HONEYWELL

CP-6 SYSTEM PROGRAMMER GUIDE

SOFTWARE

CONTROL PROGRAM-SIX (CP-6)

SYSTEM PROGRAMMER GUIDE

SUBJECT

Description of System Programming Practices and Examples of System Programming Techniques

SOFTWARE SUPPORTED

Software Release B03

ORDER NUMBER

CE62-00

January 1984

Honeywell

Preface

This guide contains descriptions of system programming practices and provides numerous examples of system programming techniques. This guide is intended for readers already familiar with the CP-6 PL-6 Reference Manual (CE44) and the CP-6 Monitor Services Reference Manual (CE33). In addition, the Honeywell training courses are an important resource for the system programmer.

The Los Angeles Development Center (LADC) of Honeywell Information Information Systems has developed Computer Aided Publications (CAP). CAP is an advanced text processing system providing automatic table of contents, automatic indexing, format control, automatic output of camera-ready masters, and other features. This manual is a product of CP-6 CAP.

Readers of this document may report errors or suggest changes through a STAR on the CP-6 STARLOG system. Prompt response is made to any STAR against a CP-6 manual; changes will be incorporated into subsequent releases and/or revisions of the manual. If high impact errors are discovered, patches to this manual may be issued and distributed via communication lines.

The information in this publication is believed to be accurate in all respects. However, Honeywell Information Systems cannot assume responsibility for any consequences resulting from the use of this document. The information contained in this manual is subject to change. New editions of this publication may be issued to incorporate such changes. The latest version of this manual may be ordered from

Honeywell Information Systems Inc.
National Distribution Operation
47 Harvard Street
Westwood, Massachusetts 02090

Telephones: Customers (617) 392-5235
Honeywell (HVN) 273-5215 (HED MA06)

 File No.: 1W13 CE62-00

Contents

Contents (cont)

Contents (cont)

Contents (cont)

Contents (cont)

TABLES

FIGURES

Contents (cont)

About This Manual

The contents of the manual are grouped into fifteen sections:

Section 1, Introduction, highlights key concepts that are essential to the system programmer.

Section 2, PL-6 for System Programming, includes an annotated example of the use of monitor services in PL-6 programs.

Section 3, X Account Tools, discusses tools available to the system programmer.

Section 4, Processor Conventions, describes guidelines which help to provide consistency in the user interface to the CP-6 system.

Section 5, Source Code Documentation, describes conventions and tools for including extractable commentary in source code files.

Section 6, Error Message Reporting, describes conventions and tools for error reporting and creation of error message files.

Section 7, User Documentation/HELP, describes the $TEXT facility which simplifies and standardizes creation of user manuals and on-line (HELP) documentation.

Section 8, Techniques: Central System, describes interfaces to central system software.

Section 9, Techniques: Communications, describes interfaces to communication software.

Section 10, Shared Run Units, describes sharing programs and the auto-sharing process.

Section 11, Special Shared Processors, describes guidelines for writing Command Processors, Debuggers, and Alternate Shared Libraries.

Section 12, Run-Time Libraries, describes how to build and associate run-time libraries.

Section 13, Library Functions, describes functions available from the :SHARED_SYSTEM run-time library.

Section 14, Compilers and Language Utilities, describes conventions and tools to aid in writing language processors.

Section 15, Interlanguage Calling, describes calling and receiving sequences and related information.

Appendix A, Job Information Table, describes the JIT information maintained for all active users of the system.

Notation Conventions

The following table gives notation conventions used in this manual to describe commands, statements, directives, and other language elements.

Notation Conventions Table

Notation	Description
Lower-case Letters	Lower-case letters indicate that the element is a variable, to be replaced with the desired value.
CAPITAL LETTERS	Capital letters indicate a literal, to be entered as shown.
Special Characters	Special characters are literals, to be entered as shown.
Numerals	Numerals standing alone are literals, to be entered as shown. Numerals embedded in or affixed to a string of capital letters are also literals, to be entered as shown, for example, PL6. Numerals embedded in or affixed to a string of lower case letters are part of the variable name to be replaced with a desired value, for example, fid1.
Braces	Elements stacked inside a pair of braces identify a required choice. The braces may be elongated to contain the possible choices, or may be represented by vertically-stacked printed braces. `{shift_count}` means that either a value for `{R1}` shift_count or the word R1 must be entered. Alternatively, the vertical OR bar is used to separate the choices, thus: `{shift_count|R1}`
OR Bar	The OR bar separates elements in a list from which one element may be, or must be, chosen.

Notation Conventions Table (cont.)

Notation	Description
{R1|shift_count}	means that either the word R1 or the value of shift_count must be entered.
Vertical Ellipsis	
	The vertical ellipsis indicates that zero or more commands or instructions have been omitted. START CCP . means that there are zero or more . statements omitted between the . CCP and END statements. END
Blanks	
	Where blanks are shown in syntax formats, one or more blanks must be used except inside character literal strings.

Section 1
Introduction

Interfaces and their use and manipulation are major strengths of the CP-6 system. A guiding goal in implementing the CP-6 system was to build a system which did not need to be modified by the users or by Honeywell in order to adapt to special requirements. The system has been successful in this effort as judged by the exceptionally small number of site specific patches to the system and the lack of changes by addition and recompilation of system or processor modules.

System programmers do change the way the system operates, however, largely through use and understanding of the interfaces documented in this guide. There are three major interfaces to the CP-6 system described in this guide:

1. The user interface
2. The compiler output or direct language interface
3. The program calling interface

File management, terminal and device I/O interfaces which are equally important in using the system are described in the CP-6 Programmer Reference Manual (CE40) and the CP-6 Monitor Services Reference Manual (CE33).

The CP-6 system presents a common, familiar, and consistent interface to the end-user. Standards used throughout the CP-6 system and in all its processors provide a consistent syntax; the same options mean the same thing throughout the system. The CP-6 parsing tools and standards are described here so that newly built processors can present the same consistent user interface. Many of the tools employed in the development of the CP-6 system are made available with the system in account X. A description of the most important and useful of the account X routines are described in Section 3.

The standard object language provides a form for all compilers to produce loadable code and the debugging schema. Tools useful to compiler writers are also discussed.

Standards for calling between programs allow subprograms in several languages to be combined into a single program with each language being used where it is strongest.

CP-6 interfaces are also provided so that a site may create its own command language, its own debugger, or its own database management system. Each of these is a type of shared program called a special shared processor. The rules for their creation and the privileged calls available to them are described in the sections that follow. User and system libraries form another level of sharing in the CP-6 system. Rules governing these libraries (which may be amended or recreated by the user) are given together with techniques which, if used, will allow change of the libraries without the need to change, reload or recompile programs which use them.

CP-6 comgroups are a unique method for communicating between programs, and between programs and terminals or devices connected to the system. Comgroups form an internal message switching network within the CP-6 system.

Examples throughout this volume will help the system programmer to make new and innovative use of the CP-6 system through its interfaces.

Section 2

PL-6 for System Programming

PL-6 is the implementation language for the CP-6 system. The PL-6 language, which is intended for block-structured programming, provides simple syntax and simple data types. See the CP-6 PL-6 Reference Manual (CE44) for a complete definition of the language. The first part of the Reference discusses concepts of the language and its main features.

PL-6 users must rely on monitor services, as documented in the CP-6 Monitor Services Reference Manual (CE33), for functions which the PL-6 compiler does not provide. The PL-6 CALL statement may be used to transfer control to any monitor service or library service. In addition to the service subroutines, the CP-6 system provides macro definition for related structures which are INCLUDEd for use in the PL-6 program.

To use a monitor service, the PL-6 program simply calls the needed service. Every monitor service has no more than one argument. The argument is the name of the Function Parameter Table, (FPT) which contains parameters specific to each monitor service. The program generates the FPT by using the macro structures supplied in the system macro library. (See the CP-6 Monitor Services Reference, CE33, for more information on the system macro library).

Figure 2-1 illustrates a program which uses several common monitor services and demonstrates the use of the system macros to set up the FPTs for the monitor services. The sample program is purposely brief. A somewhat longer sample program is illustrated in the CP-6 Monitor Services Reference Manual. PL-6 program fragments are shown throughout this handbook. In addition, programs in account .:XSI are a source of examples, as discussed in Section 3.

```
/*
     Program to REKEY a file, starting with a key of 1.000 in
     increments of 1.000
*/
REKEY: PROC MAIN;

                                   /*
                                        Locally needed substitution
                                        strings (%EQUs)
                                                                   */
%EQU TRUE#='1'B /*TRUE#*/;
%EQU FALSE#='0'B /*FALSE#*/;

                                   /*
                                       INCLUDE all the system macros
                                       so we can set up the FPTs for
                                       the monitor services.
                                                                   */
     %INCLUDE CP_6;
     %INCLUDE CP_6_SUBS;

                                   /*
                                       DCBs defined here
                                                                   */
DCL M$SI DCB;
DCL M$OO DCB;
DCL M$DO DCB;

                                   /*
                                       EXTERNALs
                                                                   */
DCL B$TCB$ PTR SYMREF;


                                   /*
                                       Local STATIC storage
                                                                   */
DCL BUF1 CHAR(256) STATIC INIT(' ');
DCL BUF1ARS UBIN WORD STATIC;



DCL M$SI$ PTR STATIC;
DCL M$OO$ PTR STATIC;

DCL 1 OO_KEY STATIC,
        2 * UBIN BYTE CALIGNED INIT(3),
        2 KEY_ UBIN(27) CALIGNED INIT(0);

DCL MY_ERROR_BUF CHAR(255) STATIC CALIGNED;

DCL FPARAM_BUF(0:1025) SBIN STATIC ALIGNED;

   /*
        Invoke the macros that got INCLUDEd in CP_6 to set up
        the information needed by the monitor services.
                                                                   */

     %FPT_ERRMSG    (FPTN=ERROR_PRINT,
                     BUF=MY_ERROR_BUF,
                     DCB=M$SI,
                     OUTDCB1=M$DO,
                     CODE=NIL);
```

Figure 2-1. Sample PL-6 Program (cont. next page)

```
     %FPT_OPEN       (FPTN=OPEN_SI_IN,
                          FPARAM=FPARAM_BUF,
                          DCB=M$SI);

     %FPT_OPEN       (FPTN=OPEN_00_OUT,
                          DCB=M$00,
                          FUN=CREATE,
                          EXIST=NEWFILE,
                          ASN=FILE,
                          ACS=DIRECT,
                          IFPARAM=FPARAM_BUF,
                          ORG=KEYED);

     %FPT_READ       (FPTN=READ_BUF1,
                          DCB=M$SI,
                          BUF=BUF1,
                          WAIT=YES);

     %FPT_WRITE      (FPTN=WRITE_00_BUF2,
                          DCB=M$00,
                          BUF=BUF1,
                          ONEWKEY=YES,
                          KEY=00_KEY,
                          WAIT=YES);

     %FPT_CLOSE      (FPTN=CLOSE_SI_REL,
                          DISP=RELEASE,
                          DCB=M$SI);

     %FPT_CLOSE      (FPTN=CLOSE_00_SAVE,
                          DISP=SAVE,
                          DCB=M$00);



                                /*
                                    Invoke the MACROs to generate
                                    based structure for accessing
                                    data in the TCB, DCB and ALTRET
                                    frame.
                                                                  */
     %F$DCB;
     %B$TCB;

     %B$ALT;


%EJECT;

     M$SI$ = DCBADDR(DCBNUM(M$SI));
     M$00$ = DCBADDR(DCBNUM(M$00));

                                /*
                                    Set up the file name in the
                                    OPEN FPT and call the monitor
                                    service routines to open the
                                    necessary files.
                                                                  */
     OPEN_00_OUT.NAME_ = VECTOR(M$SI$->F$DCB.NAME#);

     CALL M$OPEN (OPEN_SI_IN) ALTRET (MXXX);

     CALL M$OPEN (OPEN_00_OUT) ALTRET (MXXX);
```

