command Calling Sequence

The detailed command calling sequence accepts three arguments:

1. The command line, an input-only argument

2. The severity code, an output-only argument

3. A structure containing command processing information, an input-only argument

If a program that accepts only these three arguments is invoked as a command function, no pointer to the returned value is returned.

Figure 18-5 illustrates the command calling sequence, where EPF is the main subroutine of the program EPF.

Command Line: See the section earlier in this chapter entitled COMMAND CALLING SEQUENCE for information on the command line. That information applies to command functions as well.

Severity Code: See the section earlier in this chapter entitled COMMAND CALLING SEQUENCE for information on the severity code. That information applies to command functions as well.

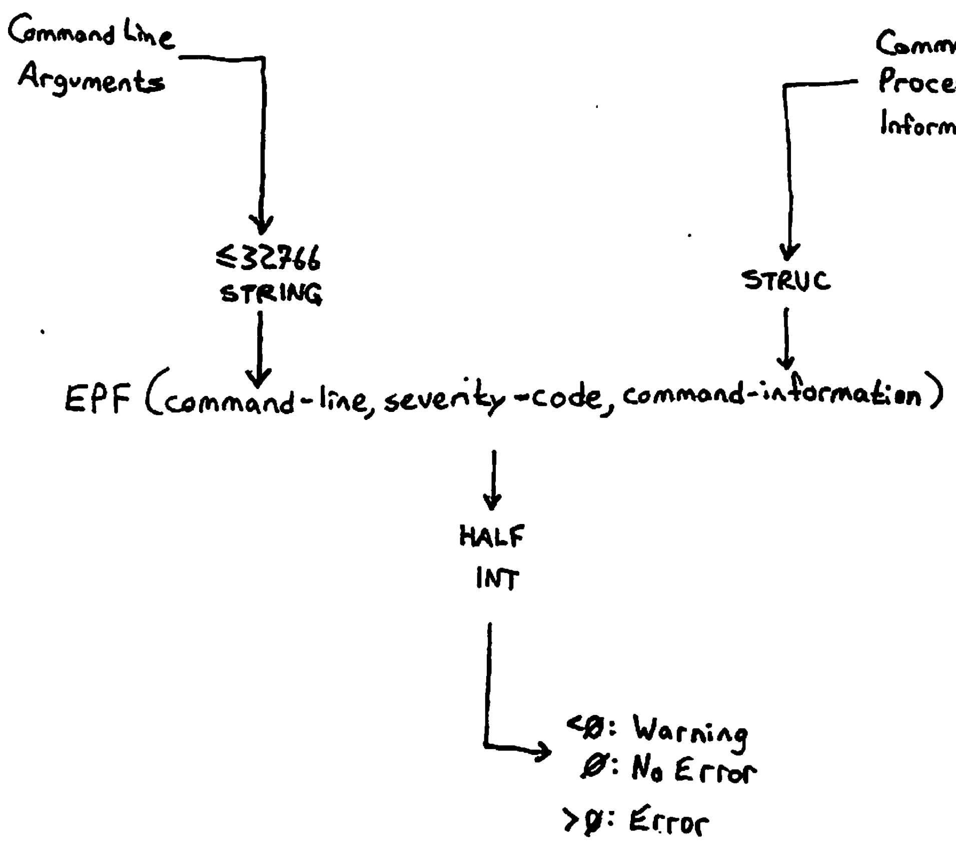
Command Processing Information: Figure 18-6 illustrates the command processing information, which is described in detail in the next section.

Currently, two versions of the command processing information structure are defined. The first two fields, the command name and the version number, are always present. If version is 0, the remainder of the command processing information structure is undefined and should not be referenced; only halfwords 0-17 (0-21 octal) are defined for a version 0 structure. If version is 1, the entire structure is defined as shown; that is, halfwords 0-25 (0-31 octal) are defined. Future versions of the structure will have higher version numbers and may define extensions to version 1 of this structure; however, the content and meaning of halfwords 0-25 will remain the same.

WARNING

Never store data into the command processing information structure for any purpose. Some calling programs may have declared only 18 halfwords of storage for a version 0

Detailed Command Calling Sequence


Command Line Arguments
Command Processing Information
≤32766 STRING
STRUC
EPF (command-line, severity-code, command-information)
HALF INT
<Ø: Warning
Ø: No Error
>Ø: Error

Detailed Command Calling Sequence
Figure 18-5

Command Processing Information (Versions Ø and 1)

Halfword oct	Halfword dec							Halfword dec	Halfword oct
Ø … 2Ø	Ø … 16	Command Name					≤32 STRING	Ø … 16	Ø … 2Ø
21	17	Version (Ø or 1)					HALF INT	17	21
22 23 24	18 19 2Ø	CPL Local Variables Pointer					PTR	18 19 2Ø	22 23 24
25	21	-DIR 1 BIT	-SEGDIR 1 BIT	-FILE 1 BIT	-ACAT 1 BIT	-RBF 1 BIT	Reserved 11 BIT	21	25
26	22	-VERIFY 1 BIT	-BOTUP 1 BIT	Reserved			14 BIT	22	26
27	23	-WALK-FROM Value					HALF INT	23	27
3Ø	24	-WALK_TO Value					HALF INT	24	3Ø
31	25	{} 1 BIT	@ + 1 BIT	{@} {+} 1 BIT	Reserved		13 BIT	25	31

Note: For a version Ø structure, only halfwords Ø-17 (Ø-21 octal) have defined values.

Command Processing Information
Figure 18-6

structure, representing halfwords 0-17, and any attempt to store beyond halfword offset 17 may corrupt memory. In addition, because the structure is an input argument to the program being invoked, the calling program may place the structure in memory that is protected against writing.

Your program should check the version number only if it needs to use information beyond halfword offset 17 (21 octal) into the command processing structure; and, in such a case, your program should check only that the version number is not 0 to ensure that the information being retrieved is valid. Do not reject version numbers higher than 1. However, if you choose, you may have your program reject version numbers that are negative, as such numbers probably indicate corrupted memory.

Command Processing Information

This section describes each field in the command processing information structure shown in Figure 18-6.

Command Name: The command name field contains the command name as specified by the user. The name may or may not include the .RUN suffix, but it will contain only the final element of a pathname. Your program may use this name rather than the name designed for it in messages displayed to the terminal, or your program may reject attempts to invoke it with a name other than that which it was designed to have.

Typically, the command name is the same name specified during the BIND session that linked the program. However, if a user copies your program to a file with a different name and invokes the copy, or if the name of the file containing the program is changed (via CNAME for example), the command name will be different from the original name of the program.

Version: The version number field contains the version number of the command processing structure. Currently, version numbers 0 and 1 are defined as described above. Higher version numbers will be used if future versions of PRIMOS extend the command processing information structure. The following table lists the currently defined version numbers and the halfwords that are defined (have meaningful values) in a structure with each version number listed:

Version	Defined Halfwords
0	0-17
1	0-25

CPL Local Variables Pointer: The CPL Local Variables Pointer is provided if the calling program is either a CPL program or a program EPF provided with a CPL Local Variables Pointer (ultimately invoked by a CPL program).

Sometimes referred to as the vcb_ptr, for Variables Control Block pointer, this pointer is used only when the program EPF wishes to read or set a CPL variable that is local to the CPL program that invoked the program EPF. Typically, such programs are designed as command functions, and the CPL program uses the &SET_VAR directive, as in:

```
    &SET_VAR MYVAR := [RESUME MYPROG]
```

However, a program that must reference more than one CPL variable must either be constrained to use only global variables (accessing them via the GV$GET and GV$SET subroutines) or must use the CPL Local Variables Pointer along with the LV$GET and LV$SET subroutines. A program constructed in the latter fashion might be invoked from a CPL program as follows:

```
    RESUME MYPROG MYVAR OTHERVAR
```

Here, the MYPROG program accepts two variable names, MYVAR and OTHERVAR in this example, and accesses them using LV$GET and LV$SET, which are described (along with GV$GET and GV$SET) in the Subroutines Reference Guide.

The CPL Local Variables pointer is NULL() (7777/0) if the invoking program is not a CPL program, or if it is not a program EPF invoked by a CPL program (either directly or via other program EPFs). A valid CPL Local Variables pointer is generated only by the invocation of a CPL program, and is valid only while that program is active; only program EPFs invoked by the CPL program, and their descendants, may use the Local Variables pointer for that CPL program.

Note

For maximum flexibility, design your program so that it accepts either global variables names beginning with a period (.) or local variable names not beginning with a period (.). Then, your program would call either GV$GET/GV$SET or LV$GET/LV$SET, depending on what type of variable name is supplied.

-DIRECTORY (-DIR) Bit: The -DIRECTORY bit is set if the command processor is matching file directories when checking wildcard-laden names. It does not necessarily mean that the file system object specified in the current invocation is a file directory.

-SEGMENT_DIRECTORY (-SEGDIR) Bit: The -SEGMENT_DIRECTORY bit is set if the command processor is matching segment directories when checking wildcard-laden names. It does not necessarily mean that the file system object specified in the current invocation is a segment directory.

-FILE Bit: The -FILE bit is set if the command processor is matching files when checking wildcard-laden names. It does not necessarily mean that the file system object specified in the current invocation is a file.

-ACCESS_CATEGORY (-ACAT) Bit: The -ACCESS_CATEGORY bit is set if the command processor is matching access categories when checking wildcard-laden names. It does not necessarily mean that the file system object specified in the current invocation is an access category.

-RBF Bit: The -RBF bit is set if the command processor is matching RBF files when checking wildcard-laden names. It does not necessarily mean that the file system object specified in the current invocation is an RBF file. (RBF files are reserved for use by Prime.)

-VERIFY (-VFY) Bit: The -VERIFY bit is set if the command processor requires user verification of file system objects selected by wildcard-laden names. It does not necessarily mean that the user has verified the file system object specified in the current invocation, because verification is requested only if the user specifies a wildcard-laden name. Use the wildcard bit, described below, if you wish to determine whether the user was actually asked to verify the current invocation for the file system object — if both the -VERIFY bit and the wildcard bit are set (1), then verification was both requested and provided.

-BOTTOM_UP (-BOTUP) Bit: The -BOTTOM_UP bit is set (1) if the -BOTTOM_UP option (abbreviatied -BOTUP) was specified on the command line, causing any treewalking to be performed at the lowest directory levels first. It does not necessarily mean that treewalking is being performed; see the treewalking bit, described below, for that information.

-WALK_FROM (-WLKFM) Value: The -WALK_FROM value is set to either the value specified following the -WALK_FROM option (abbreviated -WLKFM) on the command line or to the default value, which is 2. Level 1 is the contents of the directory itself; level 2 is the contents of the subdirectories, and so on. For example, in the treewalking specification DIR1>@@>FOO, level 1 is the DIR1 directory; if FOO exists in DIR1, it is found only if -WALK_FROM 1 is specified.

This value does not indicate whether treewalking is, in fact, being performed; see the treewalking bit, described below, for that information.

-WALK_TO (-WLKTO) Value: The -WALK_TO value is set to either the value specified following the -WALK_TO option (abbreviated -WLKTO) on the command line or the default value, which is 999. This value does not indicate whether treewalking is, in fact, being performed; see the treewalking bit, described below, for that information.

Iteration ( ) Bit: The iteration bit is set to '1'b if the command line used to invoke the program contained an iteration list (that is, contained parentheses). However, this bit is never set if the BIND subcommand NO_ITERATION (abbreviated NITR) was issued when the program was linked.

Wildcard @ + Bit: The wildcard bit is set to '1'b if the command line used to invoke the program contained a wildcard-laden entryname (that is, contained the @, +, or ^ character in the final element of a pathname or in a simple pathname). However, this bit is never set if the BIND subcommand NO_WILDCARD (abbreviated NWC) was issued when the program was linked.

Treewalk >@> >+> Bit: The treewalk bit is set to '1'b if the command line used to invoke the program contained a wildcard-laden directory name (that is, if it contained the @, +, or ^ character in a non-final element of a pathname). However, this bit is never set if the BIND subcommand NO_TREEWALK (abbreviated NTW) was issued when the program was linked.

Sample Program

The following sample PL1/G program simply displays all of the information in the command processing information structure. While it is intended primarily to illustrate how to declare and use the command processing information structure in PL1/G, it is also a useful program for experimenting with various combinations of command preprocessing features and BIND subcommands that enable, disable, or set parameters for command preprocessing features.

```
com_proc_info: proc(comline,code,cominfo);

dcl comline char(1024) var,        /* The command line. */
    code fixed bin(15),            /* Severity code. */
    1 cominfo,                     /* Command processing info. */
      2 comname char(32) var,      /* The command name. */
      2 version fixed bin(15),     /* Currently 0 or 1. */
```

```
        2 vcb_ptr ptr,              /* CPL local variables. */
        2 preprocessing_info,       /* Command preprocessing info. */
          3 mod_after_date fixed bin(31),
                                    /* -MODIFIED_AFTER date. */
          3 mod_before_date fixed bin(31),
                                    /* -MODIFIED_BEFORE date. */
          3 bak_after_date fixed bin(31),
                                    /* -BACKEDUP_AFTER date. */
          3 bak_before_date fixed bin(31),
                                    /* -BACKEDUP_BEFORE date. */
          3 type_dir bit(1),        /* -DIR option specified. */
          3 type_segdir bit(1),     /* -SEGDIR option specified. */
          3 type_file bit(1),       /* -FILE option specified. */
          3 type_acat bit(1),       /* -ACAT option specified. */
          3 type_rbf bit(1),        /* -RBF option specified. */
          3 reserved_1 bit(11),     /* Reserved for future use. */
          3 verify_sw bit(1),       /* -VERIFY option specified. */
          3 botup_sw bit(1),        /* -BOTUP option specified. */
          3 reserved_2 bit(14),     /* Reserved for future use. */
          3 walk_from fixed bin(15),
                                    /* -WALK_FROM value. */
          3 walk_to fixed bin(15),  /* -WALK_TO value. */
          3 in_iteration bit(1),    /* In iteration sequence. */
          3 in_wildcard bit(1),     /* In wildcard sequence. */
          3 in_treewalk bit(1),     /* In treewalk sequence. */
          3 reserved_3 bit(13);     /* Reserved for future use. */

%include 'SYSCOM>ERRD.INS.PL1';
%include 'SYSCOM>KEYS.INS.PL1';

dcl strings fixed bin(15), /* Number of strings. */
    last_string char(80) var, /* Last string. */
    line_to_show char(80) var;  /* Line waiting to be shown. */

dcl (tnoua,tnou) entry(char(80),fixed bin(15)),
    tovfd$ entry(fixed bin(15));

call tnoua('Command name is "',17);
call tnoua((comname),length(comname));
call tnoua('"',1);

if version=0 then
   do;  /* Version 0 means no more info. */
   call tnou('.',1);
   code=0;
   return;
   end;

if version=1 then;  /* Expected version number. */
else
   do;  /* New version, display it. */
   call tnoua(', version #',11);
   call tovfd$(version);
   end;  /* if version^=0 */
```

```
if vcb_ptr=null() then call tnou(', no CPL variables.',19);
else call tnou(', with CPL variables.',21);

call tnoua('Command line is "',17);
call tnoua((comline),length(comline));
call tnou('".',2);

strings=0;
last_string='';
line_to_show='Options: ';

if mod_after_date=0 then;
else call show_date('-MODIFIED_AFTER',mod_after_date);

if mod_before_date=0 then;
else call show_date('-MODIFIED_BEFORE',mod_before_date);

if bak_after_date=0 then;
else call show_date('-BACKEDUP_AFTER',bak_after_date);

if bak_before_date=0 then;
else call show_date('-BACKEDUP_BEFORE',bak_before_date);

if type_dir then call show_this('-DIR');
if type_segdir then call show_this('-SEGDIR');
if type_file then call show_this('-FILE');
if type_acat then call show_this('-ACAT');
if type_rbf then call show_this('-RBF');
if verify_sw then call show_this('-VERIFY');
if botup_sw then call show_this('-BOTUP');

if walk_from=2 then;  /* The default. */
else call show_value('-WALK_FROM',walk_from);

if walk_to=999 then;  /* The default. */
else call show_value('-WALK_TO',walk_to);

if in_iteration then call show_this('iteration');
if in_wildcard then call show_this('wildcard');
if in_treewalk then call show_this('treewalk');

/* Show last line if we have shown anything. */

if strings=0 then;
else
   if strings=1 then
      call tnou('Option: '||last_string,length(last_string)+8);
   else call show_this('');

code=0;
return;

show_date: proc(string,dtm);  /* Display option with date/time. */
```

```
dcl string char(32) var,
    dtm fixed bin(31);

dcl dow fixed bin(15),
    dtm_str char(21);

dcl cv$fda entry(bin(31),bin,char(21));

call cv$fda(dtm,dow,dtm_str);
call show_this(string||' '||trim(dtm_str,'11'b));

end;  /* show_date: proc */

show_value: proc(string,value);  /* Display option with integer. */

dcl string char(32) var,
    value fixed bin(15);

call show_this(string||' '||trim(char(value),'11'b));

end;  /* show_value: proc */

show_this: proc(string);  /* Display string in comma list. */

dcl string char(80) var;

dcl joiner char(6) var;

strings=strings+1;

if strings<=2 then joiner='';
else
   if string='' then
      if strings<=3 then joiner=' and ';
      else joiner=', and ';
   else joiner=', ';

if length(last_string)+length(line_to_show)+length(joiner)>79 then
   do;
   if strings<=3 then
      call tnou((line_to_show),length(line_to_show));
   else call tnou(line_to_show||',',length(line_to_show)+1);
   if string='' then line_to_show='and '||last_string;
   else line_to_show=last_string;
   end;
else
   line_to_show=line_to_show||joiner||last_string;

if string='' then call tnou((line_to_show),length(line_to_show));
else last_string=string;

end;  /* show_this: proc */

end;  /* com_proc_info: proc */
```

COMPLETE CALLING SEQUENCE

The complete calling sequence combines the command function calling sequence with the command processing information provided in the third argument of the calling sequence, as used in the detailed command calling sequence. In the command function calling sequence, described earlier, the third argument was ignored; in the detailed command calling sequence, as in the complete calling sequence, the third argument provides the program with information on the processing of the command that invoked the program.