Figure 2-1. Sample PL-6 Program (cont. next page)

```
                                /*
                                    READ record, update the key,
                                    WRITE it out and so on....
                                    until we hit the end of file.
                                                                */

     DO WHILE (%TRUE#);

          BUF1 = ' ';
          CALL M$READ (READ_BUF1) ALTRET (MUST_BE_END_OF_FILE);

          BUF1ARS = M$SI$->F$DCB.ARS#;

          00_KEY.KEY_ = 00_KEY.KEY_ + 1000;

          WRITE_00_BUF2.BUF_.BOUND = BUF1ARS;

          CALL M$WRITE (WRITE_00_BUF2) ALTRET (MXXX);

        END;                              /* DO WHILE TRUE# */

                                /*
                                    An error occurred somewhere,
                                    get the error code and print
                                    the error message.
                                                                */
MXXX: ;
     ERROR_PRINT.CODE_ = VECTOR(B$TCB$->B$TCB.ALT$->B$ALT.ERR);

     CALL M$ERRMSG (ERROR_PRINT) ALTRET (HMMM);

HMMM: ;        /*  Abort this job, something went wrong  */

     CALL M$XXX;

MUST_BE_END_OF_FILE:;  /*  So close the open files and exit. */


     CALL M$CLOSE (CLOSE_00_SAVE) ALTRET (MXXX);

     CALL M$CLOSE (CLOSE_SI_REL) ALTRET (MXXX);

     CALL M$EXIT;

END REKEY;
```

Figure 2-1. Sample PL-6 Program

Section 3

X Account Tools

X Account Policy

The X Account provides a toolcrib for unsupported tools on the CP-6 system. Tools that are placed in this account may be of use to a small or large number of users. Any tool that is used by more than two individuals is a candidate for inclusion in the X Account.

The X Account tools provide users with programming examples of techniques and standards for using the CP-6 system. There are no access restrictions to the account; most of the source code used to create it is delivered to customer sites.

X Account Support Mechanisms

The X Account contains tools that are not supported through the normal support mechanisms provided with the CP-6 system. However, in the case of deficiency or oversight, or if a new feature is needed, a Severity D STAR may be submitted to STARLOG with the subject listed as Product X. There is no guarantee that the suggestion will be implemented in a timely manner, but the tool's originator may be able to provide help and advice through STARLOG responses.

X Account Naming Conventions

As released, the X Account tools are put in the .:XSI account which contains the source, HELP, JCL and rununits of all the tools. The job $BUILD_X_ACCT copies all the essential elements into the X Account. The following naming conventions are used for the .:XSI account.

- TOOL_SIcn (source)
- TOOL_HELP (HERMAN input)
- TOOL_CRU (Sample JCL)
- TOOL_Ccn (Include, copy files)

where c is a compiler indicator, and n is a digit.

NOTE: If files in the :XSI account appear not to exist, consult the system manager to determine if the account was created with appropriate access permissions granted.

Contents of the X Account

The X Account contains approximately 175 tools; the most frequently used tools are documented in this section. The X Account is constantly being changed and updated; therefore, tools may disappear or appear in the account at the different releases. The tools may be categorized as follows:

Programmer aids
System Programmer aids
Integration aids
Installation Management aids
Documentation aids
Support aids
Microprocessor Support aids
Miscellaneous tools

Programmer Aids

Programmer aids are tools that are of general use to the application programmer in all languages. These tools cover a wide spectrum of applications -- from tools that inform the user about the status of the system to tools that list HELP files.

Table 3-1. X Account Programmer Aids

Tool Name	Description
A	Records and reports the status of batched jobs.
BANNER	Prints user specified text in block letters on a line printer.
CALENDAR	Builds, displays and stores a user's personal calendar.
DI	Displays information about the system such as the number of users, the ETMF and 90% response time.
DILDEV	Displays a user's current Logical DEVices and how much output is queued up for each of them.
EJECT	Positions output to a new page.

Table 3-1. X Account Programmer Aids (cont.)

Tool Name	Description
EMU	The Error Message Uncoder prints the CP-6 error message text associated with a specified error code.
FIND	Searches account(s) for a given filename or prefix.
GOPHER	Displays the filename and lines within it that contain the user specified string.
LISTHELP	Lists one or more HELP files on the specified destination.
OVERLAP	Reads an FPL source program and then can be directed to check for overlapping fields and/or print one or more forms described in the program.
OX	Provides a cross reference of a FORTRAN 77 program and/or subroutines.
PMDISP	Displays Performance data gathered via PMON.X or PM.X.
PMON	Software Performance MONitor used with PMDISP.X and PM.X.
RQ	Displays information about the running or input/output queues.
SETUP	Is a universal setup program which eliminates the need to go through IBEX.
SKUNK	Locks your terminal and keeps someone from using it while you're away.

Table 3-1. X Account Programmer Aids (cont.)

Tool Name	Description
SL	Displays severity level of rununits and object units.

System Programmer Aids

System programmer aids are tools that are of use to programmers working in an environment that maintains a source base. These development tools can be used by a wide range of users on the CP-6 system. Many of these tools such as CMPR or LIN are used to maintain and update the source base.

Table 3-2. X Account System Programmer Aids

Tool Name	Description
AUTO	Allows a user to raise or lower the batch queue priority of subsequently batched jobs.
BOOKWORM	A program designed to aid in the electronic preparation of table of contents and indexes from TEXT files.
CMPR	Compares two files and generates update files.
DRAW	Converts PL-6 DCL statements to pictures, for use in debugging PL-6 structures, design specs, technical manuals, etc.
EDGEMARK	Prints specified text in block letters on the edge of a print-out.
EDICT	Puts extractable commentary into code.
FORMAT	Formats (i.e., pretty prints) PL-6 source files, merges updates and inserts copyright notices.

Table 3-2. X Account System Programmer Aids (cont.)

Tool Name	Description
KEYUP	Takes a file of plus records and gives them proper edit keys.
KEYER	Validates and rekeys plus-card files.
LIN	Merges lines from a base file and puts them into plus-card format.
LISTER	Copies selected portions of unit-record listing files (produced by PL-6, PL1, PARTRGE, BMAP, or GMAP6) to the line printer.
MODEL	Allows a user on a multiprocessor system made up of different CPU types to specify the CPU on which he wants to run.
PARSE/PARSEOU	Tools used with the parser and PARTRGE to tell the user what the output nodes look like after a parse.
PARTRGE	Creates parse node object units.
SDUMP	Dumps debug schema from an object unit file, run unit file or an overlayed run unit file.
UNGMAP	Takes an object unit and produces an assembly listing from it.
WHAT	Displays information about the current running system.

Integration Aids

CP-6 integration tools are tools used in integrating and distributing the CP-6 system.

Table 3-3. X Account Integration Aids

Tool Name	Description
DRAW	Converts PL-6 DCL statements to pictures, for use in debugging PL-6 programs, design specs, technical manuals, etc.
DTOR	Converts files containing PL6 DCL statements and pre-processor directives into files with corresponding SYMREF and/or BASED DCLs.
EDGEMARK	Prints specified text in block letters on the edge of a print-out.
EXTRACT	Extracts error messages and commentary from source code.
FICHER	Takes listing and source files and creates a set of tapes for printing on microfiche.
HERMAN	Reads a text file containing HELP and HERMAN commands and creates a HELP database.
INSREC	Inserts records from a base file into another file based on a control file.
LINKMOD	Alters LINK, PCL and/or LEMUR JCL.
LOOK4	Reports on multiple occurrences of update files.

Table 3-3. X Account Integration Aids (cont.)

Tool Name	Description
MODMOVE	Controls the manipulation of update files in controlled accounts.
MPUR	Removes unwanted schema from OBJECT and RUN units.
STI	Places software technical identifiers into released software.

Installation Management Aids

Installation management aids are tools that are useful to CP-6 system managers. These tools help manage the machine efficiently by giving information about the state of the machines such as what users have certain privileges or which remote terminals are connected.

Table 3-4. X Account Installation Management Aids

Tool Name	Description
AUTO	Allows a user to raise or lower the batch queue priority of subsequently batched jobs.
COBWEB	A tool that installs and deletes shared processors.
EXPIRED	Prints the names of files which have expired as of the current date.
FWEDITOR	Builds and edits a customized firmware file from an IFAD tape.
GRAMPS	Watches for disk packs that are running out of space.

Table 3-4. X Account Installation Management Aids (cont.)

Tool Name	Description
MPCDUMP	Provides a hexadecimal dump of an MPC's main memory separated by its memory content headings.
PRIVCHECK	Checks running users privileges against the privileges for which they were authorized.
PRIVDISP	Displays the logon id of users who have the requested privileges.
SPY	Displays certain information about current users on a CP-6 system.
ST	Aids in analyzing performance by displaying certain fields from the specified users' JIT.
TERM	Tells which remote terminals are connected or have output queued.
USERS	Converts user authorization files into newer versions.

Documentation Aids

These tools are, to the largest extent, used in documentation preparation along with CP-6 TEXT. They include a proofreading dictionary as well as a tool that creates indices.

Table 3-5. X Account Documentation Aids

Tool Name	Description
BOOKWORM	A program designed to aid in the electronic preparation of table of contents and indices from TEXT files.
EDICT	Puts extractable commentary into code.
EXTRACT	Extracts error messages and commentary from source code.
FIXTEXT	Strips leading and trailing blank lines from TEXT-produced output files, thus making them more suitable for use with other processors.
FORMAT	Formats (i.e., pretty prints) PL-6 source files, merges updates and inserts copyright notices.
HERMAN	Reads a text file containing HELP and HERMAN commands and creates a HELP database.
LISTHELP	Lists one or more HELP files on the specified destination.
NOBS	A program that reads a TEXT input file and changes backspaced/underscored passages into a format compatible with CP-6 FEP input functions.
PROOF	Is a document proofreader with an accompanying dictionary.

Table 3-5. X Account Documentation Aids (cont.)

Tool Name	Description
TUNA	A program to TUNe An edit-keyed text input file so that it can be edited on an 80 column CRT screen.
UNPRINT	Reads text files and reports on any unprintable characters found.

Development Management Aids

These are tools that support the system design as well as reporting system progress.

Table 3-6. X Account Development Management Aids

Tool Name	Description
CRF	Copy Review File. CRF is used to review files to which additional information may be appended at regular intervals.
LNCOUNT	Counts the number of comment and source lines in files in controlled accounts and reports this information to a Unit Record file or Terminal.

Support Aids

These tools provide a mechanism for both programmers and support personnel to support the software and customers.

Table 3-7. X Account Support Aids

Tool Name	Description
BEAM/MAEB	Transports files between CP-6 systems.
CGDUMP	Reads a closed comgroup file and any monitor dump file, and creates a dump file that can be used with ANLZ to look at the comgroup tables.
ELBBIRD	Converts files created by the DRIBBLE command from its original form to a form more easily used in documentation.
MOVE/SCOTTY	Transports files to and from other CP-6 systems.
PATCH	Formats patches. Inserts pertinent information such as the date, STAR number, etc.
RUMSPLIT	Takes a file containing RUM directives and splits it into smaller files, each containing a single product's RUMs.
TATTLE	Informs a Honeywell programmer when a test case has arrived in the ZZZTEST account.
WOODPECKER	Allows a user without the DISPJOB privilege to display all output destined for his Workstation of Origin.

Microprocessor Support Aids

These tools provide the user with several types of assemblers.

Table 3-8. X Account Microprocessor Support Aids

Tool Name	Description
APE	A program that provides the required handshaking for down-line loading of ASMZ80.X and ASM6502.X run units into a micro-processor.
ASM6502	A 6502 Cross-Assembler for CP-6.
MSA6800	A reverse assembler for 6800-based machine code.
MSA8085	A reverse assembler for 8085-based machine code.
MSAZ80	A reverse assembler for Z80-based machine code.

Miscellaneous Tools

These tools provide programs in the common tool crib which are of common interest and relatively high usage.

Table 3-9. X Account Miscellaneous Tools

Tool Name	Description
COPYPGM	Copies records, portions of records, or constant information from one file to another. Records may be copied based on Boolean criteria supplied by the user.
LOOK	Is a cross between EDIT and PCL with some extensions. It works with most file organizations and has no built in restrictions on maximum record lengths.

X Account Tool Invocation

The tool invocation command for tools in the X Account is:

!toolname.X [options]

For those tools that, because of their nature, reside in the :SYS account (i.e., SPY) the tool invocation command is:

!toolname [options]

HELP for X Account Tools

All of the tools that are in the X Account have HELP files. These files can be exercised by the following command:

!HELP (toolname.X)

for tools in the X Account.

!HELP (toolname)

for the tools that have been moved to the :SYS account.

X Account Programming Examples

The X account, besides being a tool crib for commonly used tools can provide PL-6 programming examples. Figure 3-1 shows a user examining the HELP file for a tool called TUNA.X.

```
!HELP (TUNA.X)
TUNA is a tool to TUNE A edit-keyed text input file so that it will be
RR-EDITable on an 80-column CRT screen.
!?
Full command line syntax:

!TUNA.X text_file [{ON|INTO|OVER} scratch_file] [(options[)]]

where:

text_file is a TEXT input file containing TEXT directives

scratch_file is an optional file name to be used for scratch area.  The
default scratch file is *G.  The scratch_file is automatically deleted
upon TUNA exit.
!?
Options are:

NWA/RN     don't print warning messages about unknown macros

NWR/AP     don't attempt to wrap text from one line to the next

NUS/BS     don't attempt to change the order of char/backspace/underscore
to underscore/backspace/characer

MA/XCOMPRESS try to compress all excess spaces out of a line except
those around the first word on an input record

LEN/GTH=line_length  sets the size of the records that TUNA will attempt
to produce.  The default is 69, which is right for editing on an
80-column CRT.  Allowable range is 50 through 132.

MAXCOMPRESS and NWRAP are mutually exclusive options.
```

Figure 3-1. Browsing through X Account HELP

The source code for the X account is stored in the :XSI account. The two source code files for TUNA can be used as a model for parsing commands input that contains a list of options. Figure 3-2 examines the TUNA program (TUNA_SI6.:XSI) which calls the X$PARSE library service and contains the required DCL statements needed to define structures to be passed to X$PARSE. Figure 3-3 demonstrates how the parse nodes for TUNA (in TUNA_SIN.:XSI) correspond to CASE statements in TUNA_SI6. In both figures underscoring is used to highlight what the user typed.

```
!L TUNA?.:XSI
TUNA_SI6    TUNA_SIN
 ..   2 files listed
!E TUNA_SI6.:XSI
EDIT B03 HERE
*   File TUNA_SI6.:XSI is open input - cannot update
*FTO-9999,/OPTIONS/OR/tion/
       58                                          CONSTANT definitions here
      118 (' **** NUSBS option automatically invoked.');
      135                                         OPTIONS flags
      315                                            BASED definitions
      350       DO;                       /* MUST BE OPTIONS ON CMD LINE */
      400          END;                   /* DO IF OPTIONS ON CMD LINE */
      407          END;                   /* DO IF CONFLICTING OPTIONS */
     1087 DCL STD_ERROR CHAR(0) STATIC INIT (' **** Bad option(s)');
     1088 DCL ADV_ERROR CHAR(0) STATIC INIT (' **** Conflicting options');
*   EOF hit after 1127
*TY350
      350       DO;                       /* MUST BE OPTIONS ON CMD LINE */
*TP9
      341     ************************************************************ */
      342
      343       M$SI$ = DCBADDR(DCBNUM(M$SI));
      344       M$OU$ = DCBADDR(DCBNUM(M$OU));
      345
      346       TUNA_PCB.ROOT$ = ADDR(TUNA_NODES);
      347
      348       IF B$JIT$->B$JIT.CCARS > B$JIT$->B$JIT.CCDISP
      349       THEN
*TY348
      348       IF B$JIT$->B$JIT.CCARS > B$JIT$->B$JIT.CCDISP
*"WHOA...THIS IS HOW TO KNOW THERE'RE OPTIONS ON THE COMMAND LINE!"
*TN8
      349       THEN
      350       DO;                       /* MUST BE OPTIONS ON CMD LINE */
      351
      352            TUNA_PCB.TEXT$ = PINCRC(ADDR(B$JIT.CCBUF),B$JIT.CCDISP+1);
      353            TUNA_PCB.NCHARS = B$JIT.CCARS - B$JIT.CCDISP - 1;
      354
      355            CALL X$PARSE (TUNA_PCB) ALTRET (XPERR);
      356
*TY352
      352            TUNA_PCB.TEXT$ = PINCRC(ADDR(B$JIT.CCBUF),B$JIT.CCDISP+1);
*"THIS IS HOW TO TELL THE PARSER WHERE TO FIND THE TEXT TO PARSE"
*TY353
      353            TUNA_PCB.NCHARS = B$JIT.CCARS - B$JIT.CCDISP - 1;
*"AND THIS IS TO TELL THE PARSER HOW MANY CHARACTERS TO PARSE"
```

Figure 3-2. Program Sample from :XSI Account - Part 1

```
*"WHERE'RE THE PARSE NODES?"
*E TUNA_SIN.:XSI
*    File TUNA_SIN.:XSI is open input - cannot update
*TY
        1 /*M* TUNA_SIN - Nodes for "TUNA" program. */
        2 /*T*************************************************************
        3  *T*                                                            *
        4  *T* COPYRIGHT, (C) HONEYWELL INFORMATION SYSTEMS INC., 1982 *
        5  *T*                                                            *
        6  *T*************************************************************/
        7 /*X* DMC,DFC */
        8
        9 TUNA_NODES(D,OUT)=(< ',',TUNA_CMDS > [')'] .END|NULL_CMD [')'] .END)
       10
       11 TUNA_CMDS = [.B] ( MAXCOMPRESS | ;
       12                    NWRAP | ;
       13                    NUSBS | ;
       14                    SIZE_OPT |;
       15                    NWARN )
       16
       17 MAXCOMPRESS(1)='MA/XCOMPRESS'
       18
       19 NWRAP(2)='NWR/AP'
       20
       21 NUSBS(3)='NUS/BS'
       22
       23 NWARN(4)='NWA/RN'
       24
       25 NULL_CMD(5) = ( .B | [.B] )
       26
       27 SIZE_OPT(6) = 'LEN/GTH' '=' .DEC3
*    EOF hit after 27
*"THERE'RE THE PARSE NODES! LOOKS A LOT LIKE THE HELP FILE!"
*
*" GO BACK TO SEE WHAT THE PARSER DOES"
*E TUNA_SI6.:XSI
*    File TUNA_SI6.:XSI is open input - cannot update
*TY353
      353            TUNA_PCB.NCHARS = B$JIT.CCARS - B$JIT.CCDISP - 1;
*TN23
      354
      355            CALL X$PARSE (TUNA_PCB) ALTRET (XPERR);
      356
      357            DO WHILE (FALSE#);
      358 XPERR:          ;
      359                 CALL PARSE_ERROR(1);
      360                 GOTO GET_GONE;
      361               END;                          /* DO WHILE PARSE ERROR */
      362
      363            DO I = 0 TO TUNA_PCB.OUT$ -> TUNA$OUTBLK.NSUBLKS - 1;
      364
      365                 DO CASE (TUNA_PCB.OUT$ -> TUNA$OUTBLK.SUBLK$(I) ->
      366                      TUNA$OUTBLK.CODE);
      367
      368                 CASE (1);
      369                    MAXCOMPRESS_ = TRUE#;
      370
      371                 CASE (2);
      372                    NWRAP_ = TRUE#;
      373
      374                 CASE (3);
      375                    NUSBS_ = TRUE#;
      376
*TY368
      368                 CASE (1);
*"HMMM...THIS CASE(1) MATCHES THE NUMBER ON THE NODE IN THE PARSE NODE FILE!"
*TY377-391
```

Figure 3-3. Program Sample from :XSI Account - Part 2 (cont. next page)

```
      377                  CASE (4);
      378                     NWARN_ = TRUE#;
      379
      380                  CASE (5);
      381
      382                  CASE (6);
      383                     CALL CHARBIN (MAX_PER_LINE,
      384                          TUNA_PCB.OUT$ -> TUNA$OUTBLK.SUBLK$(I) ->
      385                          TUNA$OUTBLK.SUBLK$(0) -> TUNA$SYM.TEXT);
      386
      387                     IF MAX_PER_LINE > LEN_LARGE#
      388                          OR
      389                          MAX_PER_LINE < LEN_SMALL#
      390                     THEN
      391                     DO;
*TY383-385
      383                     CALL CHARBIN (MAX_PER_LINE,
      384                          TUNA_PCB.OUT$ -> TUNA$OUTBLK.SUBLK$(I) ->
      385                          TUNA$OUTBLK.SUBLK$(0) -> TUNA$SYM.TEXT);
*"AND IT LOOKS LIKE I HAVE TO MANUALLY CONVERT PARSED DECIMAL NUMBERS THAT"
*" THE PARSER RETURNS INTO INTERNAL FORMAT TO...."
*TN10
      386
      387                     IF MAX_PER_LINE > LEN_LARGE#
      388                          OR
      389                          MAX_PER_LINE < LEN_SMALL#
      390                     THEN
      391                     DO;
      392                          CALL PARSE_ERROR(3);
      393                          GOTO GET_GONE;
      394                        END;              /* DO IF NUMBER TOO BIG */
      395
*"DO INTERNAL COMPARISONS WITH THEM!"
*END
```

Figure 3-3. Program Sample from :XSI Account - Part 2

Section 4

Processor Conventions

Conventions followed in writing Honeywell-supplied processors are discussed in this subsection. See Section 14 for conventions specifically applying to language processors.

General Case of Run Unit Invocation

A run unit is invoked via the IBEX command

!rununit [runinformation]

where

rununit is the disk fid specifying the run unit. Standard disk fid rules apply.

runinformation is additional text to be made available to the rununit in the user's JIT (B$JIT.CCBUF). The length of the text in CCBUF is set in B$JIT.CCARS. If the command includes options, B$JIT.CCDISP contains the index of the left parenthesis preceding the options. If there are no options, CCDISP is set to the same value as CCARS. If the command is continued, B$JIT.CCBUF contains only the first line of the command; the complete command (with any leading exclamation mark replaced by a blank) is written record by record to a star file, *CONTINUATION_COMMANDS; the flag B$JIT.PRFLAGS.CONTINUED is set to indicate continuation. The file contains the semicolons entered to show continuation, as well as any comments. (At job step, the file is deleted and the flag indicating continuation is reset.)