Figure 18-7 illustrates the complete calling sequence, where EPF is the main entrypoint of the program EPF.

The first and second arguments are described in detail in the section entitled COMMAND CALLING SEQUENCE earlier in this chapter; the third argument is illustrated in Figure 18-6 and is described in the section entitled DETAILED COMMAND CALLING SEQUENCE earlier in this chapter; the fourth and fifth arguments are described in the section entitled COMMAND FUNCTION CALLING SEQUENCE. The remainder of this section explains why the complete calling sequence is useful and points out effects of combining a command and a command function in one program.

Why Use the Complete Calling Sequence?

A program that uses all five arguments in the complete calling sequence does so for one of several reasons:

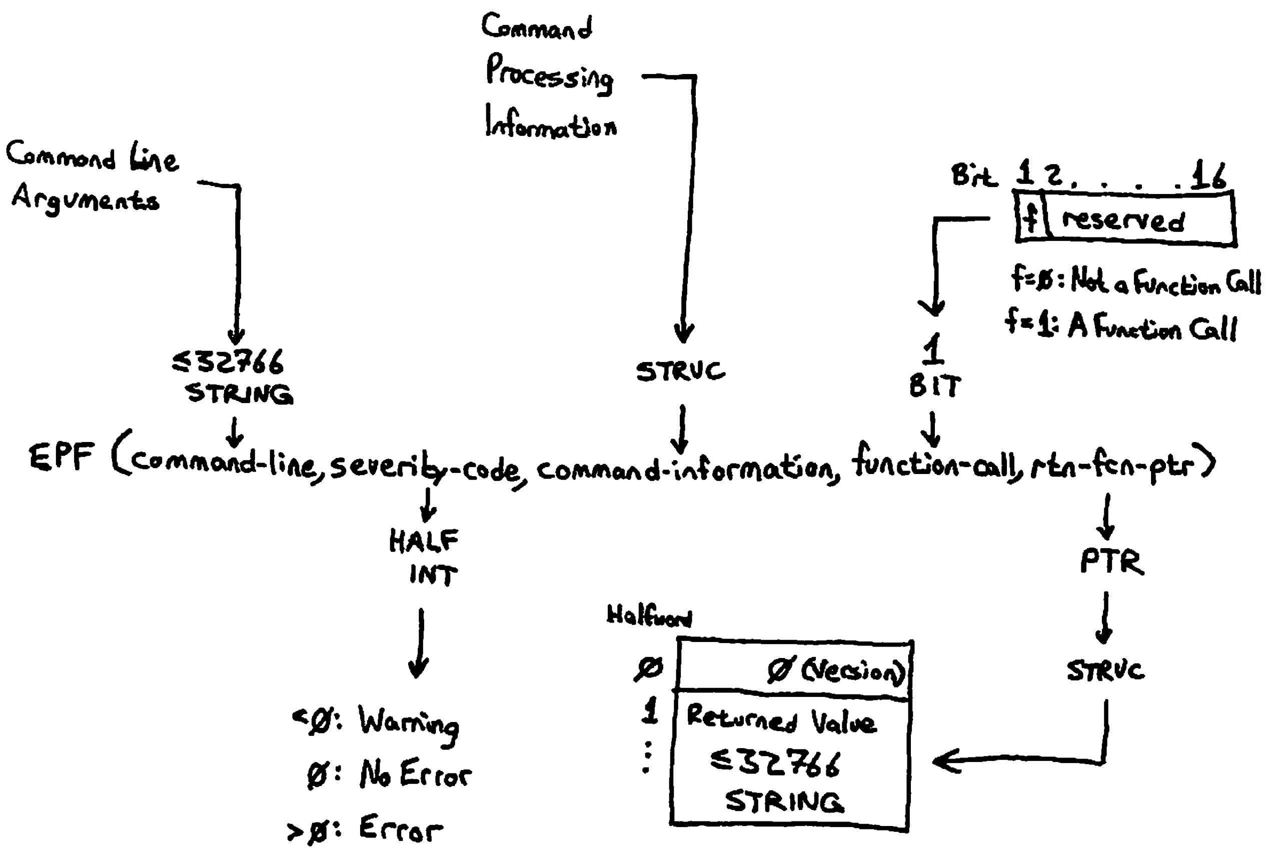
- It is a command function that needs access to CPL variables local to the CPL program that called it.
- It is a command function that needs access to its own command name.
- It is a program that may be invoked as a command function or as a command, and when invoked as a command, it wishes to make use of command preprocessing information.
- Any combination of the above three reasons, such as a program that, when invoked as a command, does not need command processing information, but when invoked as a command function, needs the CPL Local Variables pointer.

Each of these uses of the complete calling sequence is examined in more detail in the next section.

Command Function Needing Local CPL Variables

When a command function needs access to the CPL variables local to the CPL program that invoked the command function, it uses the LV$GET and

Complete Calling Sequence



Complete Calling Sequence
Figure 18-7

LV$SET subroutines to read and set the local CPL variables. An example of a command function that also sets local CPL variables is the [OPEN_FILE] function, described in the PRIMOS Commands Reference Guide and in the CPL User's Guide. Although not an EPF, this function could be written as an EPF as of Rev. 19.4, due to the program EPF interface described in this chapter.

Command Function Needing Command Name

Rarely, a command function may need access to its command name, if it wishes to make a distinction (or to enforce an equivalence) between the name of the program as built during the BIND session that linked the program and the name of the program as invoked by the user. For example, when such a program issues messages, it may wish to use its invocation name, rather than its original name, so that its name may be easily changed without making error messages originating from the program more difficult to track down.

Program Usable as a Command and as a Command Function

A program may need to be usable as both a command and as a command function. In addition, it may need access to command processing information when invoked as a command, as a command function, or in both cases.

For example, a program may, when invoked as a command, wish to use command preprocessing information to generate useful output, depending upon whether it was invoked using a wildcard, treewalking, or iteration specification. The same program, when invoked as a command function, does not need that information.

It is important to understand that the PRIMOS command processor does not perform any type of command iteration (including wildcarding, treewalking, and explicit iteration) when it is called upon to invoke a program as a command function.

Therefore, a program invoked as a command function should not expect the command preprocessing information in halfword offsets 21-25 (25-31 octal) in the command processing information structure to contain any usable information.

The PRIMOS command processor knows that a command is being invoked as a command function because its entrypoint, CP$, has a command-function bit as one of its input arguments. When set, CP$ does not perform any command iteration on the command line; instead, it passes the untouched command line directly through to the program EPF. (Other command preprocessing is performed as usual.)

However, a user-written command processor, other than CP$, may invoke a program EPF as a command function, providing useful information in halfword offsets 21-25 in the command processing information structure by passing it to EPF$INVK or EPF$RUN. If your program EPF is designed to be invoked only by such an application, it may use the command preprocessing iteration information even when invoked as a command function. This situation is expected to be quite rare.

CHAPTER 19

Invoking Programs From Within Programs

A program or library may invoke another command, program, or function. PRIMOS provides three methods of invoking a program EPF, whether or not it is a function:

- Via the CP$ subroutine, which invokes the PRIMOS command processor
- Via the EPF$RUN subroutine, which invokes any program EPF
- Via the EPF$INVK subroutine, which invokes a program EPF that is already mapped to memory, allocated, and initialized

You may also use the CP$ subroutine to invoke a command, a program, a function, a CPL program, a CPL function, or a static-mode program.

This chapter describes how to use these subroutines to invoke commands, programs, and functions. This chapter also describes how to free the memory used to store the result of a command function (the FRE$RA subroutine). Finally, this chapter explains particular items of interest when invoking other commands, programs, or functions.

Commands, Programs, and Functions

There are several ways to categorize, or group, commands and programs under PRIMOS. For example, one may consider Prime-supplied commands and programs as distinct from user-supplied commands and programs. However, the PRIMOS command processor provides a uniform interface to all commands and programs so that the category into which a particular command or program fits is usually not an important consideration.

Because of the flexibility of the PRIMOS command processor, systems may add their own commands. Therefore, categorizing commands and programs by whether they are Prime-supplied is not particularly useful when writing programs that invoke them.

In fact, there are three ways of categorizing commands and programs that are most useful:

- Where the programming instructions for the command or program reside
- In which format the programming instructions for the command or program are stored
- Whether the command or program is invoked as a function

In most cases, the PRIMOS command processor allows you to issue commands and run programs independent of their categorization. The interfaces described in this chapter, CP$, EPF$RUN, EPF$INVK, and FRE$RA pertain to different categories of commands and programs:

- CP$ can invoke any command or program, optionally as a function.
- EPF$RUN and EPF$INVK can invoke only a program EPF, optionally as a function.
- FRE$RA is used only when invoking functions, after the function has returned its value; it is used independently of the function location or format.

The categories of commands and programs are described in more detail next. As you will see, functions are commands or programs that have additional functionality.

Where the Programming Instructions Reside: The location of the programming instructions for a command or program is one of the following:

- Internal to the PRIMOS Operating System
- On disk, in the CMDNC0 UFD
- On disk, but not in the CMDNC0 UFD

The first two places are where commands are stored; the latter place is where programs are stored. A command residing in the CMDNC0 UFD is just a program in a special place, and it may be run as a program; a program not residing in the CMDNC0 UFD may be made into a command simply by copying it into CMDNC0. Therefore, the distinction between commands and programs on disk is somewhat hazy; the terms "command"

and "program" are often interchangable, and are often used together in this guide. Some, but not necessarily all, commands and programs are supplied by Prime.

Internal to PRIMOS are internal commands. These are all Prime-supplied; Prime does not support the modification of PRIMOS by customers, such as to add new internal commands. Because internal commands reside in virtual memory rather than on disk, they are treated specially by the PRIMOS command processor. In fact, some internal commands have special privileges, such as the ability to access internal PRIMOS tables.

While user-written programs cannot always perform the same functions as internal PRIMOS commands, such programs can call the PRIMOS command processor to invoke internal PRIMOS commands.

A special internal PRIMOS command is the RESUME command, abbreviated R. The RESUME command is used to run a program. Special processing is performed by the command processor to treat a RESUME command as the invocation of a program rather than the invocation of an internal PRIMOS command, although this special processing is not usually important except when handling errors and such.

Format of the Programming Instructions: The format of the programming instructions for a command or program is important to the PRIMOS command processor, because it determines how the command processor invokes the command or program. For commands and programs that reside on disk, there are three formats:

- Executable Program Format (EPF) Runfiles
- Command Procedure Language (CPL) Programs
- Static-mode Runfiles

(A fourth format, the SEG runfile, is not recognized by the PRIMOS command processor — it is recognized only by the SEG command, which itself is a static-mode runfile residing in the CMDNC0 UFD.)

Whether the PRIMOS command processor is called upon to execute a command in the CMDNC0 UFD or elsewhere on disk, it uses suffix searching to scan for the appropriate runfile. The suffixes .RUN, .SAVE, and .CPL are tried, in that order, and then a search with no suffix is tried. Based on the suffix that was in place when the runfile was found, the command processor infers the format of the runfile, as described in Chapter 16.

The most flexible format for programming instructions is the EPF, because a program written as a program EPF may be a function and in fact can determine whether it is being invoked as a function and modify its actions accordingly. In addition, a program EPF can modify CPL variables local to the CPL program that invoked it. Finally, a program EPF has the most control over selecting command processing features and determining which features are in use for a particular invocation.

The second most flexible format is the CPL program. A CPL program can be written either as a program or as a function. It can also choose how it will handle wildcards, as wildcards are not processed for CPL programs.

The least flexible format is the static-mode program. A static-mode program cannot be written as a function. The only control a static-mode program has over command processing features is by having its name begin with NX$ or NW$ to disable various combinations of such features; this requires users to enter the NX$ or NW$ prefix when entering the program name, however.

For commands internal to PRIMOS, there is only one format, and that is the format of a subroutine, or procedure, that accepts a standardized calling sequence as its arguments.

Functions: A function returns a value to the invoker of the function. This value typically replaces the invocation of the function in a CPL program command line, for example. The difference between a program that is a function and one that is not is whether the program is designed to operate as a function and whether the invoker of the program is invoking it as a function.

For example, the ABBREV -STATUS command, when used as a command, does not operate as a function — it displays the pathname of the user's abbreviation file, and the number of abbreviations defined in the file:

```
    OK, ABBREV -STATUS
    Abbreviation file: UNGER>LOGIN.ABBREVS
    Abbreviations: 183

    OK,
```

When used as a function, however, the ABBREV -STATUS modifies its behavior to display nothing to the terminal and to instead return the pathname of the user's abbreviation file as the value of its invocation:

```
    OK, TYPE Your abbreviation file is: [ABBREV -STATUS]
    Your abbreviation file is: UNGER>LOGIN.ABBREVS
    OK,
```

The displayed output came not from the ABBREV -STATUS invocation, but from the TYPE command.

The ABBREV -STATUS command is an example of a command that operates as either a command or as a function, depending on how it is used.

Typically, however, a command or program always operates as one or the other. For example, another internal command, RDY, operates only as a command — when invoked as a function, it still behaves as a command and returns no value:

```
OK, TYPE Value of RDY command is: [RDY]
OK 14:33:39  243.024  11.354
Value of RDY command is:
OK,
```

The first line of displayed output came from the invocation of the RDY command. The second line of output came from the invocation of the TYPE command, which included a function invocation of RDY that returned no result because RDY is not a function.

Conversely, a command or program may be constructed to run only as a function. For example, when invoked as a command, the internal command SUBSTR produces the following message:

```
OK, SUBSTR TEST 2 2
May only be invoked as a command function. (SUBSTR)
ER!
```

Here, the SUBSTR command detected that it was not invoked as a function, displayed an error message, and returned a positive severity code (producing the ER! prompt).

Almost all Prime-supplied functions are commands, either internal to PRIMOS or residing in CMDNC0. Functions that are commands are often called command functions. Prior to Rev. 19.4, users could write functions only in CPL; as of Rev. 19.4, they may write functions as program EPFs. Although the term program function can be used to refer to a function not supplied by Prime, the distinction is not usually important for readers of this guide; therefore, the terms function and command function are used generically to refer to any command or program that returns a function value when invoked as a function.

Any type of command or program may be written as a function except for a static-mode program. A restriction for CPL programs is that they cannot determine whether they are being invoked as functions and modify their behavior accordingly; they must either always assume they are being invoked as a function or as a program, or they must accept a command line option that is supplied by the invoker to indicate which kind of invocation is taking place. Program EPFs can determine which form of invocation is being used, as described in Chapter 18.

Deciding Which Interface to Use

To write a program, library, or subroutine that invokes another command, program, or function, you must first decide which interface to use:

- CP$
- EPF$RUN
- EPF$INVK
- FRE$RA

You make your decision based on what kind of program you wish to invoke, and whether you wish to use command preprocessing features such as variable expansion, wildcarding, and name generation.

In summary:

- Use CP$ to invoke a PRIMOS command or a program, or to include command preprocessing features.
- Use EPF$RUN to invoke a program EPF.
- Use EPF$INVK to invoke a program EPF with more control over how and when the EPF is set up.
- Use FRE$RA only if you invoke a function and accept a returned text string.

Typically, you choose only one of the CP$, EPF$RUN, and EPF$INVK subroutines; these allow your program to invoke either a program or a function. After calling a function, your program makes use of the returned text string. Your program then calls the FRE$RA subroutine to free the memory used to store the returned text string, allowing the memory to be reused.

When to Use CP$

You use the CP$ subroutine to invoke:

- Internal PRIMOS commands, such as ASSIGN
- External CPL programs
- External EPFs
- External static-mode programs

Except for external static-mode programs, any of the above may be invoked as functions.

Calling CP$ invokes the PRIMOS command processor, STD$CP. This same command processor is invoked when the user enters a response to the OK, prompt issued by PRIMOS.

User-defined abbreviations are not expanded by CP$. Therefore, you can reliably use CP$ in your program without concerning yourself with user-defined abbreviations that might change the meaning of your command lines. For example, calling CP$ to invoke the ASSIGN MT0 command always invokes that command, even if the user has defined ASSIGN or MT0 as an abbreviation via the PRIMOS abbreviation facility.

The PRIMOS command processor, invoked via CP$, determines what command is being executed as follows:

1. The first token of the command line is parsed. This is the name of the command being invoked. For example, consider the command line:

   ```
   COPY FRED>MEMO.12/31/84 *>MEMOS>MEMO.118
   ```

   Here, the name of the command is COPY.

2. The command name is checked against the list of internal PRIMOS commands. One important internal PRIMOS command is RESUME; if the command is RESUME, the program specified by the pathname following the RESUME command is invoked.

   If the command name is not RESUME, and is found in the list of internal PRIMOS commands, the appropriate command line preprocessing (such as wildcarding) is performed, and the internal PRIMOS subroutine that corresponds to the command name is invoked. The command processor returns to the caller when the internal PRIMOS subroutine has finished.

3. If the command name is not in the list of internal PRIMOS commands, the command processor searches the CMDNC0 directory for a program with the same name as the command. If found, the program is executed as if it had been RESUMEd.

When executing a program, the command processor first performs the appropriate command preprocessing (such as wildcarding), depending upon the program type. If the program is an EPF, the command preprocessing is determined by information within the EPF itself, as built using BIND subcommands. For information on BIND subcommands that describe the command preprocessing environment for an EPF, see Chapter 17. See the PRIMOS Commands Reference Guide for information on command preprocessing for static-mode and CPL programs.

Although command programs reside only in the CMDNC0 directory, CP$ can be used to invoke programs residing anywhere on disk by invoking the internal command RESUME via CP$. For example, to invoke the program ACCOUNTS_PAYABLE in the current directory, call CP$ with the following command line:

```
    RESUME ACCOUNTS_PAYABLE
```

When to Use EPF$RUN

You use EPF$RUN to invoke a program EPF. As with CP$, you pass the command line to the target program, but no command preprocessing is performed on the command line. Therefore, use EPF$RUN when you do not want any changes to be made to the command line being passed.

EPF$RUN handles all of the tasks needed to execute a program EPF, including mapping the EPF to memory, allocating the linkage area, initializing the linkage area, and optionally removing the EPF from memory when the invocation has been completed.