If the command is in the standard format (described in Section 14), the DCBs are set. If the run unit expects non-standard syntax, then it should be linked without the SIDCB, UIDCB, OUDCB, and LODCB link options.

The run unit is then called. If the run unit was linked with the STDINVOC option and the command is not in standard format, the run unit is aborted. Otherwise, it begins execution.

IBEX does not force the program invocation to be of the standard form; such enforcement would be needlessly restrictive. Rather, a flag B$JIT.PRFLAGS.NSSYNTAX is set (='1'B) if the program invocation is non-standard. The invocation line is made available to the program. The program may examine this flag and accept or reject the invocation, as desired.

Command Language Conventions

The following command language syntax rules and guidelines are considered standard. These rules are observed by the command language of IBEX, and by the command languages of the general use processors supplied for the CP-6 system by Honeywell, for example, EDIT and PCL.

1. Commands in the CP-6 common command language are not column dependent. Very few CP-6 commands are positional in the sense of having the significance of a parameter or operand denoted by its position in the command, except that some use is made of commands with a structure like simple conversational English, for example: "COPY sourcefile TO destfile."

2. Keywords can be typed in uppercase or lowercase or a combination of both. In CP-6 manuals, keywords are shown in uppercase.

3. Multiple parameters are often organized as lists, e.g., an optionlist is a parenthesis-delimited list of options, separated by commas.

4. Single parameters are connected to keywords in at least one of three ways:

   ```
   KEYWORD=parameter
   KEYWORD(parameter)
   KEYWORD=(parameter)
   ```

5. A list of more than one parameter is connected to a keyword in one of two ways.

   ```
   KEYWORD(parameter, parameter)
   KEYWORD=(parameter,parameter)
   ```

6. Comments within command streams are denoted by being enclosed in double quotation marks (").

7. Strings that contain delimiters must be enclosed by apostrophes ('); if an apostrophe is part of a string, it is denoted by adjacent apostrophes ('').

8. The semicolon is a continuation indicator for multiple line commands or the command separator for multiple commands on a single line. The semicolon cannot be used for continuation inside a keyword or text string.

Automatic File Extension

Under certain circumstances, automatic file extension occurs. File extension means the process of adding data to the end of a non-keyed file, or of merging data into a keyed file. Automatic file extension applies to the situation that an initial step (rununit) of a job or session has created a given file, and subsequent job steps also perform writes to that file. When automatic file extension applies, the writes to a given file from a subsequent job step are written at the end of the writes to that file from the previous step (or merged with them if it is a keyed file); it is automatic in the sense that there is no need for file management commands intervening between the steps to position the file cursor.

Performance of automatic file extension, at a given subsequent job step, of a particular file (fid) through its associated DCB, takes place under the following rules and conditions.

1. The writes to the file take place through an eligible DCB.

   Eligible DCBs are:

M$LO	M$SI	M$OU
M$LL	M$SO	M$EI
M$DO	M$UI	M$EO
M$PO		M$ME

2. After the first rununit is invoked, automatic file extension of a given fid through a given associated eligible DCB will take place unless or until:

   - A SET command is issued for the DCB, specifying a fid.
   - A RESET is done for this DCB, or a RESET ALL is done.

3. For writes through M$LO, automatic file extension is cancelled if a LIST command that specifies a fid is issued.

4. For writes through M$DO, automatic file extension is cancelled if a COMMENT command that specifies a fid is issued.

Explicit file extension may be used if the "EXIST=oldfile" option is used on the SET command, or if the "INTO" preposition on the rununit call is used.

File Type Codes

There is a convention concerning identification of the compiler or processor that created the object unit (or any other file) and the type of data incorporated in the file. The 2-character file type (ft) identifier codes shown in the Table 4-1 should be inserted in the FPT_OPEN.V.TYPE# field when the program is creating a file. The type code is provided for convenience and is not required for correct functioning of most system-supplied processors.

Table 4-1. File Type Codes

Codes	Meaning
First Character	
	D = Data I = Database O = Object unit R = Run unit S = Source U = Update W = Work space 1 = IDS schema or ARES model 2 = IDS subschema * = System file not modified by the user blank = undefined
Second Character	
	For Processor: A = APL a = ARES B = BASIC C = COBOL D = TRADER E = EDIT F = FORTRAN f = FPL G = GMAP6 I = IDS J = IMP K = reserved P = Performance Monitor or PARTRGE Q = IDP For Data: A = ASCII a = APL data block attributes B = ASCII and single precision c = APL component file D = Double precision S = Single precision blank = undefined or unformatted

Processor Termination Conventions

Processors which operate by reading and acting on commands input by the user (as opposed to compilers) should accept the 'END' command as sufficient cause to terminate and return control to IBEX (by issuing the M$EXIT monitor call).

Sample Interactive Processor

Several library services are available to provide a consistent user interface to interactive processors. To provide a simple, readily comprehensible example of these services, Figure 4-1 illustrates a sample processor that actually performs no useful work. Instead it highlights the mechanics of defining a command language, sample processing command input, providing assistance with command syntax, reporting error messages, and reporting HELP messages.

The sample processor, CORNER, consists of two modules of procedure. The main module (DEF$CORNER) solicits and parses commands, and reports syntax error messages and HELP messages. The internal procedure (DEG$BEAST) receives control for the CLEAN command, but for the sake of simplicity its only function is to report an error message. In addition, the sample processor contains a module of parse nodes (Figure 4-2) and a module of EQUs (Figure 4-3) which together define the command syntax for CORNER.

To demonstrate the user interface to the sample processor, Figure 4-4 shows an interactive session. Figure 4-5 shows the compilation, linking, and other steps necessary to run the sample program. The following subsections discuss various aspects of the sample processor.

```
/*M* DEF$CORNER - Main module for the CORNER processor.                */
CORNER: PROC MAIN;
/**/
/*T************************************************************
 *T*                                                           *
 *T* COPYRIGHT, (C) HONEYWELL INFORMATION SYSTEMS INC., 1983 *
 *T*                                                           *
 *T************************************************************/
/**/
%INCLUDE CP_6;
%INCLUDE DEP$FUZZY_E;                    /* PARSE NODE EQUS          */
%INCLUDE XU_MACRO_C;                     /* XUR MACROS               */
%INCLUDE XUH_MACRO_C;                    /* XUH MACROS               */
%INCLUDE XUR_ENTRY;                      /* XUR ENTRY DCLS           */
%INCLUDE XU_PERR_C;                      /* XUR ERROR EQUS           */
/**/
%VLP_NAME (FPTN = ERRF_FID,
           NAME = ':ERRCORNER.',
           STCLASS = CONSTANT);
DCL PROMPT CHAR(0) CONSTANT INIT('Corner: ');
%XUR_INIT (NAME=FPT_INIT,
           STCLASS=CONSTANT);
/**/
%XUH_PARAM (NAME = XUH_PARAM,
            STCLASS = STATIC);
DCL EXITING BIT(1) STATIC;
%VLP_ERRCODE (FPTN = ERR_CODE,
              STCLASS = STATIC);
DCL OUT$ REDEF ERR_CODE PTR;
/**/
%PARSE$OUT (NAME=OUT$BLK,
            STCLASS=BASED);
%PARSE$SYM (NAME = OUT$SYM,
            STCLASS = BASED);
/**/
DCL M$DO DCB;
DCL DEG$BEAST ENTRY(1) ALTRET;
DCL FUZZY_NODES BIT(1) SYMREF;
/**/
%EQU FALSE# = '0'B;
%EQU TRUE# = '1'B;
/**/
      EXITING = %FALSE#;
      CALL XUR$INIT(FPT_INIT);
      CALL XUR$SETERRMSG(ERRF_FID);
/**/

      DO WHILE (NOT EXITING);
         CALL XUR$GETCMD(FUZZY_NODES,OUT$,VECTOR(PROMPT)) ALTRET(PARSE_ERROR);
         DO CASE (OUT$ -> OUT$BLK.CODE); /* ON COMMAND TYPE          */
            CASE (%DEP$HELP_CMD#);
              XUH_PARAM.HELP$ = OUT$ -> OUT$BLK.SUBLK$(0) -> OUT$SYM.TEXTC$;
              CALL XUR$HELP (XUH_PARAM);
            CASE (%DEP$QUES_CMD#);
              CALL XUR$MOREMSG (XUH_PARAM);
            CASE (%DEP$QQ_CMD#);
              CALL XUR$ALLMSG (XUH_PARAM);
            CASE (%DEP$CLEAN_CMD#);
              CALL DEG$BEAST (OUT$) ALTRET(ITS_OK); /* HAVE BEAST CLEAN UP  */
ITS_OK:       ;
            CASE (%DEP$QUIT_CMD#);
              EXITING = %TRUE#;
            END;                         /* END CASE ON COMMAND TYPE */
         DO WHILE (%FALSE#);
PARSE_ERROR: ;
            CALL XUR$ECHOIF (DCBNUM(M$DO));
```

Figure 4-1. Command, Error, HELP Processing Source (cont. next page)

```
            IF ERR_CODE.ERR# = %E$SYNERR
             THEN
               CALL XUR$ERRPTR (,DCBNUM(M$DO));
            CALL XUR$ERRMSG (ERR_CODE);
            END;
         END;                              /* END WHILE NOT EXITING    */
BAILOUT: ;
      CALL XUR$CLOSE_DCBS;                 /* CLOSE ALL DCBS WITH SAVE */
      RETURN;
      END CORNER;

/*M* DEG$BEAST - BEAST module for the CORNER processor.                  */
DEG$BEAST: PROC (NODE$) ALTRET;
/**/
/*T************************************************************
 *T*                                                          *
 *T* COPYRIGHT, (C) HONEYWELL INFORMATION SYSTEMS INC., 1983 *
 *T*                                                          *
 *T************************************************************/
/**/
%INCLUDE CP_6;
%INCLUDE DE_PERR_C;                    /* ERROR CODE EQUS          */
%INCLUDE XUR_ENTRY;                    /* XUR ENTRY DCLS           */
/**/
DCL NODE$ PTR;
/**/
DCL FUZZINESS SBIN WORD STATIC INIT(99);
%VLP_ERRCODE (FPTN = ERROR_CODE,
              FCG = "DE",
              MID = "G",
              STCLASS = STATIC);
/**/
DCL M$OU DCB;
%EQU MAXFUZZ# = 18;
/**/

      IF FUZZINESS > %MAXFUZZ#
       THEN
         DO;
           ERROR_CODE.ERR# = %DEG$TOO_FUZZY#;
           ERROR_CODE.SEV = 2;
           CALL XUR$ERRMSG (ERROR_CODE,DCBNUM(M$OU));
/*E*     ERROR:   DEG-DEG$TOO_FUZZY#-2
        MESSAGE: File %%FN %too fuzzy.
       MESSAGE1: File %%FN %is too fuzzy to use as a corner.
    DESCRIPTION: The file intended for use as a corner is unsuitable
                 because it is too fuzzy for regular cleaning.  This
                 may be the result of prior contact with a beast.
                 Until this code becomes more sophisticated, the
                 work-around is to have the file dry-cleaned by a
                 professional.
*/
           ALTRETURN;
           END;
      ;                                /* Beasts too lazy to do any */
      ;                                /* cleaning; we'll do nothing */
      RETURN;
/**/
      END DEG$BEAST;
/*M* DE_PERR_C - This module contains error %EQUs for CORNER             */
/**/
/*T************************************************************
 *T*                                                          *
 *T* COPYRIGHT, (C) HONEYWELL INFORMATION SYSTEMS INC., 1983 *
 *T*                                                          *
 *T************************************************************/
/**/
```

Figure 4-1. Command, Error, HELP Processing Source (cont. next page)

```
/*  The following errors are generated in DEG$CORNER.  */
/**/
%EQU DEG$TOO_FUZZY#= 100/* DEG$TOO_FUZZY# */;
%EQU DEG$CORNER_FULL#= 101/* DEG$CORNER_FULL# */;
%EQU DEG$CORNER_INACCESSABLE#= 102/* DEG$CORNER_INACCESSABLE# */;
/**/
/*  The following errors are generated in DEF$BEAST.  */
/**/
%EQU DEF$NO_BEAST#= 103/* DEF$NO_BEAST# */;
%EQU DEF$BEAST_IN_CORNER#= 104/* DEF$BEAST_IN_CORNER# */;
```

Figure 4-1. Command, Error, HELP Processing Source

```
/*M* DEP$FUZZY_NODES - This module contains parse nodes for CORNER  */
/**/
/*T***************************************************************
 *T*                                                              *
 *T* COPYRIGHT, (C) HONEYWELL INFORMATION SYSTEMS INC., 1983     *
 *T*                                                              *
 *T***************************************************************/
/**/
%INCLUDE DEP$FUZZY_E;
/**/
FUZZY_NODES                      = (HELP_CMD      |;
                                    QQ_CMD        |;
                                    QUES_CMD      |;
                                    CLEAN_CMD     |;
                                    QUIT_CMD) .END
  HELP_CMD(%DEP$HELP_CMD#)       = 'HELP' .ASYM
  QUES_CMD(%DEP$QUES_CMD#)       = '?'
  QQ_CMD(%DEP$QQ_CMD#)           = '??'
  CLEAN_CMD(%DEP$CLEAN_CMD#)     = 'CL/EAN' .B ('CO/RNER' | 'HO/USE')
  QUIT_CMD(%DEP$QUIT_CMD#)       = ('Q/UIT' | 'END' | ['E'-] 'X/IT' )
```

Figure 4-2. Command Language Definition Nodes

```
/*M* DEP$FUZZY_E - This module contains parse %EQUs for CORNER   */
/**/
/*T***************************************************************
 *T*                                                              *
 *T* COPYRIGHT, (C) HONEYWELL INFORMATION SYSTEMS INC., 1983     *
 *T*                                                              *
 *T***************************************************************/
/**/
%EQU DEP$HELP_CMD#= 1/* DEP$HELP_CMD# */;
%EQU DEP$QUES_CMD#= 2/* DEP$QUES_CMD# */;
%EQU DEP$QQ_CMD#= 3/* DEP$QQ_CMD# */;
%EQU DEP$CLEAN_CMD#= 4/* DEP$CLEAN_CMD# */;
%EQU DEP$QUIT_CMD#= 5/* DEP$QUIT_CMD# */;
```

Figure 4-3. Command Language Definition EQUs

```
 1    !CORNER. OVER AMBER.CAT
 2    Corner: THIS IS AN ERROR SINCE I DON'T KNOW WHAT TO SAY
 3      Eh?
 4    Corner: ?
 5    ?        ??       CL/EAN   E        END      HELP     Q/UIT    X/IT
 6    Corner: CLEAN
 7                  ^
 8      Eh?
 9    Corner: ?
10    CO/RNER   HO/USE
11    Corner: CLEAN CORNER
12      File AMBER.CAT too fuzzy.
13    Corner: ?
14      File AMBER.CAT is too fuzzy to use as a corner.
15    Corner: HELP (BASIC) CAT
16    Syntax:
17    CAT[ALOG] [{ACCT = account|ACCT=account,ALL|ALL}]
18    Corner: Q
```

Figure 4-4. Command, Error, HELP Processing: Sample Session

```
!DEFAULT SI=FUZZY,UI=DIGITAL,GN=FUZZY
!JOB WSN=UPSTAIRS
!PL6 DEG$BEAST.SI OVER *DEG$BEAST,ME (LS,OU,SCH)
!PL6 DEF$CORNER.SI OVER *DEF$CORNER,ME (LS,OU,SCH)
!PARTRGE.X DEP$FUZZY_NODES.SI OVER *DEP$FUZZY_NODES,ME (LS,OU)
!LINK *DEG$BEAST,*DEF$CORNER,*DEP$FUZZY_NODES OVER CORNER.GN
!L(C=0) DE?.SI OVER *MODULE_LIST(LN)  "FIND ALL MODULES THAT MAKE 'CORNER'
!E *MODULE_LIST
IF /../;DE              "DON'T BOTHER WITH '... nnn FILES'
IF /DE_CORNER_HELP/;DE  "DON'T include the HELP source
END
!SET M$SI .SI
!SET M$UI .UI
!EXTRACT.X
DA CORNER.GN
DFILE
DA CORNER.GN
XL *MODULE_LIST
BUILD OVER :ERRCORNER.GN,DE_PERR_C.SI PRO
END
!R
!DEL CORNER$DAT,CORNER$TXT FROM .GN  "GET RID OF EXTRACT SCRATCH FILES
```

Figure 4-5. Command, Error, HELP Processing: Associated Jobs

Prompting and Parsing Command Text

Prompting and parsing are performed by a call to the XUR$GETCMD library service. In the DEF$CORNER module (Figure 4-1) the call to XUR$GETCMD references parse nodes (FUZZY_NODES). These nodes coded in PARTRGE metalanguage are stored as DEP$FUZZY_NODES (Figure 4-2) and created via PARTRGE.X (see Figure 4-5).

The PARTRGE metalanguage to define parse nodes is described in the CP-6 Monitor Services Reference Manual, Section 10. The XUR$GETCMD library service, which uses the X$PARSE service to access the parse nodes, provides a consistent user interface in Honeywell-supplied processors.

The XUR$GETCMD service returns in OUT$BLK.CODE a symbol for which the sample processor performs the DO CASE statement to cause action appropriate for each command. If the command text is illegal, the alternate return is taken with an error code set for use by the library service, XUR$ERRMSG.

Syntax Prompting at Syntax Error

The parse nodes are not only used to parse command text; the nodes can also be used to prompt the interactive user by listing the legal keywords permitted in response to the user's entry of ? following a syntax error. XUR$GETCMD provides this feature by default (it can be overridden by specifying SYNTAX=NO when XUR$INIT is called). This feature aids the interactive user who is unfamiliar with a processor or the experienced user who merely forgets the exact command syntax.

In Figure 4-4 an interactive user enters illegal command text (line 2) which is processed by XUR$GETCMD resulting in an alternate return. In the PARSE_ERROR routine, XUR$ERRMSG displays the error message (EH?) associated with the error code from XUR$GETCMD. When the interactive user enters ? (line 4), the next call to XUR$GETCMD interprets this as a request to display all keywords or syntax elements permitted at this point and thus lists all legal command names (line 5) obtaining this information from the parse nodes.

Lines 6 to 8 show a variation of the above steps. In this case, because a portion of the command text (CLEAN) is legal, the XUR$ERRPTR service is called to display an up-arrow at the point where the syntax error occurred. In this example at line 6, the CLEAN command is entered incompletely. Thus in response to the user's ? at the syntax error, XUR$GETCMD lists the keywords that are permitted following CLEAN, namely CORNER or HOUSE.

Displaying Error Messages

In Figure 4-1 the DEF$CORNER module establishes its error message file name with the call to XUR$ERRMSG. (ERRF_FID is defined via the VLP_NAME macro as the file called :ERRCORNER.) The :ERRCORNER file is created from error message commentary in the CORNER processor (via EXTRACT.X as illustrated in Figure 4-5 and discussed in this section in "Creating the Error Message File"). When an error condition is detected by the DEG$BEAST module (Figure 4-1), the XUR$ERRMSG service passes an error code as an argument. That error code, together with the pre-established error file name, allows the library service to perform the error message display.

The error commentary is coded at the point in the program where the error is detected and reported. At that point the value for the specific error (e.g., DEG$TOO_FUZZY# for which there is an EQU in DEP$FUZZY_E) and the severity are supplied.

Other portions of the error code (FCG="DE" and MID="G") are supplied as options in the %VLP_ERRCODE. It is assumed that all errors in this module have the same FCG and MID. Two levels of complexity are provided by the "MESSAGEx:" blocks. Both messages use a substitution %FN that takes the file name from the passed error DCB and splices it into the text. Note that the substitution is itself bracketed by %s. This requests that none of the bracketed material should be printed if the DCB does not contain a name (to satisfy FN). In the sample, %FN is satisfied from M$SI.

The "DESCRIPTION:" block is internal documentation. It can contain any information useful to a reader of the source code. Further description of error message formats is provided in Section 5.

In the sample session (Figure 4-4), the error is reported at line 12 with the first level message (MESSAGE0). When the user enters ? for more information about the error, XUR$GETCMD is called; the CASE %DEP$QUES_CMD is satisfied and the XUR$MOREMSG service is called. XUR$MOREMSG calls either XUR$ERRMSG or XUR$HELP depending on which was in control previously. In this case, XUR$ERRMSG takes control and displays the next level message (MESSAGE1).

Note: The EH? message displayed at line 8 in Figure 4-4 as a result of a parse error is available from a system-standard set of error messages (that M$ERRMSG uses because the parser's error code contains the XU Function Code Group).

Displaying HELP Messages

The HELP command, included in the parse nodes as any other command, is processed by the DO CASE statement in Figure 4-1 and results in a call to XUR$HELP. That service passes the HELP topic and subtopic text to the X$HELP library service which performs the HELP message reporting function.

In Figure 4-4 the sample session demonstrates that a user can request HELP for any other processor and then resume processing in the current processor. A HELP file, although not shown for the CORNER processor, could easily be provided. Techniques for creating help files are discussed in Section 7, "Preparing On-line (HELP) Documentation".

Section 5

Documenting Source Code

Extractable Commentary

Much of the documentation of the CP-6 system is created in source files that contain both code and commentary. This means that documentation is kept close to the code where it is easy to access and maintain. The commentary not only documents the code, but it can also be EXTRACTed (using the EXTRACT processor) to produce manual, error message and HELP files. For example, in Appendix A of this manual is commentary EXTRACTed from code.

Structured commentary is used to document CP-6 code as it is written. This structure enforces uniformity of documentation as well as providing a hierarchy. This hierarchy allows for a gross level of detail to be given at the module or file level, with progressive levels of detail at entry points or internal subroutines, for example. Because of this structure, the EXTRACT processor can be used to create a hierarchical database of commentary. The contents of the database can be used in a variety of forms, giving technical documentation without the code.