When to Use EPF$INVK

You use EPF$RUN to invoke a program EPF that has already been mapped to memory, allocated and initialized. As with CP$, you pass the command line to the target program, but no command preprocessing is performed on the command line. Therefore, use EPF$INVK when you do not want any changes to be made to the command line being passed.

The advantage of using EPF$INVK over EPF$RUN is that you have more control over the phases of EPF execution. However, you must call several other subroutines, described in this chapter, to map the EPF to memory, allocate the linkage area, initialize the linkage area, and after invocation to remove the EPF from memory.

When to Use FRE$RA

You use the FRE$RA subroutine after using CP$, EPF$RUN, or EPF$INVK to invoke a function only if the returned function pointer is not a null pointer (segment number 7777). Your program should call FRE$RA sometime after it finishes using the returned function value; this may be after it makes its own copy of the value, or after it finishes analyzing the value. If you have used EPF$INVK to invoke the function, it is not important whether your program calls FRE$RA before or after calling EPF$DEL to remove the EPF.

THE CP$ SUBROUTINE

There are two ways of using CP$:

- Invoking commands or programs
- Invoking functions

The calling sequence for CP$ has six arguments. When not invoking a function, you may wish to pass only three arguments; the remaining three arguments are assigned default values before being passed to the PRIMOS command processor, STD$CP.

Figure 19-1 illustrates the calling sequence for CP$. The next two sections describe how to use CP$ to invoke a command, program, or function.

Using CP$ to Invoke a Command or Program

To use the CP$ subroutine to invoke an internal PRIMOS command or a program, rather than a function, you typically need to supply only the first three arguments — command-line, code, and severity-code — of the calling sequence illustrated in Figure 19-1. If you wish to pass a pointer to local CPL variables, then you must supply five or six arguments in the calling sequence to include the cpl-local-vars-ptr argument.

Before calling CP$, your program should initialize the severity-code argument to 0, in case it is not set by the command or program being invoked.

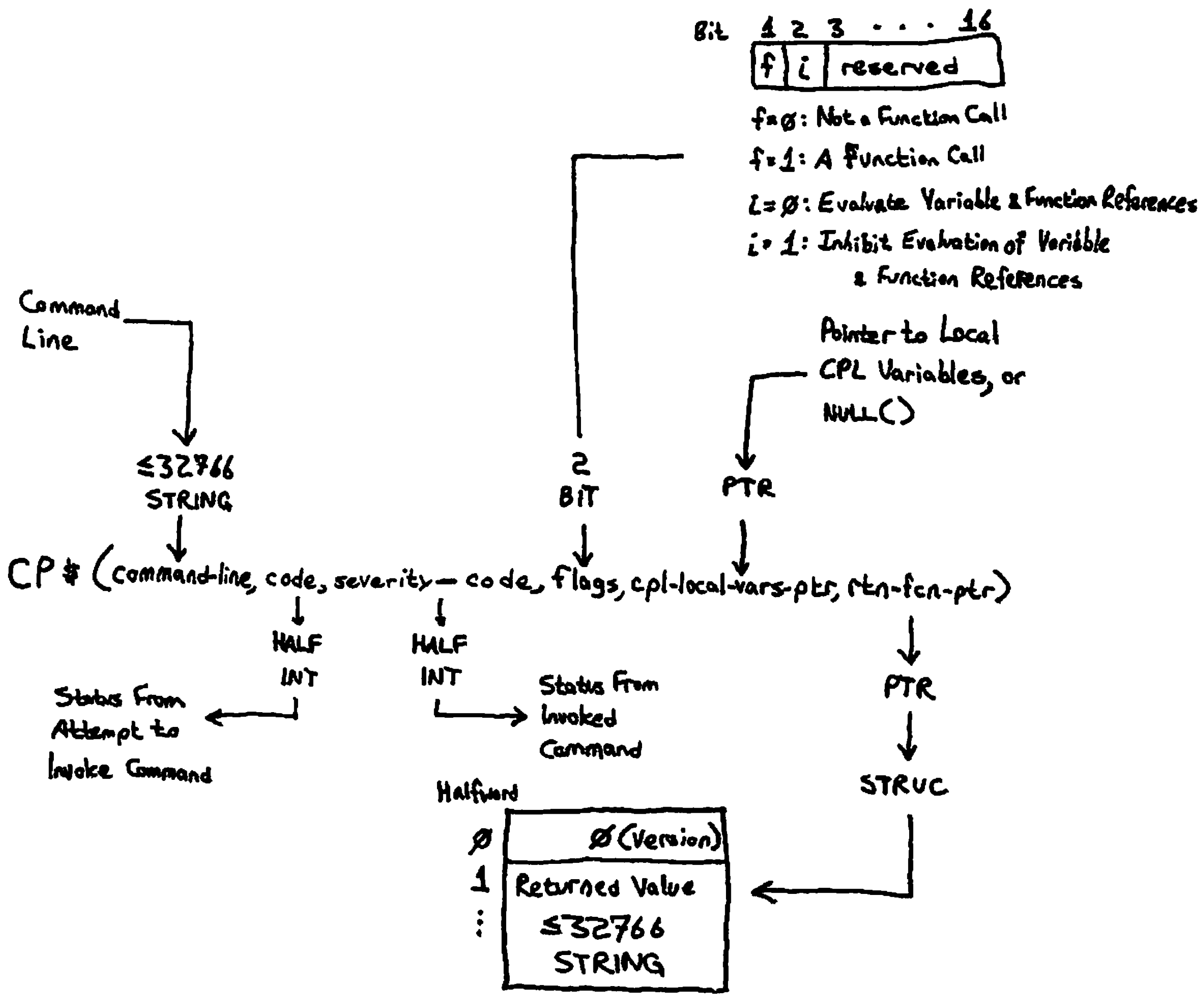
When your program calls CP$, the command processor attempts to execute the command passed in command-line. If it fails to begin execution, a standard PRIMOS error code is returned in code. If it succeeds in executing the command, 0 is returned in code, and the status of the command itself is returned in severity-code.

Ultimately, when the program you invoke via a call to CP$ is a program EPF, the severity-code argument to CP$ corresponds to and is set from the severity-code argument in the calling sequence for a program EPF, described in Chapter 18; CPL programs set this value by issuing a &RETURN directive, and static-mode programs set this value by calling the SETRC$ subroutine.

Note

The returned value of severity-code is undefined if the returned value of code is nonzero.

Invoke a Command, Program, or Function


Bit 1 2 3 . . . 16
f | i | reserved
f=Ø: Not a Function Call
f=1: A Function Call
i=Ø: Evaluate Variable & Function References
i=1: Inhibit Evaluation of Varible & Function References
Command Line
≤32766 STRING
2 BIT
Pointer to Local CPL Variables, or NULL()
PTR
CP$ (command-line, code, severity-code, flags, cpl-local-vars-ptr, rtn-fcn-ptr)
HALF INT
HALF INT
Status From Attempt to Invoke Command
Status From Invoked Command
PTR
STRUC
Halfword
Ø
Ø (Version)
1
Returned Value ≤32766 STRING

Calling Sequence of CP$
Figure 19-1

The Command Line: In command-line, simply pass the command line that you would type as a user invoking the command. The PRIMOS Commands Reference Guide contains information on command formats. For example, to assign a magnetic tape drive for use by a running program, you might have your program call CP$ with the command line:

```
ASSIGN MT0 -WAIT
```

The RESUME command is a special case, because it is an internal command that runs an external program. Use the RESUME command to invoke a program via CP$. For example, to run a program, you might have your program call CP$ with the command line:

```
RESUME MYPROG MEMO.03/08/05
```

Unless you place a tilde in front of the command line, CP$ performs certain kinds of command preprocessing on command-line before actually invoking the internal command (although it never modifies command-line itself). First, if the command line contains one or more unquoted command separator characters (;), CP$ splits up the command line into several separately handled command lines.

Then, unless inhibited by the second bit of flags, CP$ resolves command function references and variable references. Subsequent command preprocessing depends on the command or program being invoked; for example, ATTACH does not accept wildcards, but LIST_QUOTA does. See the PRIMOS Commands Reference Guide for information on command preprocessing support by Prime commands; use the LIST_EPF -COMMAND_PROCESSING command to determine what kind of command preprocessing is performed for a particular program EPF being invoked.

Note

Placing a tilde (~) in front of the command line as passed to CP$ has the effect of preventing all forms of command preprocessing. Therefore, calling CP$ with the command line

```
~SET_VAR .FOO %OPTION% is an option; [SET_1] is a function.
```

causes the global variable .FOO to be set to exactly the string shown. Without the tilde (~), the variable %OPTION% and command function reference [SET_1] would be evaluated, and the results substituted in the command line (assuming the variable and function references succeeded). In addition, the semicolon after "option" would be treated as a command separator.

The Error Code: The code argument, returned by CP$, indicates the degree of success encountered by the command processor's attempt to execute the command. For example, if the command is not found, the error code e$fntf (Not found) is returned in code.

Any nonzero value returned in code indicates that all other output arguments have undefined values, because they all depend upon the successful invocation of the command.

See the section entitled Error Codes From CP$, later in this chapter, for a partial list of error codes.

The Severity Code: The severity-code argument, returned by the invoked command via the command processor and CP$, indicate the degree of success reported by the invoked command. For example, if you invoke the ATTACH command to attach to a nonexistant subdirectory, the error code e$fntf (Not found) is returned in code.

Note

> The RESUME command is handled by the command processor in a special way. The target of the RESUME command is the program to be invoked. If the target program is not found, the error code is returned in code, not severity-code as for other commands (such as ATTACH, COPY, and so on). This allows the calling program to distinguish between a missing program and a program that cannot find the target specified on its command line.

The Function-Call Bit: The first bit of the flags argument specifies whether the call to CP$ is to invoke a function (such as GVPATH or a user-written function) or not. If flags is not supplied in the calling sequence, the function-call bit defaults to 0, meaning that a function invocation is not being made. If flags is supplied, set this bit to 0 to indicate that you are invoking a command or program rather than a function. (The use of CP$ to invoke a function is described in the next section.)

The Inhibit-Evaluation Bit: The second bit of the flags argument specifies whether command function references and variable references in the command line are to be evaluated. If flags is not supplied in the calling sequence, the inhibit-evaluation bit defaults to 0, meaning that such references are to be evaluated. If flags is supplied, set this bit to 0 if you wish such references to be evaluated, or set this bit to 1 if you wish such references to not be evaluated and instead passed to the target program.

The CPL Local Variables Pointer: The cpl-local-vars-ptr argument provides the necessary "toehold" for the target command or program to set CPL variables local to the procedure that invoked your program. Typically, you either do not supply this argument or you supply the null pointer (NULL(), which is segment 7777 offset 0). If you do not pass this argument, CP$ substitutes the null pointer when calling the PRIMOS command processor, STD$CP.

If your program may be invoked by a CPL program, and if it is using CP$ to invoke a program that may need to set one or more CPL variables local to the invoking CPL program, then your program should pass in cpl-local-vars-ptr the corresponding pointer passed to its main entrypoint in the command-information structure of the program EPF calling sequence. (See Chapter 18 for more information on the command-information structure.)

The Returned Function Value Pointer: The rtn-fcn-ptr argument is not used when invoking a command or program. It is used only when invoking a function, that is, when bit 1 of the flags argument is set to 1, as described in the next section.

Using CP$ to Invoke a Function

The CP$ subroutine may be used to invoke a command function that is either an internal PRIMOS command function, such as DATE and GVPATH, or a user-written command function, written in CPL or as a program EPF. Whether the command function being invoked is a Prime-supplied command function or a user-written command function, your program calls CP$ in the same way.

To use the CP$ subroutine to invoke a function, have your program pass all six arguments to CP$ as illustrated in Figure 19-1 earlier in this chapter.

Before calling CP$, your program should initialize the severity-code argument to 0 and the rtn-fcn-ptr to the null pointer (NULL() in PL1/G), in case these arguments are not set by the function being invoked.

When your program calls CP$, the command processor attempts to execute the command passed in command-line. If it fails to begin execution, a standard PRIMOS error code is returned in code. If it succeeds in executing the command, 0 is returned in code, the status of the command itself is returned in severity-code, and a pointer to the returned text string structure is returned in rtn-fcn-ptr.

Ultimately, when the program you invoke via a call to CP$ is a program EPF, the rtn-fcn-ptr argument to CP$ corresponds to the rtn-fcn-ptr argument in the calling sequence for a program EPF, described in Chapter 18; CPL programs set this value by issuing a &RESULT directive.

Notes

1: The returned values of severity-code and rtn-fcn-ptr are undefined if the returned value of code is nonzero.

2: When invoking a command function, no wildcarding, iteration, or treewalking is performed. In addition, the command separator character, the semicolon (;) is not honored — it is treated as any other character.

The Command Line: In command-line, use the RESUME command, or the command name itself, just as you would when invoking a command or program. Do not enclose the command line in square brackets ([ ]) as you would in a CPL program.

For example, to determine the user's abbreviation file, call CP$ with the command line:

```
ABBREV -STATUS
```

The pathname of the abbreviation file, -OFF, or both is returned in the structure pointed to by rtn-fcn-ptr, as described below.

To invoke a user-written command function, you might have your program call CP$ with the following command line:

```
RESUME PROGRAMS>GET_RECORD 1154 -DATABASE PAYROLL
```

Again, the information is returned in a structure pointed to by rtn-fcn-ptr.

Unless you place a tilde in front of the command line or set the second bit of flags to 1, CP$ resolves (nested) command function references and variable references.

The Error Code: The code argument, returned by CP$, has the same meaning for function invocation as for command or program invocation, described earlier in this chapter.

The Severity Code: The severity-code argument, returned by the invoked function via the command processor and CP$, has the same meaning for function invocation as for command or program invocation, described earlier in this chapter.

The Function-Call Bit: The first bit of the flags argument specifies whether the call to CP$ is to invoke a function (such as GVPATH or a user-written function) or not. Set this bit to 1 to indicate a function invocation.

The Inhibit-Evaluation Bit: The second bit of the flags argument has the same meaning for function invocation as for command or program invocation, as described earlier in this chapter.

The CPL Local Variables Pointer: The cpl-local-vars-ptr argument has the same meaning for function invocation as for command or program invocation, as described earlier in this chapter.

The Returned Function Value Pointer: The rtn-fcn-ptr argument contains a pointer to the returned function value when CP$ returns, or the null pointer if no function value has been returned. Actually, rtn-fcn-ptr points to a structure that contains the returned value, as illustrated in Figure 19-1.

Note

> If the invoked command did not return a value, then rtn-fcn-ptr may not have modified. Therefore, set it to the null pointer before calling CP$, and check it after CP$ returns to make certain that a result has been returned.

In PL/1, the declaration of the returned function value structure is:

```
dcl 1 rtn_function_structure based(rtn-fcn-ptr),
      2 version fixed bin(15),
      2 text_string char(32766) var;
```

If version does not contain 0, do not attempt to use text_string, because a nonzero version indicates a new version of the returned structure. However, version should contain 0, and text_string should contain the returned text string.

After using the returned text string, your program should free the returned text string structure to the pool of available memory. Use the FRE$RA subroutine to do this. FRE$RA is described later in this chapter.

If your program is written in FORTRAN, access to the returned function value is difficult. Here is a programming discipline that allows a FORTRAN program to copy the returned function value, pointed to by an INTEGER*4 pointer variable named RFNPTR, into an INTEGER*2 array of characters named RTNFCN and a length variable named RTNLEN. The maximum number of characters that can be held by RTNFCN is set in a parameter named RTNMAX.

```
      INTEGER*2 GCHAR,IXS,IXD,RTNFCN(512),RTNLEN,RTNMAX
      INTEGER*4 RFCPTR
C
      PARAMETER RTNMAX=1024
C ...
C ... CALL CP$ HERE, check error code
C ...
C
C Check if the returned pointer is the null pointer.
C
      IF (AND(RFCPTR,:177600000).NE.:17760000) GO TO 98710
C
C Null pointer, assume zero-length result.
C
      RTNLEN=0
      GO TO 98800  /* Do not call FRE$RA with a null pointer!
C
C Have a pointer, see if version 0.
C
98710 IXS=0  /* Source string index.
      IF (GCHAR(RFNPTR,IXS)+GCHAR(RFNPTR,IXS).EQ.0) GO TO 98720
C
C Not version 0, unknown version, assume null value.
C
      RTNLEN=0
      GO TO 98790  /* Do call FRE$RA to deallocate the structure.
C
C Get length of returned function value in RTNLEN.
C
98720 RTNLEN=LS(GCHAR(RFNPTR,IXS),8)+GCHAR(RFNPTR,IXS)
C
C Now, IXS should be 4 which is the beginning of the value itself.
C Copy the value into RTNFCN until the end of the source or the end
C of the destination is reached.
C
      IF (RTNLEN.EQ.0) GO TO 98790  /* Null value!
C
      IXD=0  /* Destination string index.
C
C Loop until string copied.
C
98730 CALL SCHAR(LOC(RTNFCN),IXD,GCHAR(RFNPTR,IXS))
      IF (IXS.LT.RTNLEN.AND.IXD.LT.RTNMAX) GO TO 98730
C
C Now free the structure.
C
98790 CALL FRE$RA(RFNPTR)
C
C Done!
C
98800 CONTINUE
```

Error Codes From CP$

An output argument, code, informs the calling program of the success or failure of the operation. This argument is a HALF INT value. Symbols are provided to allow PL1/G, FORTRAN, Pascal, and PMA programs to substitute mnemonic keywords for numeric values.

If code is 0, the operation was entirely successful. Otherwise, code has one of many values. Typical values and their meanings follow. Not all possible error codes are listed; for example, PRIMENET-related error codes such as E$RLDN (The remote line is down) may be returned by CP$, but are not listed.

Note

When you use CP$ to invoke a program EPF, either via the RESUME command or by specifying a program EPF in CMDNC0, an error code returned by the EPF$RUN subroutine is returned by CP$. Therefore, consult the list of error codes returned by EPF$RUN, later in this chapter, for information on additional error codes returnable by CP$.