Different types of comments document different portions of code. Figure 5-1 illustrates several comment types. An M type comment, for example, gives a one-line overview description of a file, while a D type comment documents entry points. Associated with the different type comments are keywords. These keywords, used in conjunction with a given type comment, insure full documentation of a block of code. For example, when documenting an entry point with a D type comment the keywords used are CALL (where form of the call is given), PARAMETERS (where the parameters of the call are described), etc. Certain keywords can be selected for inclusion in EXTRACT reports.

```
  1 /*M* FOO$OUTSYM Main routine for OUTSYM ghost */
  2 /*T************************************************************
  3  *T*                                                           *
  4  *T* COPYRIGHT, (C) HONEYWELL INFORMATION SYSTEMS INC., 1980 *
  5  *T*                                                           *
  6  *T************************************************************/
  7 /*X* DMC,PLM=6,IND=0,IDT=2,SDI=2,CTI=0,ENU=0,AND,DCI=4,CSU=2,...*/
  8 /**/
  9 /*P* NAME:          FOO$OUTSYM
 10
 11       PURPOSE:       To provide the main routine for the OUTSYM ghost.
 12
 13       DESCRIPTION:   FOO$OUTSYM is the main routine for the OUTSYM
 14                      ghost, which provides and/or controls all the
 15                      CP-6 output symbiont functions performed in the
 16                      host.
 17
                                 .
                                 .
                                 .
43 */
                                 .
                                 .
                                 .
688 /*D* NAME:          EXT_EVENT
689       PURPOSE:       To process an external event
690       DESCRIPTION:   EVNT$ points to an F0$EVNT frame, in which is a
691                      code, and a possible CITE$ pointing to an F0$CITE
692                      frame.
693 */
                                 .
                                 .
                                 .
771 /*S* SCREECH_CODE: FOO-S$NODEV
772       TYPE:          SNAP
773       MESSAGE:       Event reported on non-existent device
774       REMARKS:       A disconnect, break, or similar event has
775                      been reported, but the named device
776                      cannot be found.                          */
```

Figure 5-1. Source File Containing EXTRACTable Commentary

From the code shown in Figure 5-1 a programmer could, for example, use EXTRACT to create a report of all *M* comments, or a report of all *P* and *D* comments, or a separate file of screech codes from all *S* comments.

Commentary Rules

The following rules apply to producing commentary in an extractable form:

1. Commentary should not be on the same line as code.

2. To allow EXTRACT to identify commentary in various types of source files, it is necessary for the first non-blank source line to conform to these requirements:

Language	First non-blank source line must contain:	
PL6	/* or ;	in any column(s)
FORTRAN	C	in column 1
BASIC	REM	starting in column 1 (see NOTE)
APL	$COM	starting in column 1
IBEX	! or !"	starting in column 1
TEXT	. (period)	in column 1
IDL	-- (dashes)	starting in column 1
other	*	in column 1 or 7

   For example, a PL6 program starting with

```
C:    PROC
```

   is interpreted as a FORTRAN program.

   NOTE: For BASIC programs the first record must not have a line number. EXTRACT skips line numbers of subsequent records.

3. The syntax for a comment is of the form:

```
{cmtstart}{cmttype}[*]{text}{cmtend}
```

   The text can be any sequence of ASCII characters except {cmtend}. For example, a PL6 comment appears as

```
/*M* FOO$OUTSYM Main routine for OUTSYM ghost */
```

   where */ is not permitted with in the text string.

   If {cmtstart} does not end with *, then the programmer must supply *. For example, an M comment in FORTRAN would be coded as

```
C*M*text
```

4. Comments can extend to the maximum record length (e.g., 140 for PL6, 80 for FORTRAN).

5. Comments may be continued over a number of records. In PL6, comments may be continued without a /* in front of each record. However, the last line in a block of commentary must be terminated with a */.

   In non-PL6 code, lines that are continuations of a previous commented line must include either a comma as the comment type or the same letter as the comment type. Using the same comment type but repeating the previously used keyword begins a new comment, as illustrated in the following examples.

   Examples:

```
*M* EXAMPLE This is an example of non-PL6 code
*T* .....
*P* NAME: EXAMPLE of non-PL6 extract code
*,*       so that EXTRACT knows this belongs
*,*       with the 'P' comment, we use ','
```

```
          ENTDEF EXTRACT
          SYMDEF .......
    *,*       This also belongs to the 'P'comment
    *D* NAME: This is a new comment type 'D'
    *,*       Second line of type 'D'
    *D*       Third line of type 'D'
    *D* NAME: New type 'D' comment

    *D* NAME: Another new 'D' comment
    *,* Some more of the 'D' comment
```

6. Upper and lower case letters should be used to make commentary more readable.

7. Keywords must include a final colon (:), e.g., PURPOSE:

"Commentary Tools" in this section discusses a processor that formats commentary to these standards. Conventions to follow in placement of commentary within a file are also discussed later in this section.

Comment Types

Table 5-1 gives a summary of the different comment types that can be used when documenting a file.

Table 5-1. Summary of Commentary Types

Type	Description
B	Data definitions.
D	Detail on ENTRY points, PROCs and macros.
E	Error message.
F	Preamble or overview for a routine or ENTRY point.
I	Detail on internal subroutine.
K	Keyword definition or data description.

Table 5-1. Summary of Commentary Types (cont.)

Type	Description
M	One-line description of a file.
N	Denotes that code is to be added for a deferred feature.
O	Message to system operator.
P	Preamble or overview for a module or function.
S	Screech code message.
T	Denotes that copyright notice is to be inserted by FORMAT or PL6FMT.
W	Gives warning and explanation of error codes, also describes bugs and/or inefficiencies.
X	Source formatting controls used by FORMAT or PL6FMT.

M Comments

Description:

A type *M* comment is a one-line description of a file. There should only be one *M* comment in a file and it must be the first line of the file.

Format:

/*M* name - description */

Example:

```
/*M* PAYROLL - Program that computes employee payroll and taxes.*/
```

P and F Comments

Description:

P and *F* comments are preambles or overviews. A *P* is a preamble to a file; *F* is a preamble to a routine or entry point. The preambles are descriptions that include significant features or limitations of a file or routine, but do not include details.

Keywords:

The following keywords may be used in conjunction with *P* and *F* comments:

Keyword	Description
NAME:	The first record of any group of *P* or *F* comments must be the NAME keyword. The name should be the file name for *P* and the PROC/ENTRY name for *F*.
PURPOSE:	Describes the purpose of the file, routine, or entry point.
DESCRIPTION:	Describes the function and special features which this module performs.
REFERENCE:	Gives a cross-reference to manuals and/or other reports or documents.

D Comments

Description:

Type D comments give detailed description and should be inserted for every entry point, including the PROC statement. Descriptive *D* comments may be interspersed with the code to explain individual lines or blocks of code. The *D* comments appear at the point of entry rather than the front of a module. They include the information necessary for the user to know how to use the routine and what to expect of it in the way of usage, interfaces, input, output, etc. These comments will be collected by EXTRACT and included in the *D* (detail) report. If a routine contains more than one entry point with similar details, only the first *D* comment needs to include full description for each keyword; only the keyword ENTRY: and any differences need be specified on *D* comments for later entry points.

Keywords:

The following keywords may be used in conjunction with *D* comments:

Keyword	Description
NAME:	Defines an entry point. It must be the first record of a group of *D* comments. The name is used for sorting.
ENTRY:	Needed only if multiple entries require the same detail report. If a routine contains two entry points which are conceptually alike, the programmer, when documenting the second entry point need only enter NAME:, ENTRY: and the keywords and paragraphs which distinguish the second entry point from the first. This saves the programmer a lot of typing because the EXTRACT processor then copies the common fields from the first entry point into the second entry point.
CALL:	Gives the calling sequence for this routine, including the alternate return (ALTRET), if used.
PARAMETERS:	Describes parameters of the call. This field along with the CALL commentary should be sufficient to describe how to invoke the routine.
INTERFACE:	Lists routines that this routine calls and routines that can call this one. Describes declaration of external entry points. Also lists any INCLUDE files needed to use this routine.

Keyword Description

ENVIRONMENT:

Describes anything special about how to run this routine such as mapped/unmapped, master/slave, file authorization required (privileges), inhibit or not, etc.

INPUT:

Data accessed (external as opposed to global) to perform a function. Can also list the inputs from the calling sequence.

OUTPUT:

Data altered and intended as the result of this operation/routine. Can also list the outputs of the calling sequence.

SCRATCH:

Data altered but not intended as results (side effects).

DESCRIPTION:

Describes what the routine does and how it functions.

ALTRETURN:

Describes the conditions which cause an alternate return, if the routine is defined with the ALTRET option.

Example:

```
/*D*     NAME:        ZYF$CALL_BUILTIN_FUNC
        CALL:        CALL ZYF$CALL_BUILTIN_FUNC (V_RESULT,
                          BUILTIN_ID,ARG_COUNT,V_ARG_LIST) ALTRET(ERR);
        INPUTS:      BUILTIN_ID, UBIN - %EQUs are in MIIL definition.
                     ARG_COUNT, UBIN - actual number of arguments
                             provided.
                     V_ARG_LIST - array of up to 63 "values" representing
                             the arguments.
        OUTPUTS:     V_RESULT - a "value" representing the function result.
        DESCRIPTION: Verify the argument count and call the proper
                     routine to do the real work.
                     If trouble is encountered, a diagnostic will be
                     issued and a default value of the appropriate type
                     will be returned to minimize later confusion.
        ALTRETURN:   If a non-recoverable difficulty is detected.  */
```

B Comments

Description:

B comments are data definitions. All external data bases must contain a definition for each field. The *B* comment must follow the field it describes.

Format:

```
/*B* name - description
*/
```

I Comments

Description:

I comments describe internal subroutines with no external calls.

Keywords:

The following keywords may be used in conjunction with *I* comments:

Keyword	Description
NAME:	Defines the name of an internal subroutine. The first record of a *I* group must be the NAME keyword.
PURPOSE:	Describes the function performed by this internal module.
CALL:	Documents the calling sequence. Also discussed here are the means by which this routine could altreturn, if possible.
PARAMETERS:	Lists and describes the input and output parameters.
DATA:	Describes any additional data which might be required by this routine. An example would be the use of globals.
DESCRIPTION:	Describes what this routine does, how it functions, and the details of its purpose.

Example:

```
/*I*     NAME:         CHECK
        PURPOSE:      Check that an acceptable number of arguments
                      has been specified.
                      Print message and ALTRETURN if not.
        INPUTS:       WANT, UBIN - number of arguments function
                              wants.  163 is a special case for
                              minimum/maximum who can handle 1 thru 63.
                      NAME - name of function followed by '.'
        DESCRIPTION:      It would be a whole lot easier to have constant
                      arrays containing the textual representations
                      and number of arguments expected but this
                      introduces maintenance problems.
                                    .
                                    .
                                    .
*/
```

E Comments

Description:

E comments are used to create error message files and manual appendixes. The comments should describe what went wrong to cause delivery of this message. These comment lines appear in the listings at the point of first occurrence, since multiple uses of one error code and message are possible. The comment should include the error code, if any, and a description of the error condition. Further discussion of error message reporting is included elsewhere in this manual.

Keywords:

The following keywords may be used in conjunction with *E* comments:

Keyword	Description

ERROR:

This is the error code consisting of the FCG (Functional Code Group), MID, MON, error code, and severity level (e.g., FMA-E$E0F-3). An EQU should also be made to relate this error comment to its appropriate call. The ERROR: keyword must be the first record of the group.

MESSAGE:

Contains the actual text of the error message. This field must not be used with the numbered message fields.

MESSAGE0:

This field contains the first level of a description of what the error is. It is used in conjunction with at least one of the following MESSAGEn fields. This field should not be used with the unnumbered message field.

MESSAGE1:

This is the next level of error message, to be displayed after a user types a ? after receiving MESSAGE0. Additional keywords, MESSAGE2, MESSAGE3, ..., MESSAGE7, may be specified.

DESCRIPTION:

This field is used as an aid to the programmer to describe how this error can occur, a possible work around, and a means by which the error might become eliminated in the future. It is never delivered with the error message.

W Comments

Description:

A *W* comment can be used anywhere in a file to describe a block of code, its possible bugs and/or inefficiencies. It is also used to give warnings to the programmer or to explain problems.

Format:

```
/*W* commentary
*/
```

S Comments

Description:

S comments are used to create the Screech code message file and manual appendixes. These comments are included where a Screech condition occurs. EXTRACT will supply the filename for the "REPORTED BY" keyword.

Keywords:

The following keywords may be used in conjunction with *S* comments:

Keywords	Description
SCREECH_CODE:	Screech code to identify the condition (e.g., FOF-S$OFFADD). The SCREECH_CODE: keyword must be the first record of the group.
TYPE:	Screech, SUA (Single User Abort), or SNAP
MESSAGE:	Explanation of what caused the Screech.
REMARKS:	Further description of the cause.

Example:

```
/*S* SCREECH_CODE: FOF-S$OFFADD
     TYPE:        SNAP
     MESSAGE:     A batch job added a file to OUTSYM after
                  going OFF.
     REMARKS:     OUTSYM receives an event from MONKEY
                  whenever a batch job leaves the system.
                  This SCREECH occurs when OUTSYM receives
                  an output file add from a batch job which
                  has been marked off.  This means that
                  OUTSYM or the monitor is very confused
                  about that job.
                  */
```

K Comments

Description:

The *K* may be used for keyword definitions or for data descriptions to be extracted as user documentation.

Format:

```
/*K* name - definition or description
*/
```

O Comments

Description:

A *O* comment is a message to the system operator. Properly formatted, it will provide the appendix to the operators manual; thus keywords are important. A *O* comment should appear in code wherever an operator message is reported.

Keywords:

The following keywords may be used in conjunction with *O* comments:

Keyword	Description
MESSAGE:	Contains the actual text of the message.
ACTIONS:	Instructs the operator on the course of action to take.
MEANING:	Describes the message in detail.

T Comments

Description:

A *T* comment is placed at the point in a file where the Honeywell copyright notice will be inserted. This comment does not have to be inserted by the programmer. It is inserted by FORMAT.X or other services, prior to the program files' release. By convention, the *T* comment is placed after the *M* comment at the head of a file.

Format:

```
/*T*/
```

N Comments

Description:

N comments are placed at a point in a file where code is to be added at a future date. They can also be used to identify minor features that are not supported as a reminder to the module owner.

Format:

```
/*N* keyword - description
*/
```

X Comments

Description:

X comments are used to place PL6 format controls in a file. These comments may be placed at any point in a file, and may be used numerous times. The last encountered set of format commands are the ones followed when formatting. A full description of the format commands appears in in the PL-6 Reference Manual (CE44), "Format Facility".

Format:

```
/*X* (format option),(option), ...
*/
```

Placement of Commentary in a File

Certain comment types are intended for the beginning of modules and routines; others comment types are embedded in the code at the point of significance. Figure 5-2 illustrates where to include each comment type in a file. The notes following each comment describe how the different comment types are used.

```
/*M*        */        *M*, *T*, *X* and *P* comments occur once
/*T*        */        at the beginning of a file.  *P* comments
/*X*        */        describe the function of a file.
/*P*        */

/*D*         */       *D* comments can be used to document the
%MACRO                details of a macro.
%MEND;


/*F*        */        *F* and *D* comments are used to indicate
/*D*        */        a procedure or entry point in a file.  They
A: PROC;              may appear any number of times.

/*K*        */        *K* and *B* comments are used to define data
/*B*        */        types.
DCLs

/*F*        */
/*D*        */
B: ENTRY;

/*N*                  *N*, *O*, *S*, *W* and *E* comments may
 *O*                  appear at any appropriate point in a file.
 *S*
 *W*
 *E*        */
CODE FOR MAIN PROGRAM;

/*I*        */
Routine internal to this PROC;
END A;

%EOD;                 %EOD indicates the start of a new procedure.
/*F*        */
/*D*        */
Z: PROC;

/*I*        */
Routine internal to this PROC;
END Z;

%EOD;
  .
  .
```

Figure 5-2. Placement of Commentary in Source Code File

Commentary Tools

The X Account provides a multitude of tools which assist the user in formatting and updating commentary, extracting commentary into a database and exercising the database. Two of the more useful tools (EDICT.X and EXTRACT.X) are described briefly below. For more information on available tools please see Section 3, "Programmer Aids" and "Documentation Aids."

EDICT.X

EDICT.X is an interactive tool which formats commentary into an extractable form. EDICT.X formats commentary, supplies keywords, allows use of EDIT features, etc. For more information, invoke EDICT.X and request HELP within that processor.

EXTRACT.X

The EXTRACT processor extracts error messages and commentary from source code into a data base which then can be used to create Error Message files, HELP files and Reference Manual files (portions of the Monitor Services Reference Manual, for example).

Text Blocking in Extractable Commentary

A double colon can be placed after a Keyword to cause EXTRACT to block the text on the left margin. A single colon tells EXTRACT to indent the text.

Example 1:

The following example illustrates what the output would look like if a single colon is used after the keyword:

```
Keyword: The text will be EXTRACTed
         in a block format like this
         when a single colon is used.
```

Example 2:

The following example illustrates what the output would look like if a double colon is used after the keyword:

```
Keyword:  The text will be EXTRACTed in a block format like this when a
double colon is used.
```

Sample EXTRACT.X Job

The EXTRACT processor provides a number of commands to request extracting and formatting of specific types of commentary. In addition, certain commands also perform alphabetical sorting to generate easy-to-use reports. See HELP (EXTRACT.X) for more information. Figure 5-3 shows a sample use of EXTRACT: source file commentary is first displayed; an EXTRACT job using the DOC command (and others) is performed. Figure 5-4 shows the resulting document processed by TEXT, illustrating the commentary sorting, formatting, and addition of page headers and footers performed by EXTRACT.X.

```
!C EDICT_SOURCE
/*M* EDICT - EXTRACTABLE DOCUMENTATION IN CRISMAN TERMS */
/*T*************************************************************/
/*T*                                                           */
/*T* COPYRIGHT, (C) HONEYWELL INFORMATION SYSTEMS INC., 1983 */
/*T*                                                           */
/*T*************************************************************/
/*X*   PLM=2,STI=2,IND=2,CTI=4,DCI=5,PRB=YES,ECI=3,CSI=3,THI=2,
       IAD=2,DIN=2,ENI=4,CLM=0,CCC,MER=NO,CCE,SQB=YES,MCI=YES  */
/*P*
     NAME: Document
     PURPOSE:
          This program prompts the user for input for the *M* *P* *F*
          *I* *D* program documentation standards.
     DESCRIPTION:
          Using standard invocation syntax this program creates a
          separate file containing the document material.
          This documentation was created with this program!
     REFERENCE:
          Any problems?  see gary palmer .
*/
/*I*
     NAME: Promptfile
     PURPOSE:
          This routine does the structuring of the *M* document, since
          .
          .
          .
     CALL:
          Call promptfile;
     DATA:
          This routine also structures the *P* document section.
     DESCRIPTION:
          Create the *M* and *P* documents.  Then ask if you want to
          document a module.
*/
   .
   .
   .
/*I*
     NAME: Dotext
     PURPOSE:
          Does the body of text for any subsection of a document type.  For
          example: it would create the paragraph/s  for purpose or call.
     CALL:
          Call DOTEXT(HEAD,P_FLEG);
     PARAMETERS:
          HEAD: Name of the subsection we are documenting.  Required parameter.
          P_FLAG: Indicate whether this is a required section or not. Optional
     DATA:
          None.
     DESCRIPTION:
          Sets up the section defined by head, if the input is nil and this is
          .
```

Figure 5-3. Sample EXTRACT.X Job (cont. next page)

```
          .
          .
*/
EDICT: PROC MAIN;
%INCLUDE CP_6;
%INCLUDE B$JIT;
%INCLUDE B_ERRORS_C;
   .
   .
   .
*
*
!XEQ EX_EDICT_JOB
$JOB
$RESOURCE TIME=1,MEM=216
$DONT ECHO
  File EDICT_DOC does not exist
          CP-6 EXTRACT Version B03 - February 1982

   > DA EX
   - New Data Base - EX.SAMPLE
   > EX EDICT_SOURCE
   - Extracting from - EDICT_SOURCE.SAMPLE
   > DOC ME,EDICT_DOC
   > SECTION=EDICT
   > HEADING=DOCUMENTATION FOR EDICT.X
   > ?
   - Document file EDICT_DOC Created.
   -    10 Records in Data Base EX.SAMPLE - Saved.
   - Good Bye.

EX$DAT      EX$TXT
     2 files,      8 granules deleted
$ "------> TEXT EDICT_DOC TO LP
DATA IGNORED:
   1 records ignored
```

Figure 5-3. Sample EXTRACT.X Job

```
                    DOCUMENTATION FOR EDICT.X
                          EDICT_SOURCE


                                                                    00009
NAME: Document
PURPOSE:  This program prompts the user for input for the *M* *P*
       *F* I* *D* program documentation standards.
DESCRIPTION:  Using standard invocation syntax this program
           creates a separate file containing the document
           material.  This documentation was created with this
           program!
REFERENCE:  Any problems?  see gary palmer .

                                                                    00100
               NAME: Dotext
               PURPOSE:  Does the body of text for any subsection
                      of a document type.  For example: it would
                      create the paragraph/s for purpose or call.
               CALL:  Call DOTEXT(HEAD,P_FLEG);
               PARAMETERS:  HEAD: Name of the subsection we are
                         documenting.  Required parameter.
                         P_FLAG: Indicate whether this is a
                         required section or not. Optional
                         DATA:  None.
               DESCRIPTION:  Sets up the section defined by head,
                             .
                             .
                             .
-------------------------------------------------------------------------------

COPYRIGHT, (C) HONEYWELL INFORMATION SYSTEMS INC., 1983

                                EDICT-1



                    DOCUMENTATION FOR EDICT.X
                          EDICT_SOURCE


                                                                    00021
               NAME: Promptfile
               PURPOSE:  This routine does the structuring of the
                      .
                      .
                      .
               CALL:  Call promptfile;
                   DATA:  This routine also structures the *P*
                   document section.
               DESCRIPTION:  Create the *M* and *P* documents.
                         Then ask if you want to document a
                         module.

               .
               .
               .
                             EDICT-2
```

Figure 5-4. TEXTed Document of EXTRACTed Commentary

Section 6

Error Message Reporting

The CP-6 system provides a centralized facility for the storage, selection and delivery of error messages. A monitor service, X account tool, library routine and set of standards provide an easy and uniform access to error messages.

Error Message Source

CP-6 error messages are embedded in source code as type E comments. The error messages are placed in the code directly after the call to M$ERRMSG. These comments document the code and are also used to create error message files. Figure 4-1 shown earlier in this manual illustrates the error message source format.

Error messages should provide the user with meaningful as well as accurate information. Therefore, messages should be a sentence that is specific and clear. Substitution fields (discussed later) should be used frequently to provide the user with as much detailed information as is possible. The error encountered should be specifically referenced. Layered messages, to be displayed successively at the request, should expound on the causes and cures of the error.

Error Codes

Each error message is associated with a standard CP-6 error code. The code is split into four sections: the FCG, the MID, the MON and SEV. The FCG, or Functional Code Group, is a two-character alphabetic field which indicates which processor or portion of the operating system is reporting or has detected the error. MID is a character that indicates which module of the processor detected the error. MON determines whether or not the message came from the monitor. Error number is a number that distinguishes a particular error from others with the same FCG, MID, and MON.