Keyword	Value	Meaning
<ok>	0	The operation was successful.
E$EOF	1	End of file. Typically, this error indicates an attempt to invoke a text file (such as a CPL file) as a static-mode program. Alternatively, this error indicates a file that has been truncated by FIX_DISK during system maintenance procedures. In the latter case, you must replace the program with a backup copy.
E$FIUS	3	File in use. Indicates an attempt to run a program that is open for writing.
E$NRIT	10	Insufficient access rights. You do not have access to the program.
E$DIRE	14	Operation illegal on directory. Typically, this error indicates an attempt to invoke a segment directory, such as a .SEG file, with the RESUME command. Alternatively, this error indicates an attempt to invoke a file directory.
E$FNTF	15	Not found. If the command is the RESUME command, the target program could not be found. Otherwise, the command is not an internal command, and a program with the

		same name could not be found in CMDNC0.
E$BNAM	17	Illegal name. The RESUME command specifies a filename not conforming to filename syntax rules.
E$ITRE	57	Illegal treename. The RESUME command specifies a pathname not conforming to pathname syntax rules.
E$CMND	68	Bad command format. The command name, the first token on the command line, is more than 32 characters long or does not conform to filename syntax rules.
E$BARG	71	Invalid argument to command. The RESUME command is not followed by a program name.
E$NDAM	109	Not a DAM file. The target program is a .RUN file, indicating an EPF, but is not a DAM file. The fault is in the installation of the program being invoked.
E$BVER	158	Incorrect version number. Typically, this error means that the command function invoked by the call to CP$ returned a structure containing an invalid version number. Alternatively, this error means that the target EPF contains an invalid version number. In both cases, the fault is in the command function, not the calling program. The command function is an EPF, because a CPL program should never cause this error. If the command function is in fact a CPL program, contact your Customer Support Center.
E$NINF	159	No information. You do not have access to the program.

*** THERE SHOULD BE ONE OR TWO ERROR CODES INDICATING THAT A FUNCTION REFERENCE OR VARIABLE REFERENCE IN THE COMMAND LINE WAS INVALID. CURRENTLY RETURNS "REMOTE LINE DOWN" OR "BAD STARTUP" CODES FOR BAD VARIABLE REFERENCES, WHICH COME FROM CPL ERROR CODES. STD$CP SHOULD TRANSLATE ANY ERRORS FROM EVAL_A INTO A PARTICULAR ERROR CODE. ***

THE EPF$RUN SUBROUTINE

The EPF$RUN subroutine is used in the following manner:

1. The calling program opens the program EPF file to be invoked.

2. The calling program calls EPF$RUN, passing the file unit number of the opened program EPF file.

3. The calling program closes the program EPF.

4. After the EPF$RUN subroutine completes, the calling program checks the returned error code to determine whether the program EPF was successfully invoked by EPF$RUN.

5. If the error code from EPF$RUN is 0, the calling program uses the information returned by EPF$RUN to determine whether the program EPF completed successfully or unsuccessfully, and optionally to access the returned text string (if the program EPF was invoked as a command function).

6. If the error code from EPF$RUN is 0, and the calling program invoked the program EPF as a command function, the calling program uses the FRE$RA subroutine to return the memory used to store the returned text string to the free memory pool.

These steps are described in detail below. Following the steps, a listing of error codes that may be returned by EPF$RUN is presented.

Step 1: Open the EPF File

Your program must first open the target program EPF file for VMFA-read before calling EPF$RUN. VMFA stands for Virtual Memory File Access, a mechanism that provides efficient data retrieval from disk storage by mapping disk records into memory via the virtual memory mechanism. PRIMOS implements a limited form of VMFA called read-only VMFA, and supports this mechanism for use only by the EPF mechanism.

To open the target program EPF file for VMFA-read, use the k$vmr key when you invoke the SRCH$$, TSRC$$, or SRSFX$ subroutines. For example, a PL1/G program might use the following call:

```
call srsfx$(k$vmr+k$getu,'MY_EPF',unit,type,1,'.RUN',basename,i,
     code);
```

A FORTRAN program might use the following statement:

```
CALL SRCH$$(K$VMR+K$GETU,'MY_EPF.RUN',10,UNIT,TYPE,CODE)
```

Typically, you add k$getu to the k$vmr key, to specify that a free file unit is to be found by PRIMOS. If you do, the file unit number used is returned in unit. If you do not add k$getu, you must pass a valid file unit number in unit.

If code is 0 when SRSFX$, TSRC$$, or SRCH$$ returns, the file is open on the indicated file unit. Otherwise, the file is not open, and code contains an error code indicating the problem. If an error occurred, EPF$RUN cannot be called to invoke the EPF, because it is not open.

See the Subroutines Reference Guide for details on the SRSFX$, TSRC$$, and SRCH$$ subroutines.

Step 2: Invoke EPF$RUN

After your program has opened the target program EPF file, it calls EPF$RUN. Figure 19-2 illustrates the calling sequence for the EPF$RUN subroutine.

Although the calling sequence contains eight arguments, there are two cases in which only the first three arguments need be passed. The other five arguments are not used by EPF$RUN or by EPF$INVK (which EPF$RUN calls to invoke the EPF) — they are simply passed to the main entrypoint of the program EPF, corresponding to the five arguments in the complete calling sequence of a program EPF as described in Chapter 18. The two cases in which only the first three arguments to EPF$RUN need be passed are:

- When the k$restore_only value for key is used, in which case the target EPF is not actually invoked
- When the main entrypoint of the target EPF is known to accept no arguments

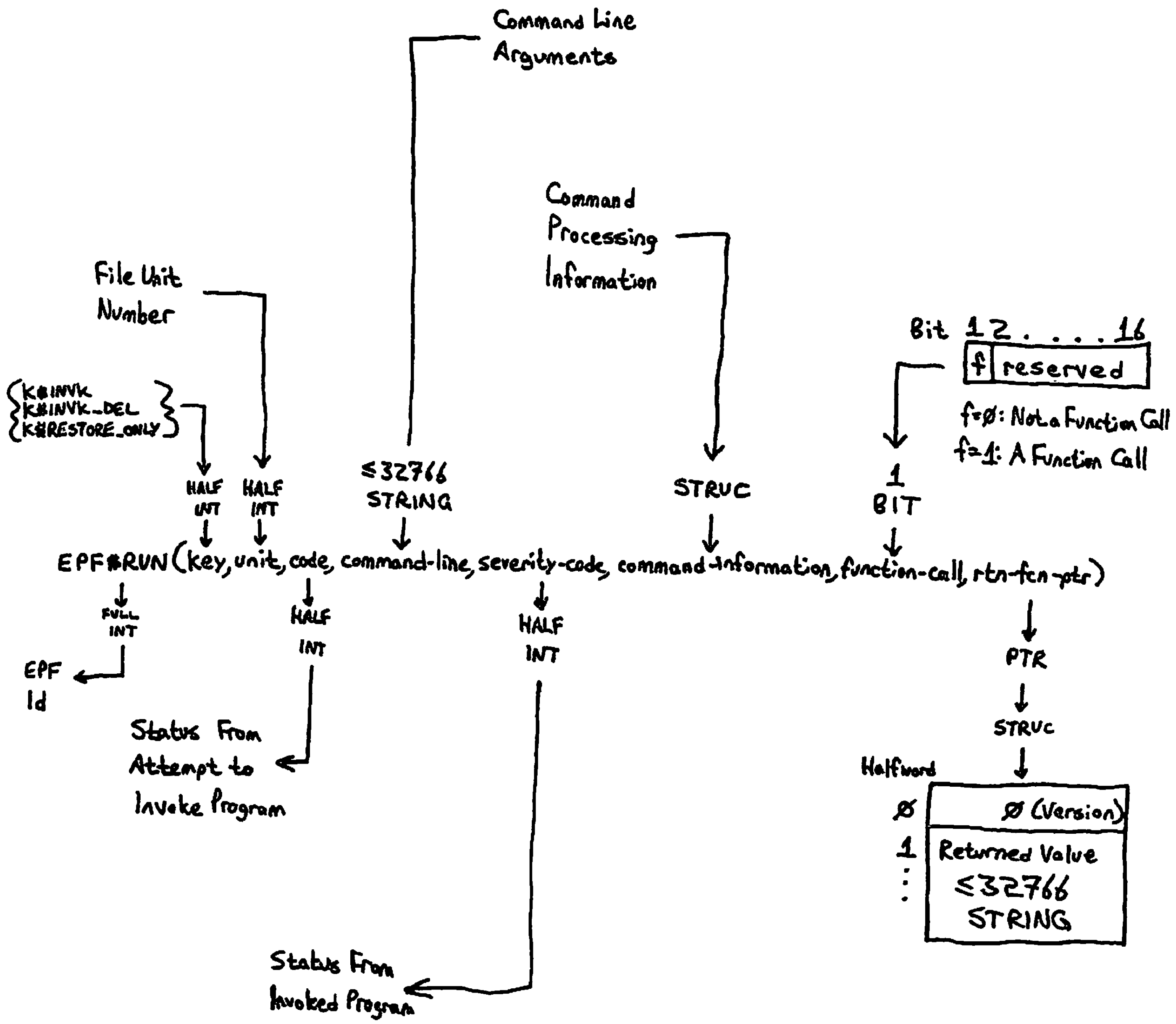
The arguments for the EPF$RUN subroutine are described below.

The Key: For key, specify k$invk, k$invk_del, or k$restore_only. Both k$invk and k$invk_del cause the target EPF to be invoked; however, k$invk causes the program EPF to be left in the EPF cache after it completes, whereas k$invk_del causes the program EPF to be removed from the EPF cache after it completes.

The k$restore_only key causes all activities up to, but not including, the invocation of the program EPF to be performed; use the EPF$INVK and EPF$DEL subroutines, described later in this chapter, to complete the process of executing a program EPF.

The EPF cache is a mechanism in PRIMOS to optimize frequent reuse of EPFs. Therefore, use the k$invk key if the target program EPF may be invoked more than once by the program or user. Use the k$invk_del key if you are certain that the invocation of the target program EPF by the calling program will be the last such invocation by that user for some time.

Run a Program EPF



Calling Sequence of EPF$RUN
Figure 19-2

The File Unit: Pass the file unit on which the target program EPF is open for VMFA-read (from Step 1) in unit.

The Error Code: When EPF$RUN returns, the value in code indicates the success or failure of the operation. If code is 0, the target program EPF was successfully invoked, although it may not have completed successfully.

If code is not zero, an error occurred while trying to invoke the EPF. In this case, your program should display an error message (using the ERRPR$ subroutine) and perhaps log the error; however, your program should not make use of any other information returned by EPF$RUN, such as severity-code or rtn-fcn-ptr, because these variables are assigned only as a result of successful invocation of the EPF.

See the section entitled Error Codes From EPF$RUN, later in this chapter, for a partial list of error codes.

The Command Line: Pass the command line containing the command arguments for the target program EPF in command-line; if there are no arguments, pass the null string.

Note

Do not include the RESUME command or the program name in the command-line argument. Otherwise, the target program EPF treats the RESUME command as the first token in the command line, and the pathname of the program as the second token, rather than treating the information following RESUME program-name as the command line.

The Severity Code: When the EPF$RUN subroutine returns, if code is 0, severity-code contains the severity code of the invoked EPF. The interpretation of severity-code is strictly dependent on the program EPF itself; however, it is typically set and interpreted as follows:

Value	Meaning
0	Program completed successfully
< 0	Successful completion, defined operation not performed (warning)
> 0	Program did not complete successfully (error)

Note

Because severity-code may not be set by the target program EPF, preset it to 0 before calling EPF$RUN, so that the default value indicates successful completion. This is particularly important when invoking a program that does not use its command line to receive information, and hence may have a main entrypoint that does not accept any arguments.

The Command Information: There are currently two versions of the command information structure that your program can pass. Both of these versions are illustrated in Chapter 18. Typically, you can pass a version 0 structure, which contains only the command name and the version number. If your program must pass a pointer to local CPL variables, or if your program performs command preprocessing such as wildcards, it must pass a version 1 structure.

In PL1/G, version 0 the command-information structure is declared as follows:

```
dcl 1 command_state static,
      2 command_name char(32) var init(''),
      2 version fixed bin(15) init(0);
```

In PL1/G, version 1 the command-information structure is declared as follows:

```
dcl 1 command_state static,
      2 command_name char(32) var init(''),
      2 version fixed bin(15) init(1),
      2 cpl_local_vars_ptr ptr init(null()),
      2 cp_iter_info,  /* Command iteration info. */
        3 mod_after_date fixed bin(31) init(0),
        3 mod_before_date fixed bin(31) init(0),
        3 bk_after_date fixed bin(31) init(0),
        3 bk_before_date fixed bin(31) init(0),
        3 type_dir bit(1) init('1'b),
        3 type_segdir bit(1) init('1'b),
        3 type_file bit(1) init('1'b),
        3 type_acat bit(1) init('1'b),
        3 type_rbf bit(1) init('0'b),
        3 mbz1 bit(11) init('00000000000'b),
        3 verify_sw bit(1) init('0'b),
        3 botup_sw bit(1) init('0'b),
        3 mbz2 bit(14) init('00000000000000'b),
        3 walk_from fixed bin(15) init(2),
        3 walk_to fixed bin(15) init(999),
        3 in_iteration bit(1) init('0'b),
        3 in_wildcard bit(1) init('0'b),
        3 in_treewalk bit(1) init('0'b),
            3 mbz3 bit(13) init('0000000000000'b);
```

Before calling EPF$RUN, set command_name to the name of the target program EPF you are invoking (32 characters maximum). If you know the name of the program while writing the program, you may place the name in the INITIAL attribute for the declaration of command_name. If, in Step 1, your program called SRSFX$, then store the basename variable, returned by SRSFX$, in command_name. command_name should not contain the .RUN suffix of the program. The degree of flexibility you have in setting command_name depends solely upon the program EPF you are invoking; therefore, consult the specification for the appropriate program.

The INITIAL attributes used above indicate the default settings used by PRIMOS. If your program is performing wildcard selection, matching, treewalking, and so on, you may wish to have your program modify cp_iter_info appropriately.

If the program being invoked references CPL variables local to the CPL program that invoked it (and therefore the CPL program that invoked your program EPF), store the pointer passed to the main entrypoint of your program EPF (in the command-information structure argument) into cpl_local_vars_ptr before calling EPF$RUN. See Chapter 18 for more information on the command-information structure passed to program EPFs.

Function Call: The function-call bit indicates to the target EPF whether it should return a function value. If you do not intend to use the target program EPF as a command function, set this bit to 0. If you do intend to use the target program EPF as a command function, set this bit to 1.

The Returned Function Value Pointer: The rtn-fcn-ptr variable has the same meaning for EPF$RUN as it does for CP$ when used to invoke a function, as described earlier in this chapter.

The EPF Id: The returned value of EPF$RUN, when invoked as a function that returns a FULL INT value, is an internal PRIMOS identifier of the EPF that is valid only if code is 0 and your program did not supply a key value of k$invk_del. You may use this identifier in subsequent calls to EPFCPF, EPFINVK, and EPF$DEL, which are described below.

You do not need to declare EPF$RUN as a function if you do not intend to use the returned EPF identifier.

Step 3: Close the EPF File

After EPF$RUN returns, close the file unit on which the target program EPF is open by calling SRCH$$. For example:

```
    call clo$fu(unit,i);  /* Don't overwrite CODE! */
```

Note

It is not necessary to repeatedly open and close a program EPF file when repeated invocations of the EPF are to be performed. The program EPF file can be opened once, invoked several times via EPF$RUN, and then closed once.

Step 4: Check the EPF$RUN Error Code

After closing the EPF file, check the returned *code* value. If *code* is 0, proceed to step 4. Otherwise, *code* contains a standard PRIMOS error code; use ERRPR$ or ERTXT$ to report the error to the user or to log the error. A listing of possible error codes that may be returned by EPF$RUN is provided later in this chapter, following the description of Step 6.

Step 5: Check the Returned Command Status

After you check the returned error code, check the returned *severity-code* value to determine whether the target program EPF completed successfully. The exact meaning of *severity-code* is defined by the target program EPF. Typically, if *severity-code* is 0, the program completed successfully; if *severity-code* is less than 0, the program encountered problems or unusual conditions but probably completed successfully; if *severity-code* is greater than 0, the program completed unsuccessfully.

Step 6: Use and Free the Returned Function Value Structure

If you invoked the target program EPF as a function, if *code* was set to 0 by EPF$RUN, and if *rtn-fcn-ptr* was not set to the null pointer by the target program EPF, your program should first use (such as by copying) the returned function value and then return its structure to the pool of available memory. Use FRE$RA to do this, as described later in this chapter.

Error Codes From EPF$RUN

An output argument, code, informs the calling program of the success or failure of the operation. This argument is a HALF INT variable. Symbols are provided to allow PL1/G, FORTRAN, Pascal, and PMA programs to substitute mnemonic keywords for numeric values.

If code is 0, the operation was entirely successful. Otherwise, code has one of many values. Typical values and their meanings follow. Not all possible error codes are listed; for example, PRIMENET-related error codes such as E$RLDN (The remote line is down) may be returned by EPF$RUN, but are not listed.

Note

When you use EPF$RUN which itself invokes other EPF$ subroutines, an error code returned by any of those subroutines is returned by EPF$RUN. Therefore, consult the lists of error codes returned by EPFMAP, EPFALLC, EPF$INIT, EPF$INVK, and EPF$DEL, later in this chapter, for information on additional error codes returnable by EPF$RUN.

Keyword	Value	Meaning
<ok>	0	The operation was successful.
E$EOF	1	End of file. This error indicates a file that has been truncated by FIX_DISK during system maintenance procedures. You must replace the program with a backup copy.
E$UNOP	3	Unit not open. There is no file open on unit. You must open the target program EPF for VMFA-read before calling EPF$RUN to invoke the EPF.
E$BKEY	28	Bad key in call. You are not passing a valid key value to EPF$RUN.
E$BUNT	29	Bad unit number. You are not passing a valid unit value to EPF$RUN.
E$ROOM	55	No room. You cannot invoke the EPF because there is insufficient dynamic storage available to allocate internal EPF information. Use the LIST_EPF and REMOVE_EPF commands to remove inactive EPFs, thereby freeing up dynamic storage.
E$NMTS	106	No more temp segments. You cannot invoke the EPF because you would exceed your limit on dynamic segments. This limit is

		displayed using the LIST_LIMITS command. You should use the LIST_EPF and REMOVE_EPF commands to remove inactive EPFs, thereby freeing up dynamic segments, and attempt to run your program again. If you need more dynamic segments, contact your System Administrator.
E$NMVS	107	No more VMFA segments. You cannot invoke the EPF because there are insufficient segments. The condition may be temporary, in which case an attempt to invoke the target EPF later might succeed. If the condition recurs, consult your System Administrator about increasing the number of VMFA segments on your system (by changing the NVMFS configuration directive in the system startup file).
E$BVER	158	Incorrect version number. Typically, this error means that the function invoked by the call to EPF$RUN returned a structure containing an invalid version number. Alternatively, this error means that the version number of the EPF itself is invalid. In both cases, the fault is in the target EPF, not the calling program.

THE EPF$INVK SUBROUTINE

The EPF$INVK subroutine provides a more controlled, step-by-step interface to the invocation of a program EPF than does the EPF$RUN subroutine. In most ways, however, the use of EPF$INVK is identical to the use of EPF$RUN. This section concentrates primarily on the differences between the use of these two subroutines.

The EPF$INVK subroutine is used in the following manner:

1. The calling program opens the program EPF file to be invoked.

2. The calling program calls EPF$MAP to map the EPF to memory, passing the file unit number of the opened program EPF file and obtaining an EPF identifier for use with the other EPF$ subroutines (except for EPF$RUN, described above).

3. The calling program closes the program EPF.

4. The calling program optionally calls EPF$CPF to obtain information on the EPF, such as its selection of command processing features, passing the EPF identifier.

5. The calling program calls EPF$ALLC to allocate the linkage areas for the EPF.

6. The calling program calls EPF$INIT to initialize the linkage areas for the EPF.

7. The calling program calls EPF$INVK to invoke the program EPF.

8. After the EPF$INVK subroutine completes, the calling program checks the returned error code to determine whether the program EPF was successfully invoked by EPF$INVK.

9. If the error code from EPF$INVK is 0, the calling program uses the information returned by EPF$INVK to determine whether the program EPF completed successfully or unsuccessfully, and optionally to access the returned text string (if the program EPF was invoked as a function).

10. If the error code from EPF$INVK is 0, and the calling program invoked the program EPF as a function, the calling program uses the FRE$RA subroutine to return the memory used to store the returned text string to the free memory pool.

11. The calling program calls EPF$DEL to remove the program EPF from memory.

Some of these steps are described in the section entitled <u>THE EPF$RUN SUBROUTINE</u>, earlier in this chapter; Steps 1 and 3 correspond to the same-numbered steps, while Steps 8 through 10 correspond to Steps 4 through 6 in the aforementioned section. These steps are not described below.

Step 2 and Steps 4 through 6 correspond to calling the EPF$RUN subroutine with a <u>key</u> value of <u>k$restore_only</u> as described earlier in this chapter. You may choose to use EPF$RUN rather than EPF$MAP, EPF$ALLC, and EPF$INIT if that is more appropriate for your application. After calling EPF$RUN with the <u>k$restore_only</u> key, close the program EPF file as described in Step 3, then continue with Step 7 of the above procedure to invoke the EPF.

For repeated invocations of the same program EPF, repeat Steps 6 through 10. It is because avoiding Steps 1 through 5 and Step 11 for subsequent invocations of an EPF saves time that the use of EPF$INVK is sometimes preferred over the use of EPF$RUN.

Steps peculiar to the use of EPF$INVK are described in detail below.

<u>Step 2: Invoke EPF$MAP</u>

The calling program calls EPF$MAP to map the EPF to memory, passing the file unit number of the opened program EPF file and obtaining an EPF identifier for use with the other EPF$ subroutines (except for EPF$RUN, described above). This corresponds to Phase 4 of the life of an EPF, as described in Chapter 1.

Figure 19-3 illustrates the calling sequence for the EPF$MAP subroutine.

The EPF$MAP subroutine may be used to map either a program EPF or a library EPF. Although this chapter does not describe the use of EPF$ subroutines on library EPFs, most of them work identically with library EPFs as they do with program EPFs. The exception is EPF$INVK, which supports only the invocation of program EPFs.

The arguments in the EPF$MAP calling sequence are described next.

The Key: Specify either k$any or k$copy for key. (The value k$dbg is used only by DBG, Prime's source-level debugger. You may use it, but it only increases the amount of virtual memory used by an EPF compiled with the -DEBUG option, without providing any additional functionality.)

The k$any key is most often used, because it specifies that the EPF is to be mapped to any available segments. The procedure (PROC) segments of a mapped EPF cannot be modified by a user, because they may be shared between users by PRIMOS.

The k$copy key is used when the invoking program intends to modify the procedure (PROC) segments of the EPF. Instead of mapping the procedure segments to memory, k$copy causes EPF$MAP to copy their contents into memory as for static-mode programs. Use the k$copy key if you plan to set breakpoints in an EPF via VPSD, for example.

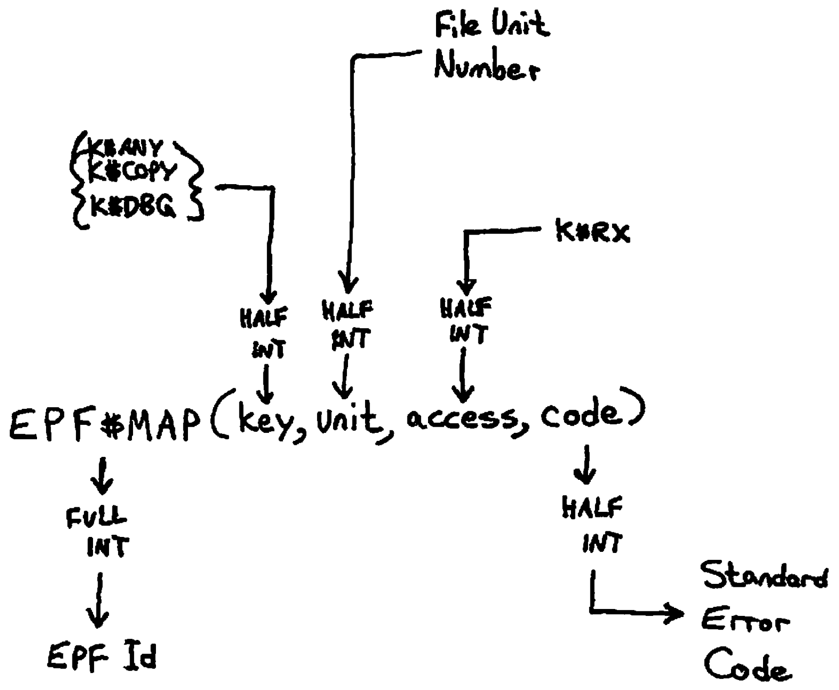
The File Unit Number: Pass the file unit number of the EPF in unit. This is the unit on which your program opened the EPF runfile for VMFA-read in Step 1. Once you have called EPF$MAP, you can close this unit.

The Segment Access: Pass k$rx in access. This represents the desired segment access. Only one other value is allowed in access, the value k$r. However, both k$rx and k$r result in the same effective segment access — read and execute access. Therefore, always use k$rx access in case k$r is someday redefined to mean something different (such as read-only access).

The Error Code: A standard error code is returned in code. Possible errors codes are summarized later in this chapter.

The EPF Identifier: The returned FULL INT value is an identifier of the mapped EPF that your program passes to subsequent EPF$ subroutines to identify the EPF.

Map an EPF to Memory



Calling Sequence of EPF$MAP
Figure 19-3

Step 4: Invoke EPF$CPF (Optional)

The calling program optionally calls EPF$CPF to obtain information on the EPF, such as its selection of command processing features, passing the EPF identifier to identify the EPF.

Figure 19-4 illustrates the calling sequence of the EPF$CPF subroutine.

The epf-id and code arguments have the obvious meanings. The epf-info structure, which may be used by your program to select valid command processing features, has the following declaration in PL/1:

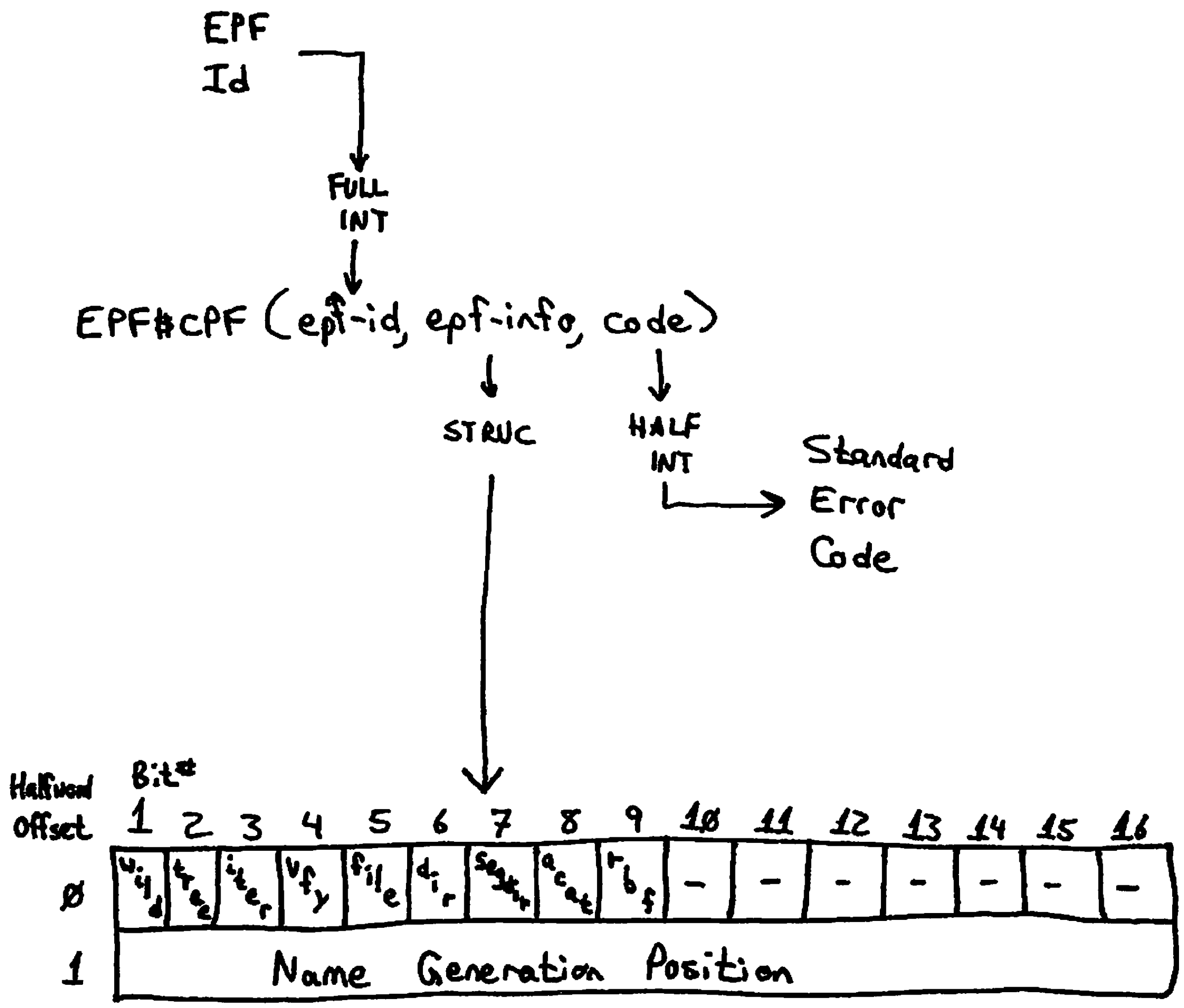
```
dcl 1 epf_info, /* EPF info data structure */
      2 command_flags,
        3 wildcards bit(1), /* Enable wildcards. */
        3 treewalks bit(1), /* Enable treewalks. */
        3 iteration bit(1), /* Enable iteration. */
        3 verify bit(1), /* Verify wildcard selections. */
        3 file_types,
          4 file bit(1), /* Select files. */
          4 directory bit(1), /* Select directories. */
          4 segdir bit(1), /* Select segment directories. */
          4 acat bit(1), /* Select access categories. */
          4 rbf bit(1), /* Select RBF files. */
          4 reserved bit(7), /* Ignore. */
      2 name_generation_position fixed bin(15); /* Token #. */
```

For wildcards, treewalks, and iteration, a bit set to 1 indicates that PRIMOS is to perform the corresponding function. For example, if wildcards is '1'B, PRIMOS intercepts a specification of @@ and expands the command line to several command lines, one for each file system object in the directory (as limited by the object selection in file_types). If wildcards is '0'B instead, PRIMOS passes a specification of @@ to the program EPF without modification, and no expansion takes place due to that specification.

The verify bit is the default setting of the -VERIFY or -NO_VERIFY (-VFY or -NVFY) options. When '1'B, the default is -VERIFY; when '0'B, the default is -NO_VERIFY. Actual verification takes place only when wildcards are being processed by the command processor — that is, when wildcards is set to '1'B and the command line contains an actual wildcard specification.

The file_types bits indicate the default settings of the -FILE, -SEGMENT_DIRECTORY (-SEGDIR), -DIRECTORY (-DIR), -ACCESS_CATEGORY (-ACAT), and -RBF options. A bit set to '1'B indicates that the corresponding file type is to be processed. The file_types bits are used only during wildcard processing, as with the verify bit. For example, if the command RESUME MYPROG XYZ is given, MYPROG is invoked for the file system object named XYZ even if XYZ is a directory and the directory bit is reset to '0'B. However, if the command RESUME MYPROG XYZ@@ is given (and the wildcards bit is '1'B), the XYZ directory is

Obtain Information on EPF



Calling Sequence of EPF$CPF
Figure 19-4

not selected if directory is '0'B, because wildcard processing is taking place.

The name_generation_position variable is an integer that specifies which token following the program or command name is to be used as the name generation source pattern. Normally, this variable is set to 1, meaning that the first token after the RESUME MYPROG tokens is to be used as the source pattern. For example, the command line

```
RESUME MYPROG FOO BAR ==
```

produces an effective command line of:

```
RESUME MYPROG FOO BAR FOO
```

However, if name_generation_position is 2, the second token is used instead. For example, given the same command line above, the effective command line produced when name_generation_position is 2 is:

```
RESUME MYPROG FOO BAR BAR
```

For a program EPF installed in CMDNC0, the token count begins at the same point; that is, following the program name. Therefore, the following two command lines always produce the same result with regard to name generation pattern processing:

```
MYPROG A B ==

RESUME CMDNC0>MYPROG A B ==
```

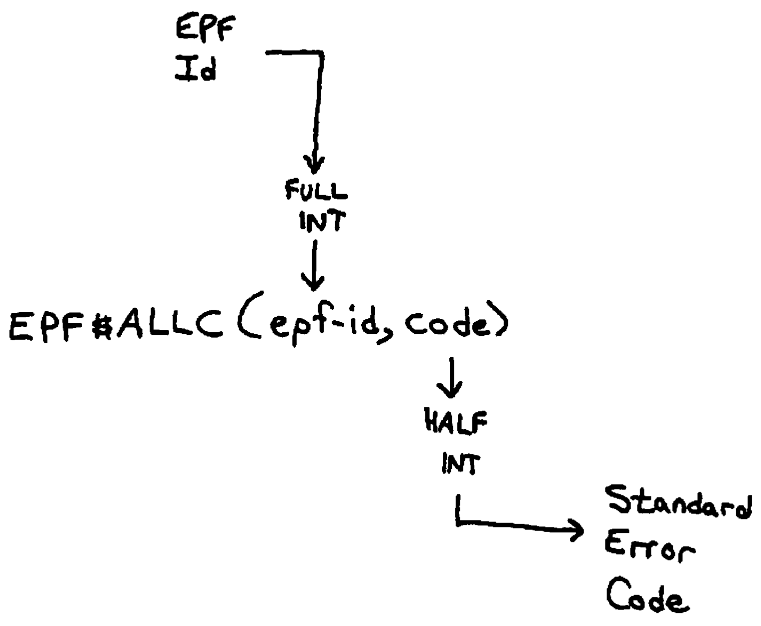
Step 5: Invoke EPF$ALLC

The calling program now calls EPF$ALLC to allocate the linkage areas for the EPF, passing the EPF identifier. This step corresponds to Phase 5 of the life of an EPF.

Figure 19-5 illustrates the calling sequence of the EPF$ALLC subroutine.

The epf-id and code arguments have the usual meanings.

Allocate Linkage Areas for EPF



Calling Sequence of EPF$ALLC
Figure 19-5

Step 6: Invoke EPF$INIT

The calling program calls EPF$INIT to initialize the linkage areas for the EPF, passing the EPF identifier. This step corresponds to Phase 6 of the life of an EPF.

Figure 19-6 illustrates the calling sequence of the EPF$INIT subroutine.

The epf-id and code arguments have the usual meanings.

The key argument specifies whether a complete initialization is to be performed. The first time EPF$INIT is called for an EPF that has just had its linkage allocated via EPF$ALLC, key must be set to k$initall, which specifies complete initialization. After calling EPF$INVK, in the next step, a subsequent invocation of the program requires only a call to EPF$INIT with a key of k$reinit to reinitialize only certain portions of the linkage areas for the EPF before calling EPF$INVK again.

Specifically, while a key of k$initall specifies complete initialization of the linkage areas, a key of k$reinit specifies that only faulted IPs (dynamic links) and static data are to be reinitialized. ECBs, static IPs, and other nonfaulted IPs are not reinitialized — once initialized, they do not need to be initialized again unless the program modifies them during execution (which is considered poor programming practice).

If a program being invoked by your program seems to fail in strange ways after the first invocation, have your program use the k$initall key exclusively to see if the problem is caused by the invoked program — it might be modifying linkage data that should not be modified once it has been initialized by EPF$INIT.

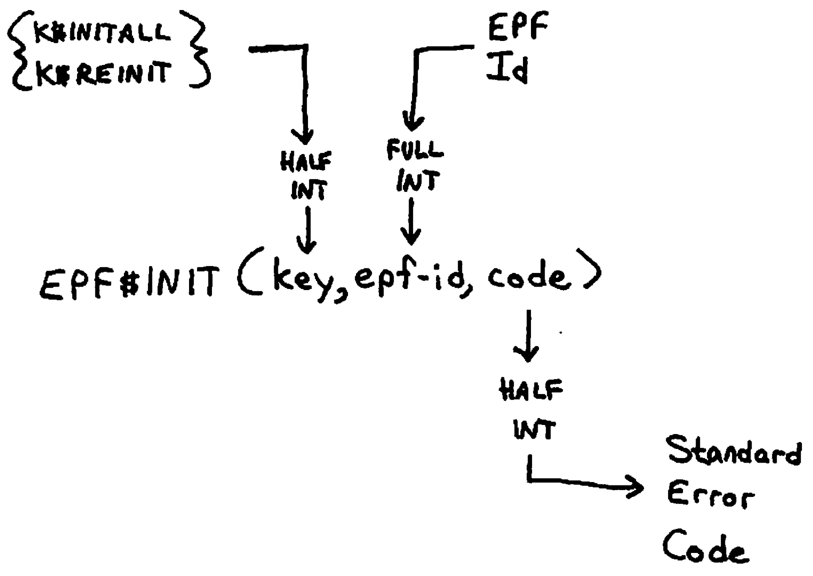
Step 7: Invoke EPF$INVK

The calling program calls EPF$INVK to invoke the program EPF, passing the EPF identifier. This step corresponds to Phase 7 of the life of an EPF.

Figure 19-7 illustrates the calling sequence for the EPF$INVK subroutine.

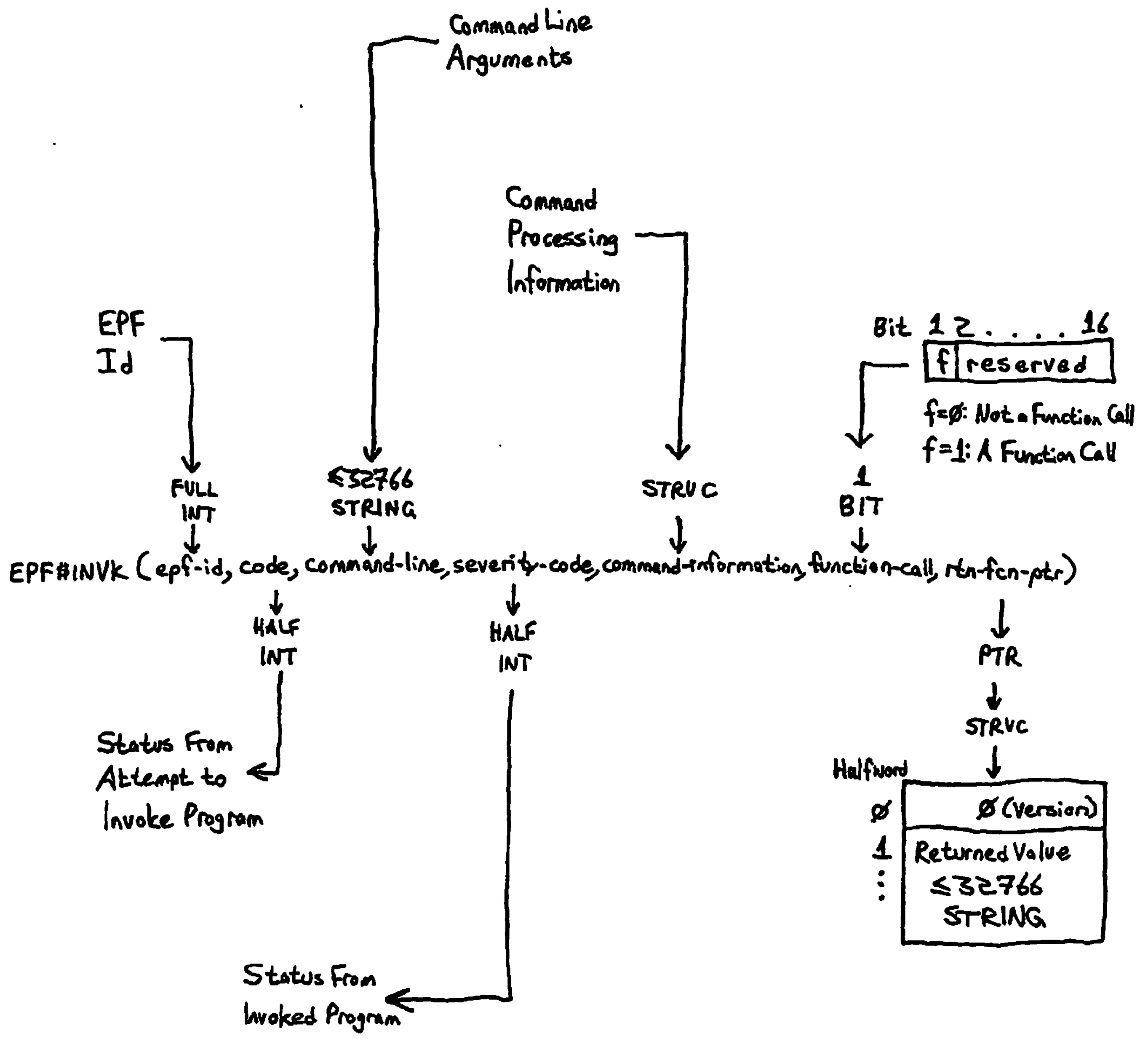
The epf-id and code arguments have the usual meanings. The remaining arguments correspond precisely to the same arguments to the EPF$RUN subroutine, described earlier in this chapter. In fact, as with EPF$RUN, the latter five arguments may be omitted if the main entrypoint of the target program EPF does not accept any arguments.

Initialize Linkage Areas for EPF



Calling Sequence of EPF$INIT
Figure 19-6

Invoke a Program EPF


Command Line Arguments
Command Processing Information
EPF Id
Bit 1 2 16
f reserved
f=Ø: Not a Function Call
f=1: A Function Call
FULL INT
≤32766 STRING
STRUC
1 BIT
EPF$INVK (epf-id, code, command-line, severity-code, command-information, function-call, rtn-fcn-ptr)
HALF INT
HALF INT
PTR
STRUC
Status From Attempt to Invoke Program
Halfword
Ø
1
Ø (Version)
Returned Value ≤32766 STRING
Status From Invoked Program

Calling Sequence of EPF$INVK
Figure 19-7

Step 11: Invoke EPF$DEL

The calling program calls EPF$DEL to remove the program EPF from memory, passing the EPF identifier. This step corresponds to Phase 10 of the life of an EPF.

Figure 19-8 illustrates the calling sequence of the EPF$DEL subroutine.

The epf-id and code arguments have the usual meanings.

If the EPF is still in use by this process, such as when a user types Control-P while the program is executing, then the EPF is not removed and an error code (e$swpr) is returned in code.

The EPF is not actually removed from the system's virtual memory if other users have the EPF mapped to their memory. However, it is unmapped from the calling user's memory, and is removed from the system's virtual memory when the last user unmaps it from his or her memory.

Error Codes From EPF$ Subroutines

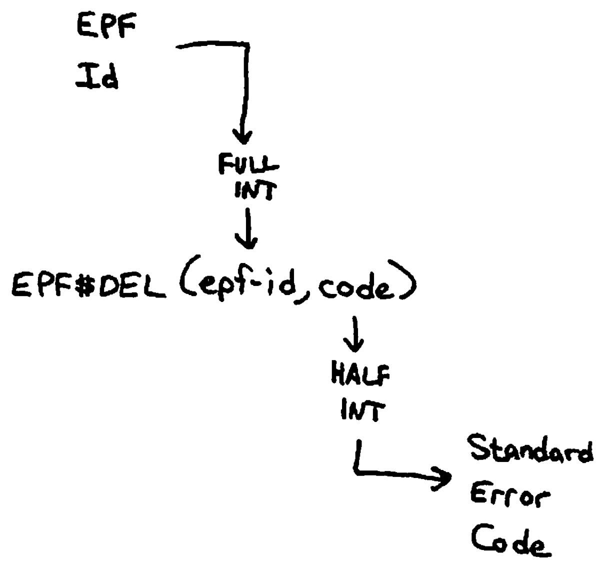
All of the EPF$ subroutines may encounter errors. In addition, opening a file for VMFA-read may result in an error that pertains specifically to the VMFA mechanism rather than the file access mechanism. An output argument, code, informs the calling program of the success or failure of the operation. This argument is a HALF INT value. Symbols are provided to allow PL1/G, FORTRAN, Pascal, and PMA programs to substitute mnemonic keywords for numeric values.

If code is 0, the operation was entirely successful. Otherwise, code has one of many values. Typical values and their meanings are listed for each EPF$ subroutine. Not all possible error codes are listed; for example, PRIMENET-related error codes such as E$RLDN (The remote line is down) may be returned by one or more of these subroutines, but are not listed.

Error Codes Involving the K$VMR Key: Error codes specific to opening a file for VMFA-read (using the k$vmr key) are listed below. Other error codes applying to opening files in general may also be returned.

Keyword	Value	Meaning
<ok>	0	The operation was successful.
E$NRIT	10	The user has insufficient access to open the target file for VMFA-read. Currently, Read access to the file is required.
E$NDAM	109	The target object is not a DAM file; this

Remove an EPF From Memory



Calling Sequence of EPF$DEL
Figure 19-8

error code is also returned if an attempt is made to open the cache directory by specifying the k$curr value for the filename or by specifying a null pathname.

Error Codes From EPF$MAP: Error codes that may be returned by EPF$MAP are:

Keyword	Value	Meaning
<ok>	0	The operation was successful.
E$UNOP	3	The unit specified in unit is not open.
E$BAR	6	An invalid segment access has been specified in access. It must be either k$rx or k$r.
E$BKEY	28	The value of key is invalid.
E$BUNT	29	The value specified in unit is an invalid file unit number.
E$NMVS	107	There are not enough VMFA segments in the system to accommodate the EPF. If this errors persists, contact your System Administrator, who may wish to increase the number of VMFA segments on your system (via the NVMFS configuration directive in the system configuration file).
E$NMTS	108	There are no more temporary segments available into which the EPF procedure segments can be copied.
E$NDAM	109	The file open on unit is not a DAM file.
E$NOVA	110	The file open on unit is not open for VMFA-read. It must be opened using the k$vmr key.
E$BVER	158	Invalid EPF version. The file open for VMFA-read is either a corrupted EPF, not an EPF, or an EPF generated by a future revision of PRIMOS that is not supported by the current revision of PRIMOS.
E$EPFT	217	The file open for VMFA-read on the file unit is not a valid EPF. Either the file contains a corrupted EPF or is not an EPF at all, or the file contains an EPF generated by a revision of PRIMOS beyond

		Rev. 19.4 that is not recognized by Rev. 19.4 PRIMOS.
E$EPFL	222	The EPF file is too large for the current EPF implementation. More segments are required by the EPF than are supported by the current revision of PRIMOS. If you are using the -DEBUG option, recompile the program without the option to reduce its size. Alternatively, consider splitting the program up into smaller pieces, such as one program EPF and one or more library EPFs.

Error Codes From EPF$CPF: Error codes that may be returned by EPF$CPF are:

Keyword	Value	Meaning
<ok>	0	The operation was successful.
E$BPAR	6	The epf-id passed represents an EPF that is no longer mapped to memory.

Error Codes From EPF$ALLC: Error codes that may be returned by EPF$ALLC are:

Keyword	Value	Meaning
<ok>	0	The operation was successful.
E$BPAR	6	The epf-id passed represents an EPF that is no longer mapped to memory.
E$BVER	158	Invalid EPF version. The EPF is either a corrupted EPF, not an EPF, or an EPF generated by a future revision of PRIMOS that is not supported by the current revision of PRIMOS. Because this condition is checked by EPF$MAP, this error is not likely to occur when calling EPF$ALLC unless it is called out of sequence.
E$EPFT	217	The EPF is not a valid EPF. Either the file contains a corrupted EPF or is not an EPF at all, or the file contains an EPF generated by a revision of PRIMOS beyond Rev. 19.4 that is not recognized by Rev. 19.4 PRIMOS. Because this condition is checked by EPF$MAP, this error is not

		likely to occur when calling EPF$ALLC unless it is called out of sequence.
E$ILTD	219	The EPF contains an invalid linkage descriptor. The problem is not with the calling program; this error usually indicates a corrupted EPF file.

Error Codes From EPF$INIT: Error codes that may be returned by EPF$INIT are:

Keyword	Value	Meaning
<ok>	0	The operation was successful.
E$BPAR	6	The epf-id passed represents an EPF that is no longer mapped to memory.
E$BKEY	28	Either the key argument is invalid (not k$initall or k$reinit), or the k$reinit key is specified but the linkage areas for the EPF have not yet been fully initialized (by specifying the k$initall key in a call to EPF$INIT).
E$BARG	71	The EPF$ALLC has not yet been successfully called to allocate linkage areas for this EPF.
E$BVER	158	Invalid EPF version. The EPF is either a corrupted EPF, not an EPF, or an EPF generated by a future revision of PRIMOS that is not supported by the current revision of PRIMOS. Because this condition is checked by EPF$MAP and EPF$ALLC, this error is not likely to occur when calling EPF$INIT unless it is called out of sequence.
E$EPFT	217	The EPF is not a valid EPF. Either the file contains a corrupted EPF or is not an EPF at all, or the file contains an EPF generated by a revision of PRIMOS beyond Rev. 19.4 that is not recognized by Rev. 19.4 PRIMOS. Because this condition is checked by EPF$MAP and EPF$ALLC, this error is not likely to occur when calling EPF$INIT unless it is called out of sequence.
E$ILTD	219	The EPF contains an invalid linkage descriptor. The problem is not with the

		calling program; this error usually indicates a corrupted EPF file.
E$ILTE	220	The EPF contains an invalid linkage descriptor. The problem is not with the calling program; this error usually indicates a corrupted EPF file.

Error Codes From EPF$INVK: Error codes that may be returned by EPF$INVK including any codes that may be returned by EPF$DEL in addition to those listed below.

Keyword	Value	Meaning
<ok>	0	The operation was successful.
E$BPAR	6	The epf-id passed represents an EPF that is no longer mapped to memory.
E$BVER	158	Invalid EPF version. The EPF is either a corrupted EPF, not an EPF, or an EPF generated by a future revision of PRIMOS that is not supported by the current revision of PRIMOS. Because this condition is checked by EPFMAP, EPFALLC, and EPF$INIT, this error is not likely to occur when calling EPF$INVK unless it is called out of sequence.
E$EPFT	217	The EPF is not a valid EPF. Either the file contains a corrupted EPF or is not an EPF at all, or the file contains an EPF generated by a revision of PRIMOS beyond Rev. 19.4 that is not recognized by Rev. 19.4 PRIMOS. Because this condition is checked by EPFMAP, EPFALLC, and EPF$INIT, this error is not likely to occur when calling EPF$INVK unless it is called out of sequence.
E$ECEB	221	The command environment breadth limit has been reached; the currently running program can call no more programs. Use the LIST_LIMITS command to display command environment limits, or use the CE$BRD subroutine to determine the command environment breadth limit from within a program.

Error Codes From EPF$DEL: Error codes that may be returned by EPF$DEL are:

Keyword	Value	Meaning
<ok>	0	The operation was successful.
E$BPAR	6	The epf-id passed represents an EPF that is no longer mapped to memory.
E$BVER	158	Invalid EPF version. The EPF is either a corrupted EPF, not an EPF, or an EPF generated by a future revision of PRIMOS that is not supported by the current revision of PRIMOS. Because this condition is checked by EPFMAP, EPFALLC, EPF$INIT, and EPF$INVK, this error is not likely to occur when calling EPF$DEL unless it is called out of sequence.
E$EPFT	217	The EPF is not a valid EPF. Either the file contains a corrupted EPF or is not an EPF at all, or the file contains an EPF generated by a revision of PRIMOS beyond Rev. 19.4 that is not recognized by Rev. 19.4 PRIMOS. Because this condition is checked by EPFMAP, EPFALLC, EPF$INIT, and EPF$INVK, this error is not likely to occur when calling EPF$DEL unless it is called out of sequence.
E$SWPR	225	The EPF is suspended by this user (process), and hence cannot be unmapped from memory. This error code is returned if a program attempts to call EPF$DEL to unmap itself.

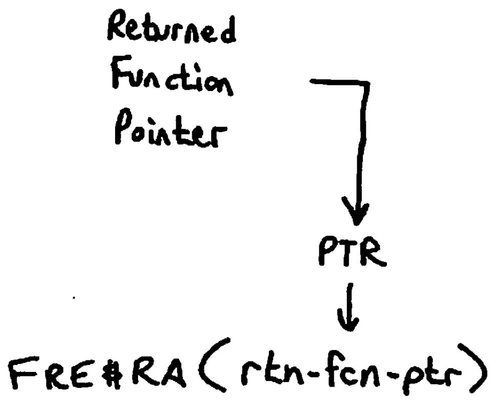
THE FRE$RA SUBROUTINE

After calling CP$, EPF$RUN, or EPF$INVK to invoke a function and after making use of the returned function value, your program must call FRE$RA to free the memory used to hold the returned function value. (Call FRE$RA only if the function your program invoked actually returned a function value.)

Figure 19-9 illustrates the calling sequence of FRE$RA. Simply pass the returned function pointer (rtn-fcn-ptr).

For information on how returned function values are set, see Chapter 18, including the descriptions of the ALS$RA and ALC$RA subroutines.

Free a Returned Function Value Structure



Calling Sequence of FRE$RA
Figure 19-9

SAMPLE PROGRAMS

The first sample program is called SLOW_INVOKE. It takes an EPF name and command arguments for the EPF as arguments to the program, and it then performs each step associated with executing the target EPF. After each step, it pauses so that the user may use the LIST_EPF -DETAIL command to see how far it has gotten. Although not necessarily a useful example by itself, this program does illustrate how each step is performed, and also shows the PL1/G declarations for the appropriate subroutines and structures.

```
slow_invoke: proc(x_command_line,code,command_state,command_flags,
      return_function_ptr);

dcl x_command_line char(1024) var,
    code fixed bin(15),
    1 command_state,
      2 com_name char(32) var,
      2 version fixed bin(15),
      2 vcb_ptr ptr,
      2 cp_iter_info,
         3 mod_after_date fixed bin(31),
         3 mod_before_date fixed bin(31),
         3 bk_after_date fixed bin(31),
         3 bk_before_date fixed bin(31),
         3 type_dir bit(1),
         3 type_segdir bit(1),
         3 type_file bit(1),
         3 type_acat bit(1),
         3 type_rbf bit(1),
         3 mbz1 bit(11),
         3 verify_sw bit(1),
         3 botup_sw bit(1),
         3 mbz2 bit(14),
         3 walk_from fixed bin(15),
         3 walk_to fixed bin(15),
         3 in_iteration bit(1),
         3 in_wildcard bit(1),
         3 in_treewalk bit(1),
         3 mbz3 bit(13),
    1 command_flags,
      2 command_function_call bit(1),
      2 mbz bit(15),
    return_function_ptr ptr;

%include 'SYSCOM>ERRD.INS.PL1';
%include 'SYSCOM>KEYS.INS.PL1';

dcl epf_unit fixed bin(15),
    epf_id fixed bin(31),
    epf_filename char(128) var,
    i fixed bin(15),
    command_line char(1024) var,
        epf_command_line char(1024) var,
        basename char(32) var,
        suffix_used fixed bin(15),
        type fixed bin(15),
        command_status fixed bin(15);

    dcl errpr$ entry(fixed bin(15),fixed bin(15),char(80),
          fixed bin(15),char(80),fixed bin(15)),
        srsfx$ entry(fixed bin(15),char(128) var,fixed bin(15),
          fixed bin(15),fixed bin(15),char(32) var,char(32) var,
          fixed bin(15),fixed bin(15)),
        clo$fu entry(fixed bin(15),fixed bin(15)),
        tnou entry(char(80),fixed bin(15)),
        epf$map entry(fixed bin(15),fixed bin(15),fixed bin(15),
          fixed bin(15)) returns(fixed bin(31)),
        epf$allc entry(fixed bin(31),fixed bin(15)),
        epf$init entry(fixed bin(15),fixed bin(31),fixed bin(15)),
        epf$invk entry(fixed bin(31),fixed bin(15),char(1024) var,
          fixed bin(15),
          1, 2 char(32) var,
             2 fixed bin(15),
             2 ptr,
             2, 3 fixed bin(31),
                3 fixed bin(31),
                3 fixed bin(31),
                3 fixed bin(31),
                3 bit(1),
                3 bit(1),
                3 bit(1),
                3 bit(1),
                3 bit(1),
                3 bit(11),
                3 bit(1),
                3 bit(1),
                3 bit(14),
                3 fixed bin(15),
                3 fixed bin(15),
                3 bit(1),
                3 bit(1),
                3 bit(1),
                3 bit(13),
           1, 2 bit(1),
              2 bit(15),
           ptr),
        epf$del entry(fixed bin(31),fixed bin(15));

    command_line=trim(x_command_line,'11'b);
    i=index(command_line,' ');

    if i=0 & command_line=''
       then do;
            code=e$ivcm;
            call errpr$(k$irtn,code,'Specify EPF filename',20,
                  'SLOW_INVOKE',11);
            return;
            end;

    if i=0
       then do;
            epf_filename=command_line;
            epf_command_line='';
            end;
       else do;
            epf_filename=substr(command_line,1,i-1);
            epf_command_line=trim(substr(command_line,i+1),'11'b);
            end;

    call srsfx$(k$getu+k$vmr,epf_filename,epf_unit,type,1,'.RUN',
         basename,suffix_used,code);
    if code^=0
       then do;
            call errpr$(k$irtn,code,(epf_filename),
                 length(epf_filename),'SLOW_INVOKE',11);
            return;
            end;

    call tnou('SRSFX$ complete',15);
    call pause_me;

    epf_id=epf$map(k$any,epf_unit,k$rx,code);
    call clo$fu(epf_unit,i);
    if code^=0
       then do;
            call errpr$(k$irtn,code,'Mapping '||epf_filename,
                 length(epf_filename)+8,'SLOW_INVOKE',11);
            return;
            end;

    call tnou('EPF$MAP complete',16);
    call pause_me;

    call epf$allc(epf_id,code);
    if code^=0
       then do;
            call clo$fu(epf_unit,i);
            call errpr$(k$irtn,code,'Allocating '||epf_filename,
                 length(epf_filename)+11,'SLOW_INVOKE',11);
            return;
            end;

    call tnou('EPF$ALLC complete',17);
    call pause_me;

    call epf$init(k$initall,epf_id,code);
    if code^=0
       then do;
            call clo$fu(epf_unit,i);
            call errpr$(k$irtn,code,'Initializing '||epf_filename,
                    length(epf_filename)+13,'SLOW_INVOKE',11);
                return;
                end;

    call tnou('EPF$INIT complete',17);
    call pause_me;

    command_status=0;
    command_state.com_name=basename;
    call epf$invk(epf_id,code,epf_command_line,command_status,
          command_state,command_flags,return_function_ptr);
    if code^=0
       then do;
            call clo$fu(epf_unit,i);
            call errpr$(k$irtn,code,'Invoking '||epf_filename,
                 length(epf_filename)+10,'SLOW_INVOKE',11);
            return;
            end;

    call tnou('EPF$INVK complete',17);
    call pause_me;

    call epf$del(epf_id,code);
    if code^=0
       then do;
            call clo$fu(epf_unit,i);
            call errpr$(k$irtn,code,'Removing '||epf_filename,
                 length(epf_filename)+9,'SLOW_INVOKE',11);
            return;
            end;

    call tnou('EPF$DEL complete',16);
    call pause_me;

    code=command_status;
    return;

    pause_me: proc;

    dcl pause_ char(32) var static init('PAUSE$');

    dcl signl$ entry(char(32) var,ptr options(short),fixed bin(15),
          ptr options(short),fixed bin(15),bit(1) aligned);

    call signl$(pause_,null(),0,null(),0,'1'b);

    end;  /* pause_me: proc */

    end;
```

The next sample program, called DISPLAY_EPF_INFO, displays command processing information for an EPF by mapping it to memory, calling EPF$CPF, and then removing the EPF from memory. It illustrates how to process the information returned by EPF$CPF.

```
display_epf_info: proc(command_line,code,command_state,
      command_flags,return_function_ptr);

dcl command_line char(1024) var,
    code fixed bin(15),
    1 command_state,
      2 com_name char(32) var,
      2 version fixed bin(15),
    1 command_flags,
      2 command_function_call bit(1),
      2 mbz bit(15),
    return_function_ptr ptr;

%include 'SYSCOM>ERRD.INS.PL1';
%include 'SYSCOM>KEYS.INS.PL1';

dcl epf_unit fixed bin(15),
    epf_id fixed bin(31),
    epf_filename char(128) var,
    i fixed bin(15),
    basename char(32) var,
    suffix_used fixed bin(15),
    type fixed bin(15);

dcl errpr$ entry(fixed bin(15),fixed bin(15),char(80),
      fixed bin(15),char(80),fixed bin(15)),
    srsfx$ entry(fixed bin(15),char(128) var,fixed bin(15),
      fixed bin(15),fixed bin(15),char(32) var,char(32) var,
      fixed bin(15),fixed bin(15)),
    clo$fu entry(fixed bin(15),fixed bin(15)),
    tnou entry(char(80),fixed bin(15)),
    epf$map entry(fixed bin(15),fixed bin(15),fixed bin(15),
      fixed bin(15)) returns(fixed bin(31)),
    epf$del entry(fixed bin(31),fixed bin(15));

if command_line=''
   then do;
        code=e$ivcm;
        call errpr$(k$irtn,code,'Specify EPF filename',20,
             (com_name),length(com_name));
        return;
        end;

epf_filename=command_line;

call srsfx$(k$getu+k$vmr,epf_filename,epf_unit,type,1,'.RUN',
      basename,suffix_used,code);
if code^=0
```

```
   then do;
        call errpr$(k$irtn,code,(epf_filename),
             length(epf_filename),(com_name),length(com_name));
        return;
        end;

epf_id=epf$map(k$any,epf_unit,k$rx,code);
call clo$fu(epf_unit,i);  /* Close the unit. */
if code^=0
   then do;
        call errpr$(k$irtn,code,'Mapping '||epf_filename,
             length(epf_filename)+8,(com_name),length(com_name));
        return;
        end;

call say_nl(trim(char(epf_id),'11'b));

call show_epf_info(epf_id);  /* Display the information. */

call epf$del(epf_id,code);
if code^=0 & code^=e$swpr
   then do;
        call clo$fu(epf_unit,i);
        call errpr$(k$irtn,code,'Removing '||epf_filename,
             length(epf_filename)+9,(com_name),length(com_name));
        return;
        end;
   else if code=e$swpr
          then call say_nl('(Still suspended by this process.)');

code=0;
return;

show_epf_info: proc(epf_id);

dcl epf_id fixed bin(31);

dcl code fixed bin(15),
     1 epf_info,  /* EPF info data structure */
       2 command_flags,
         3 wildcards bit(1),
         3 treewalks bit(1),
         3 iteration bit(1),
         3 verify bit(1),
         3 file_types,
           4 file bit(1),
           4 directory bit(1),
           4 segdir bit(1),
           4 acat bit(1),
           4 rbf bit(1),
           4 reserved bit(7),
       2 name_generation_position fixed bin(15);

dcl epf$cpf entry(fixed bin(31),
        1, 2, 3 bit(1),
              3 bit(1),
              3 bit(1),
              3 bit(1),
              3,
                 4 bit(1),
                 4 bit(1),
                 4 bit(1),
                 4 bit(1),
                 4 bit(1),
                 4 bit(7),
           2 fixed bin(15),
           fixed bin(15));

    /* Call EPF$CPF to get the information. */

    call epf$cpf(epf_id,epf_info,code);
    if code^=0 then call errpr$(k$irtn,code,'Calling EPF$CPF',15,
                            (com_name),length(com_name));
       else do;

    /* Command processing info. */

            call say_nl('');
            call say_nl('Info on '||epf_filename||':');
            call say_nl('');

            call say('Command processing:');

            if epf_info.wildcards then call say(' wild');
            if epf_info.treewalks then call say(' tree');
            if epf_info.iteration then call say(' iter');
            if epf_info.verify then call say(' vfy');

            call say_nl('');
            call say('Object selection:');

            if epf_info.file then call say(' file');
            if epf_info.directory then call say(' dir');
            if epf_info.segdir then call say(' segdir');
            if epf_info.acat then call say(' acat');
            if epf_info.rbf then call say(' rbf');

            call say_nl('');
            call say_nl('Name generation position: '
                  ||trim(char(name_generation_position),'11'b));

            call say_nl('');

            end;

    end;  /* show_epf_info: proc */

    say: proc(text);
    dcl text char(*) var;

    dcl tnoua entry(char(*),fixed bin(15));

    call tnoua((text),length(text));

    end; /* say: proc */

    say_nl: proc(text);

    dcl text char(*) var;

    dcl tnou entry(char(*),fixed bin(15));

    call tnou((text),length(text));

    end; /* say: proc */

    end;
```

IF A PROGRAM INVOKES ITSELF

A program may invoke itself recursively, either directly by calling itself using CP$, EPF$RUN, or EPF$INVK, or indirectly by calling another program or collection of programs that ultimately call the original program.

A program invoking itself recursively via CP$, EPF$RUN, or EPF$INVK, whether directly or indirectly, does not necessarily produce the same results as if it calls itself by invoking its own main entrypoint. In both cases, dynamic storage is allocated and initialized for each invocation. However, static storage is allocated only during program invocation; it is allocated for all procedures in that program each time the program is invoked. Once the program is running, no additional static storage is allocated by PRIMOS.

PRIMOS allocates and initializes one copy of static storage per program invocation. Static storage includes COMMON and STATIC EXTERNAL areas except for those explicitly named using the SYMBOL subcommand of BIND. In addition, static storage contains subroutine linkage pointers, static data (SAVE or DATA in FORTRAN, STATIC in PL/1), and program constants.

Because PRIMOS separates program invocations so that they cannot destroy one another's data, one program can be invoked and then suspended, reinvoked, then the original invocation can be continued by issuing the START command. The second invocation of the program does not affect the first invocation of the program; therefore, the results of the first invocation are essentially unchanged.

Of course, if a program makes use of data that is not in static or dynamic storage, such as COMMON or STATIC EXTERNAL storage specified using the SYMBOL command, then separate invocations of the program are not necessarily independent of each other. Other data not in static or dynamic storage includes system objects such as attach points, files, file units, and so on. PRIMOS does not provide a fully recursive command environment, it provides only a separation of per-program data between program invocations. See Chapter 21 for more information on this subject.

TERMINAL INPUT AND OUTPUT

Keep in mind that invoking a command from within a program does not redirect terminal input or output. For example, if you invoke the LD command from within a program, the output from LD is sent to the user terminal, and responses to the --More-- prompt are solicited from the user terminal.

Therefore, you may wish to use the COMO$$ subroutine or the internal PRIMOS command COMOUTPUT to redirect terminal output to a command output file. To redirect terminal input to a command input file written by your program, you may use COMI$$ or the internal PRIMOS command COMINPUT; alternatively, when supported by the command, you may specify an option indicating how to substitute for terminal input. (For example, LD accepts a -NO_WAIT option, which specifies that --More-- prompts are not to be issued.)

Most functions are designed and written to not perform any terminal I/O, or to allow the invoking program to disable or redirect terminal I/O by specifying command line options.

CHAPTER 20

The Command Processor Stack

To be supplied in the First Edition.

CHAPTER 21

The Recursive Command Environment

To be supplied in the First Edition.

APPENDIX A

New Features for the Advanced Programmer

To be supplied in the First Edition.

APPENDIX B

Error Codes and Messages

Listed in this Appendix are the standard PRIMOS file system error codes. The description of each error code is in the form:

▶ E$xxxx (nn) error-message

description-of-error

The mnemonic for the error code is E$*xxxx*; the value of the mnemonic is *nn*; the error message displayed by ERRPR$ for that error code is *error-message*; and *description-of-error* is the description of the error code.

Mnemonics for error codes are defined by files in SYSCOM for several languages:

Language	File Name in SYSCOM
FTN	ERRD.INS.FTN
Pascal	ERRD.INS.PASCAL
PL/1-G	ERRD.INS.PL1
PMA	ERRD.INS.PMA

You use the appropriate %INCLUDE (Pascal and PL/1-G) or $INSERT (FTN and PMA) in your program to provide definitions of all the standard error codes for your program.

The Subroutines Reference Guide contains more information on these four files.

STANDARD FILE SYSTEM ERROR CODES

▶ E$EOF (1) — End of file.

Description to be supplied.

▶ E$BOF (2) — Beginning of file.

Description to be supplied.

▶ E$UNOP (3) — Unit not open.

The file unit is closed, or is not open for the type of operation being requested. For example, an attempt to read from a file that is open only for writing causes this error, as does an attempt to write to a file that is open only for reading.

This error code is also returned if an attempt is made to truncate a file that is not open for writing.

▶ E$UIUS (4) — Unit in use.

Description to be supplied.

▶ E$FIUS (5) — File in use.

The file system object being accessed is already open on another file unit, or by another user. This error occurs if an attempt is made to:

- Open an object that is already open by another user or by the same user on another file unit, and the read/write lock of the object disallows the attempt
- Rename an object that is open by another user or by the same user on another file unit
- Rename a file directory that is in use as an attach point by any user
- Set a quota on a nonquota directory that is in use or contains

other files or directories that are in use

- Change the open mode of a file unit, by calling CH$MOD or SRCH$$ (with the K$CACC key), when the object is open by another user or by the same user on another file unit and the new open mode conflicts with the other open mode
- Truncate a file or segment directory that is open by another user or by the same user on another file unit

If your program is accessing a file that may occasionally be in use, consider having your program retry the aborted operation several times, sleeping for a second or so in between each operation. For example:

```
code=e$fius; /* Assume error. */
do i=1 to 60 while(code=e$fius); /* Up to 60 seconds wait. */
   call cnam$$(oldnam,oldlen,newnam,newlen,code);
   if code=e$fius then call sleep$(1000); /* Sleep a sec. */
   end;
```

If you need to be able to read a file while it is being written, you can change the read/write lock of the file by using the RWLOCK command or the SATR$$ subroutine. The read/write lock is normally SYS, causing the system default to be used. (The system default is typically EXCL for "exclusive", meaning "n readers or 1 writer", as described above.) Changing the read/write lock of a file to UPDT (for "update") allows n readers and 1 writer to access the file simultaneously. Changing the read/write lock to NONE (no lock) allows n readers and m writers to access the file simultaneously.

See Chapter 12 for more information on the read/write lock.

▶ E$BPAR (6) Bad parameter.

Description to be supplied.

▶ E$NATT (7) No UFD attached.

Usually occurs only when the directory to which the user is attached is removed from the system, as when a disk is shut down. Use one of the AT$xxx subroutines, or the ATTACH or ORIGIN command, to re-establish a cache attach point.

▶ E$FDFL (8) Directory too large.

Description to be supplied.

▶ E$DKFL (9) The disk is full.

The operation requires an additional record to be allocated on a disk partition, but all records on that partition are already allocated. Use the AVAIL command to display the number of total and available records on a disk partition.

Some operations are nonrecoverable after returning this error code. For example, the WTLIN$ subroutine does not restore the file location pointer to the original location when it encounters this error; the file location is undefined. Other operations, such as the PRWF$$ subroutine, reset the file location pointer to the value it held before the disk-full error was encountered.

When designing programs that manipulate data bases, make sure you design them to handle disk-full and quota-exceeded conditions correctly, by performing appropriate cleanup before actually returning the error code to the calling program or to the user.

▶ E$NRIT (10) Insufficient access rights.

Description to be supplied.

▶ E$FDEL (11) File open on delete.

An attempt to delete a file directory or a segment directory failed because the directory was in use by another user or by the same user on another file unit.

▶ E$NTUD (12) Not a UFD.

The attempted operation requires the target file system object to be a file directory, but it is not a file directory. This error is returned by the following operations:

- Attach
- Set quota (Q$SET)
- Check for acl vs. non-acl (ISACL$)
- Read directory entry (DIRRD, ENTRD, RDEN$$)

▶ E$NTSD (13) Not a segment directory.

The attempted operation requires the target file system object to be a segment directory, but it is not a segment directory.

▶ E$DIRE (14) Operation illegal on directory.

The file being truncated is a segment directory. Segment directories can be truncated using only SGDR$$.

▶ E$FNTF (15) Not found.

Description to be supplied.

▶ E$FNTS (16) Not found in segment directory.

Description to be supplied.

▶ E$BNAM (17) Illegal name.

Description to be supplied.

▶ E$EXST (18) Already exists.

Description to be supplied.

▶ E$DNTE (19) The directory is not empty.

Description to be supplied.

▶ E$SHUT (20) Bad shutdown attempted.

Description to be supplied.

▶ E$DISK (21) Disk I/O error.

Description to be supplied.

▶ E$BDAM (22) — Bad DAM file.

Description to be supplied.

▶ E$PTRM (23) — Pointer mismatch found.

Description to be supplied.

▶ E$BPAS (24) — Bad password.

Description to be supplied.

▶ E$BCOD (25) — Bad code in error vector.

Description to be supplied.

▶ E$BTRN (26) — Bad truncate of segment directory.

Description to be supplied.

▶ E$OLDP (27) — Old partitions not supported.

Description to be supplied.

▶ E$BKEY (28) — Bad key in call.

Description to be supplied.

▶ E$BUNT (29) — Bad unit number.

The file unit number supplied is not a valid file unit number. Note that file units 1-128 are always valid unit numbers (unless the System Administrator has drastically reduced the number of valid file units by using the FILUNT directive in the system configuration file). Larger file units may become valid as a user uses more dynamically allocated units. File unit numbers less than 1 are invalid in most cases.

▶ E$BSUN (30) Bad segment directory unit.

Description to be supplied.

▶ E$SUNO (31) Segment directory unit not open.

Description to be supplied.

▶ E$NMLG (32) Name is too long.

Description to be supplied.

▶ E$SDER (33) Segment directory error.

Description to be supplied.

▶ E$BUFD (34) The directory is damaged.

Description to be supplied.

▶ E$BFTS (35) Buffer too small.

Description to be supplied.

▶ E$FITB (36) The file is too long.

Description to be supplied.

▶ E$NULL (37)

Description to be supplied.

▶ E$IREM (38) Illegal remote reference.

Description to be supplied.

▶ E$DVIU (39) The device is in use.

Description to be supplied.

▶ E$RLDN (40) The remote line is down.

The system on which the file resides cannot be reached from the local system. Therefore, no disks on that remote system can be accessed.

▶ E$FUIU (41) All file units in use.

No more file units are available for the calling process. This usually indicates that the program is not closing units it has finished using, since the number of available file units is usually very large.

This error may also indicate that a remote system being used by the calling process has run out of file units on which to handle this process's remote requests.

▶ E$DNS (42) Device is not started.

Description to be supplied.

▶ E$TMUL (43) Too many subdirectory levels.

Description to be supplied.

▶ E$FBST (44) FAM - bad startup.

Description to be supplied.

▶ E$BSGN (45) Invalid segment number.

Description to be supplied.

▶ E$FIFC (46) Invalid FAM function code.

Description to be supplied.

▶ E$TMRU (47)	Maximum remote users exceeded.

Description to be supplied.

▶ E$NASS (48)	Device not assigned.

Description to be supplied.

▶ E$BFSV (49)	Bad FAM SVC.

Description to be supplied.

▶ E$SEMO (50)	Semaphore overflow.

Description to be supplied.

▶ E$NTIM (51)	No timer.

Description to be supplied.

▶ E$FABT (52)	FAM - aborted.

Description to be supplied.

▶ E$FONC (53)	FAM - operation not complete.

Description to be supplied.

▶ E$NPHA (54)	No phantoms are available.

Description to be supplied.

▶ E$ROOM (55)	No room.

Description to be supplied.

▶ E$WTPR (56) — Disk is write-protected.

On a write-protected disk, a file can neither be opened for writing nor be created. (A disk is write-protected by using the ADDISK command, described in the System Operator's Guide, Volume II.)

▶ E$ITRE (57) — Illegal treename.

This indicates that the pathname supplied to TSRC$$ does not conform to the syntax rules for a pathname. See the Prime User's Guide for a description of the syntax of a pathname.

▶ E$FAMU (58) — FAM - already in use.

Description to be supplied.

▶ E$TMUS (59) — Max number of users exceeded.

Description to be supplied.

▶ E$NCOM (60) — Null command line.

Description to be supplied.

▶ E$NFLT (61) — Unable to find fault frame.

Description to be supplied.

▶ E$STKF (62) — Bad stack format.

Description to be supplied.

▶ E$STKS (63) — Bad stack format signalling.

Description to be supplied.

▶ E$NOON (64) — No on-unit found.

Description to be supplied.

▶ E$CRWL (65) — Fatal error in crawlout.

Description to be supplied.

▶ E$CROV (66) — Stack overflow in crawlout.

Description to be supplied.

▶ E$CRUN (67) — Crawlout unwind failed.

Description to be supplied.

▶ E$CMND (68) — Bad command format.

Description to be supplied.

▶ E$RCHR (69) — Reserved character.

Description to be supplied.

▶ E$NEXP (70) — Bad use of EXIT.

Description to be supplied.

▶ E$BARG (71) — Invalid argument to command.

Description to be supplied.

▶ E$CSOV (72) — Concealed stack overflow.

Description to be supplied.

▶ E$NOSG (73) — Segment does not exist.

Description to be supplied.

▶ E$TRCL (74) Command line truncated.

Description to be supplied.

▶ E$NDMC (75) No SMLC DMC channels.

Description to be supplied.

▶ E$DNAV (76) Device not available.

Description to be supplied.

▶ E$DATT (77) Device already attached.

Description to be supplied.

▶ E$BDAT (78) Bad output data.

Description to be supplied.

▶ E$BLEN (79) Bad length.

Description to be supplied.

▶ E$BDEV (80) Bad device number.

Description to be supplied.

▶ E$QLEX (81) Queue length exceeded.

Description to be supplied.

▶ E$NBUF (82) No buffer space.

Description to be supplied.

▶ E$INWT (83) Input waiting.

Description to be supplied.

▶ E$NINP (84) No input available.

Description to be supplied.

▶ E$DFD (85) Device forcibly detached.

Description to be supplied.

▶ E$DNC (86) DPTX not configured.

Description to be supplied.

▶ E$SICM (87) Illegal 3270 command.

Description to be supplied.

▶ E$SBCF (88) Bad copy FROM device number.

Description to be supplied.

▶ E$VKBL (89) Keyboard locked.

Description to be supplied.

▶ E$VIA (90) Invalid AID byte.

Description to be supplied.

▶ E$VICA (91) Invalid cursor address.

Description to be supplied.

▶ E$VIF (92) Invalid field address.

Description to be supplied.

▶ E$VFR (93) Field required.

Description to be supplied.

▶ E$VFP (94) Field prohibited.

Description to be supplied.

▶ E$VPFC (95) Protected field check.

Description to be supplied.

▶ E$VNFC (96) Numeric field check.

Description to be supplied.

▶ E$VPEF (97) Past end of field.

Description to be supplied.

▶ E$VIRC (98) Invalid read mod character.

Description to be supplied.

▶ E$IVCM (99) Invalid command.

Description to be supplied.

▶ E$DNCT (100) Device not connected.

Description to be supplied.

▶ E$BNWD (101) Bad number of words.

Description to be supplied.

▶ E$SGIU (102) Segment in use.

Description to be supplied.

▶ E$NESG (103) Not enough segments.

Description to be supplied.

▶ E$SDUP (104) Duplicate segment number.

Description to be supplied.

▶ E$IVWN (105) Invalid VMFA window number.

Description to be supplied.

▶ E$WAIN (106) Window already in address space.

Description to be supplied.

▶ E$NMVS (107) No more VMFA segments.

Description to be supplied.

▶ E$NMTS (108) No more temp segments.

Description to be supplied.

▶ E$NDAM (109) Not a DAM file.

Description to be supplied.

▶ E$NOVA (110) Not open for VMFA.

Description to be supplied.

▶ E$NECS (111) Not enough contiguous segments.

Description to be supplied.

▶ E$NRCV (112) Requires receive enabled.

Description to be supplied.

▶ E$UNRV (113) User not receiving now.

Description to be supplied.

▶ E$UBSY (114) User busy, please wait.

Description to be supplied.

▶ E$UDEF (115) User unable to receive messages.

Description to be supplied.

▶ E$UADR (116) Unknown addressee.

Description to be supplied.

▶ E$PRTL (117) Operation partially blocked.

Description to be supplied.

▶ E$NSUC (118) Operation unsuccessful.

Description to be supplied.

▶ E$NROB (119) No room in output buffer.

Description to be supplied.

▶ E$NETE (120) Network error detected.

This indicates that a problem with a remote file access has occurred. It is possible that trying the operation again might be successful. If not, it may be necessary for the user to close all remote file units on the remote system and to issue the ORIGIN command before retrying the remote access.

▶ E$SHDN (121) Disk has been shut down.

The disk on which the file resides has been shut down (using the SHUTDN command as described in the System Operator's Guide, Volume II). The disk is no longer available for use, until the system operator uses the ADDISK command to add the disk again.

▶ E$UNOD (122) Unknown node (PRIMENET).

Description to be supplied.

▶ E$NDAT (123) No data found.

Description to be supplied.

▶ E$ENQD (124) Enqueued only.

Description to be supplied.

▶ E$PHNA (125) Protocol handler not available.

Description to be supplied.

▶ E$IWST (126) E$INWT enabled in config.

Description to be supplied.

▶ E$BKFP (127) Bad key for this protocol.

Description to be supplied.

▶ E$BPRH (128) Bad PH specified in config.

Description to be supplied.

▶ E$ABTI (129) I/O abort in progress.

Description to be supplied.

▶ E$ILFF (130) Illegal DPTX file format.

Description to be supplied.

▶ E$TMED (131) Too many emulate devices.

Description to be supplied.

▶ E$DANC (132) DPTX already configured.

Description to be supplied.

▶ E$NENB (133) Remote node not enabled.

Description to be supplied.

▶ E$NSLA (134) No NPX slaves available.

The remote system on which the file resides has become overloaded with remote file access requests. The operation may be attempted later, with possible success.

▶ E$PNTF (135) Procedure not found.

Description to be supplied.

▶ E$SVAL (136) Slave validation error.

The user's remote ID for the system on which the file resides is incorrect. The user must use the ADD_REMOTE_ID command, described in the PRIMOS Commands Reference Guide, to establish the correct remote ID for the system. Until then, all attempts to access data on that remote system will fail with this error code.

▶ E$IEDI (137) I/O error or device interrupt.

Description to be supplied.

▶ E$WMST (138) Warm start occurred.

Description to be supplied.

▶ E$DNSK (139) Pio instruction did not skip.

Description to be supplied.

▶ E$RSNU (140) Remote system not up.

The remote system on which the file resides is in the process of starting up, but is not yet honoring remote file access requests. A remote system honors remote file access requests once the operator SETIME command has been issued at the supervisor terminal for that system. See the System Operator's Guide, Volume II for details on the SETIME command.

▶ E$S18E (141)

Description to be supplied.

▶ E$NFQB (142) No free quota blocks.

Description to be supplied.

▶ E$MXQB (143)	Maximum quota exceeded.
Description to be supplied.	
▶ E$NOQD (144)	Not a quota disk.
Description to be supplied.	
▶ E$QEXC (145)	Quota set below current usage.
Description to be supplied.	
▶ E$IMFD (146)	Operation illegal on MFD.
Description to be supplied.	
▶ E$NACL (147)	Not an ACL directory.
Description to be supplied.	
▶ E$PNAC (148)	Parent not an ACL directory.
Description to be supplied.	
▶ E$NTFD (149)	Not a file or directory.
Description to be supplied.	
▶ E$IACL (150)	Entry is an access category.
Description to be supplied.	
▶ E$NCAT (151)	Not an access category.
Description to be supplied.	

▶ E$LRNA (152) Cannot access like reference.

Description to be supplied.

▶ E$CPMF (153) Category protects MFD.

Description to be supplied.

▶ E$ACBG (154) ACL too big.

Description to be supplied.

▶ E$ACNF (155) Access category not found.

Description to be supplied.

▶ E$LRNF (156) Like reference not found.

Description to be supplied.

▶ E$BACL (157) Incorrect access control list format.

Description to be supplied.

▶ E$BVER (158) Incorrect version number.

Description to be supplied.

▶ E$NINF (159) No information.

Description to be supplied.

▶ E$CATF (160) Directory still contains access categories.

Description to be supplied.

▶ E$ADRF (161) Directory still contains ACL subdirectories.

Description to be supplied.

▶ E$NVAL (162) Validation error.

Description to be supplied.

▶ E$LOGO (163)

Description to be supplied.

▶ E$NUTP (164) No unit table for phantom.

Description to be supplied.

▶ E$UTAR (165) Unit table already returned.

Description to be supplied.

▶ E$UNIU (166) Unit table not in use.

Description to be supplied.

▶ E$NFUT (167) No unit table available.

Description to be supplied.

▶ E$UAHU (168) User already has unit table.

Description to be supplied.

▶ E$PANF (169) Priority ACL not found.

Description to be supplied.

▶ E$MISA (170) Missing argument to command.

Description to be supplied.

▶ E$SCCM (171) System console command only.

Description to be supplied.

▶ E$BRPA (172) Bad remote password.

Description to be supplied.

▶ E$DTNS (173) Date and time not set.

Description to be supplied.

▶ E$SPND (174) Remote procedure call still pending.

Description to be supplied.

▶ E$BCFG (175) Network config. mismatch or slave validation error

The remote system on which the file resides does not agree with the network configuration of the local system. See the System Administrator's Guide, or your System Administrator, for assistance.

▶ E$BMOD (176) Illegal access mode.

Description to be supplied.

▶ E$BID (177) Illegal identifier.

Description to be supplied.

▶ E$ST19 (178) Disk format does not support this revision of PRIMOS.

Description to be supplied.

▶ E$CTPR (179) Object is category-protected.

Description to be supplied.

▶ E$DFPR (180) Object is default-protected.

Description to be supplied.

▶ E$DLPR (181) File is delete-protected.

Description to be supplied.

▶ E$BLUE (182) Bad LUBTL entry.

Description to be supplied.

▶ E$NDFD (183) No driver for device.

Description to be supplied.

▶ E$WFT (184) Wrong file type.

Description to be supplied.

▶ E$FDMM (185) Format/data mismatch.

Description to be supplied.

▶ E$FER (186) Bad format.

Description to be supplied.

▶ E$BDV (187) Bad dope vector.

Description to be supplied.

▶ E$BFOV (188) F$IOBF overflow.

Description to be supplied.

▶ E$NFAS (189) Top-level directory not found or inaccessible.

The first directory name supplied in the pathname could not be located on any of the system disks.

▶ E$APND (190) Asynchronous procedure still pending.

Description to be supplied.

▶ E$BVCC (191) Bad virtual circuit clearing.

Description to be supplied.

▶ E$RESF (192) Improper access of restricted file.

Description to be supplied.

▶ E$MNPX (193) Illegal multiple hops in NPX

Description to be supplied.

▶ E$SYNT (194) SYNTanx error

Description to be supplied.

▶ E$USTR (195) Unterminated STRing

Description to be supplied.

▶ E$WNS (196) Wrong Number of Subscripts

Description to be supplied.

▶ E$IREQ (197) Integer REQuired

Description to be supplied.

▶ E$VNG (198) Variable Not in namelist Group

Description to be supplied.

▶ E$SOR (199) Subscript Out of Range

Description to be supplied.

▶ E$TMVV (200) Too Many Values for Variable

Description to be supplied.

▶ E$ESV (201) Expected String Value

Description to be supplied.

▶ E$VABS (202) Variable Array Bounds or Size

Description to be supplied.

▶ E$BCLC (203) Bad Compiler Library Call

Description to be supplied.

▶ E$NSB (204) NSB labelled tape was detected

Description to be supplied.

▶ E$WSLV (205) Slave ID mismatch.

Description to be supplied.

▶ E$VCGC (206) Virtual circuit got cleared.

Description to be supplied.

▶ E$MSLV (207) Exceeds the MAX number of slaves/user.

Description to be supplied.

▶ E$IDNF (208) Slave ID number not found.

Description to be supplied.

▶ E$NACC (209) Not accessible.

Description to be supplied.

▶ E$UDMA (210) Not Enough DMA channels.

Description to be supplied.

▶ E$UDMC (211) Not Enough DMC channels.

Description to be supplied.

▶ E$BLEF (212) Bad tape record length and EOF.

Description to be supplied.

▶ E$BLET (213) Bad tape record length and EOT.

Description to be supplied.

▶ E$ALSZ (214)	ALLOCATE request too small.

Description to be supplied.

▶ E$FRER (215)	FREE request with invalid pointer.

Description to be supplied.

▶ E$HPER (216)	User storage heap is corrupted.

Description to be supplied.

▶ E$EPFT (217)	Invalid EPF type.

Description to be supplied.

▶ E$EPFS (218)	Invalid EPF search type.

Description to be supplied.

▶ E$ILTD (219)	Invalid EPF LTD linkage descriptor.

Description to be supplied.

▶ E$ILTE (220)	Invalid EPF LTE linkage descriptor.

Description to be supplied.

▶ E$ECEB (221)	Exceeding command environment breadth.

Description to be supplied.

▶ E$EPFL (222)	EPF file exceeds file size limit.

Description to be supplied.

▶ E$NTA (223) EPF file not active for this user.

Description to be supplied.

▶ E$SWPS (224) EPF file suspended within program session.

Description to be supplied.

▶ E$SWPR (225) EPF file suspended within this process.

Description to be supplied.

▶ E$ADCM (226) System Administrator command only.

Description to be supplied.

▶ E$UAFU (227) Unable to allocate file unit

Description to be supplied.

▶ E$FIDC (228) File inconsistent data count

Description to be supplied.

▶ E$INDL (229) Insufficient Dam index levels

Description to be supplied.

▶ E$PEOF (230) Past End Of File

Description to be supplied.