The last field, SEV, serves two purposes:

1. It indicates which level of message complexity is desired when part of a code is passed to M$ERRMSG.

2. It determines what will happen to a caller when part of a monitor service call does not have an ALTRET. For more information see the description of M$MERC in the Monitor Services Reference Manual.

Example:

```
DEG-E$FUZZERR-2
```

where

DE specifies the FCG. This field may be blank.

G specifies the MID. This field may be blank.

E$FUZZERR is the particular name or 1-to-4 digit number unique to this particular error. If a name is specified, an EQU must be included to define the error number.

2 specifies the severity of the error.

Layers of Error Messages

CP-6 error messages may have up to eight levels. The text of the message(s) is specified following in *E* commentary following the keyword MESSAGE if a single level is used, or following the keywords MESSAGE0, MESSAGE1, ... ,MESSAGE6, MESSAGE7 if multiple levels are used. The contents of each level, up to but not including the highest one used, may be built upon the lower levels. Each message has different levels of complexity with the lowest-level message being short and to the point and each successive message elaborating on causes and cures.

The highest level of error message must be self-contained since it is the only message that a batch user receives. The environment at the time of the error dictates which message is to be printed for a specific error. Interactive, time-sharing users are given the lowest-level (least verbose) message. The user can enter "?" to obtain the next level of message. This continues until the user figures out the problem, or the messages are exhausted. A second construct, "??", displays all remaining messages in order of ascending complexity. Batch users are unable to request additional error messages since the command stream was written before the occurrence of the error. To accommodate them, the highest available level of message is displayed.

The selection of the original level and processing of the ?/?? commands takes a moderate amount of code. To save code and enforce consistent treatment, library routines are provided for the message display and subsequent elaborations. For details, please see the documentation for XUR$ERRMSG, XUR$MOREMSG and XUR$ALLMSG in the Monitor Services Reference Manual.

Field and Phrase Substitution

An error message consists of one or more lines of text with optional substitution fields. Substitution fields allow a canned message to be made specific with details about the causes of the error. As an example, the monitor message for "file does not exist" actually gives the name of the offending file. This technique is discussed under "Field and Phrase Substitution" in the M$ERRMSG section of the Monitor Services Reference Manual.

Default Error Messages

Default error messages can be defined pertaining to:

- All occurrences of the same error condition within all modules in a Functional Code Group. In this case the Module ID is set to NIL.
- All occurrences of the same error condition within all Functional Code Groups in the user procedure. In this case the Functional Code Group and Module ID are set to NIL.

When the requested error message is not available, M$ERRMSG can go to great lengths in search of a suitable substitute. The method used by M$ERRMSG to determine which message to display is discussed in Monitor Services Reference Manual in the SUBMESS Option subheading following the description of the M$ERRMSG service.

Examining the Error Code After Monitor Service ALTRET

If the caller has not aborted, the error code can be examined in the ALTRET frame. The entire standard error code fits into a single 36-bit word. The structure is described under VLP_ERRCODE in the Monitor Services Reference Manual. For more information on accessing the TCB see Section 8, "Accessing the TCB."

Creating the Error Message File

The EXTRACT.X processor can be invoked to create a data base of commentary, and to build an error message file in the format required (i.e., a file that is keyed by the modified error code). Although M$ERRMSG can access an error message file with any specified name, EXTRACT will supply default names for error message files it creates based on the FCG and MON.

Figure 6-1 illustrates a job that creates an error message file named in Figure 4-1.

```
   1.000 !L(C=0) DE?.SI OVER *LIST(LN)
   2.000 !E *LIST
   3.000 IF /../;DE
   4.000 FD 0-99999.999,/DE_CORNER_HELP/
   5.000 !SET M$SI .SI
   6.000 !SET M$UI .UI
   7.000 !EXTRACT.X
   8.000 DA CORNER
   9.000 DFILE
  10.000 DA CORNER
  11.000 XL *LIST
  12.000 BUILD OVER :ERRCORNER PRO
  13.000 END
  14.000 !R
  15.000 !DEL CORNER$DAT,CORNER$TXT
```

Notes

- Prior to calling EXTRACT, use the List command to place names of all the modules with the FCG of DE into *LIST (line 1) Using EDIT, delete from *LIST all lines containing ".. n FILES LISTED" and the HELP source file which contains no error messages (lines 2-4).

- Specify the accounts from which the source and update files named in *LIST are to come (lines 5-6).

- Invoke EXTRACT.X (line 7), establish the prefix for the names of the work files (line 8), delete any old work files (DFILE at line 9), reestablish the prefix for work file names (line 10), create a data base of commentary from the files named in *LIST (line 11), and build the error message file (line 12) in the format required by M$ERRMSG.

Figure 6-1. Sample Job to Create Error Message File

Foreign Language Error Message Files

The CP-6 error-message system supports messages in various languages. Each user's JIT contains a one-byte value determined by his declared native language. This byte is appended to the name used by M$ERRMSG in opening the message file, so supporting a new language is as simple as placing translated messages into a file with the appropriate name. More information on this feature and the naming convention for error message files can be found in the Monitor Services Reference Manual under the FILENAME parameter description of M$ERRMSG.

Finding the Error Message File

When calling the M$ERRMSG monitor service, a programmer must supply the name and account of the appropriate error message file. If the program and its error message file can be assumed to be in the same account, the program can obtain the account as follows. Every running program has DCB M$LM associated with it as DCB #2. The progran can find what its name is by accessing the DCBN.NAME#.C field, and its account by accessing DCBN.ACCT#. The DCBN.ACCT# can be specified as the account on the call to M$ERRMSG. As long as the error message file and the run unit are transported to different accounts as a set, the error message file can always be found.

Section 7

User Documentation/HELP

$TEXT Facility

The CP-6 user documentation -- including this Guide -- is created via the $TEXT.:DOCUM facility. This facility can produce several forms of documentation from source text files: hard-copy documents, document unit-record files for electronic distribution, and HELP files.

$TEXT capabilities include:

- Assembly of complete manuals from multiple files.
- Automatic generation of title page, table of contents, and index.
- Automatic layout of headings, tables, and figures.
- Generation of HELP files from source text files.

Following the conventions discussed in this section assures device-independent output for hard-copy and HELP files.

Files that can be processed by $TEXT are keyed files built via EDIT. The files contain text, plus TEXT control words and macro control words. Once source text files are built, compiling a complete document is a fast, simple process. "Document assembly" can be requested through a menu-driven interface described in this section. The same interface allows creation of a HELP file; see "Preparing On-line (HELP) Documentation" in this section for details.

The tools for the user of $TEXT include:

- The EDIT processor to build source text files.
- The menu-driven interface, $TEXT.:DOCUM (which uses $FASTEXT.:DOCUM and invokes the TEXT processor).
- Macros which are stored in :LIBRARY (e.g., :MAT.:LIBRARY).

Conventions for text source files are highlighted in this section. A full description of $TEXT capabilities is provided in guide CE59. Full information on TEXT is provided in the CP-6 Text Processing Reference Manual (CE48); a HELP file is also available. In addition, the CP-6 Text Processing Administrator Guide (CE52) contains related information.

File Naming Conventions

File naming convention are crucial to document assembly. Source text file names must be of the form: CEdocnum_secnum or HAdocnum_secnum. For example, the file CE40_01 is manual CE40, section 1. The section number is used during document assembly to order the files. The section numbers are organized as follows:

Section Number	File Identified
00	The front matter section, which contains the following standard introductory material: the Preface, Title Page, a call to the Table of Contents file, About This Manual, and the Syntax Notation page.
01 thru 49	The numbered document sections.
50 thru 59	The appendixes.
90	The glossary.

Table of Contents and Index files are created and included in a hard-copy manual during document assembly.

NOTE: After processing by $TEXT, the resulting document may be printed in hard copy form or stored in one unit record file (from which multiple copies could be made). The name for such a unit record file may be any legal FID.

Document Assembly

The user can initiate a dialog with $TEXT during which it prompts the user for all the information required to perform document assembly.

To begin the document assembly dialog with $TEXT, the user enters the following:

```
!XEQ $TEXT.:DOCUM
```

$TEXT responds with:	The user enters:
FASTEXT B03 Files to Format>	The names of the files to be printed. A wildcarded file name in the form prefix? is specified to identify multiple files.
In Account>	The name of the account in which the files reside.
Device or Destination>	The location of the file or printer device to which the output is to be routed.
TEXT Options>	The options to be used to format the document.
DRAFT or FINAL Format>	The format to be used for the document.
Pagesize (type of paper)>	The type of paper that the document is to be printed on.
Number of Copies>	The number of hard copies desired.

Extra Files>	Whether to generate a table of contents, an index, and/or a typeset header file as part of the document assembly.
Go for It?>	Whether to begin the assembly process, initiate a response change mode, or terminate document assembly.

All user responses are followed by a <CR>. For all prompts, there is a default. The user selects the default by entering <CR> only. Typing HELP<CR> in response to any prompt (except "Go for it") displays appropriate responses to the prompt. For complete information on document assembly, please see CE59.

Summary of Control Words and Macros

Text source files contain text, TEXT control words, and macro control words. TEXT control words regulate indentation, spacing, etc. The macros control placement and underlining of section and subsection headings, positioning of headings and table columns, positioning of figure and table headings, and identification of index entries.

Each control word or macro is placed on a line by itself, beginning in the first character position of the line. Control words begin with a period in position 1. Macros begin with periods in positions 1-2. Table 7-1 summarizes the subset of TEXT control words frequently used in files processed by $TEXT. Table 7-2 summarizes the frequently used macros recognized by $TEXT.

Table 7-1. TEXT Control Word Summary

Control Word	Description
.SPB [n]	Creates n blank line(s), where a page break is acceptable
.SPF [n]	Creates n blank line(s), where a page break is unacceptable
.FIF	Causes the following copy to appear on output exactly as it is entered in the file
.FIN	Causes text formatting to resume. That is, after a .FIF and portion of text to be output exactly as entered in the file, .FIN causes text to fill full lines in the output produced.
.INL n	Indents the following text lines by n positions

Table 7-1. TEXT Control Word Summary (cont.)

Control Word	Description
.UNL n	Starts the following line n positions to the left from the indent position specified by the previous .inl control word
.BRP	In files to be processed by $TEXT, "break page" (.BRP) is restricted to use in the end-of-file sequence. ($TEXT performs page layout.)
.SRV name expr	In files to be processed by $TEXT, .SRV SECTION {n|"x"} creates the section or appendix identifier for use on page 1, in page numbering, and in the table of contents. The identifier is a number for sections, a letter for appendixes (e.g., "A"), or "g" for a glossary.
.TRF xy	Translates character x to character y until the next .trf control word is encountered. $TEXT assumes that the character ^ is to be translated to a blank (i.e., it assumes .TRF ^) except in figures.

Table 7-2. Macro Summary

Macro	Description
..:LOH "head"	Creates a section name.
..:L{1|2|3|4}H "head[;help_info]"	Creates a subsection head of level 1-4, formatted according to $TEXT conventions. For levels 1-3, creates an entry in the table of contents and index, if requested. By default, creates a HELP topic of level 1-3 heads with associated level 4 heads as subtopics.

Table 7-2. Macro Summary (cont.)

Macro	Description
..:MAT "table_info"	Creates a table from subsequent text, controlling spacing before and after the table, table title, table headings, and table layout. Table types are as follows: matrix, 2-column formatted, and unformatted. Also creates an entry in the list of tables, if a table of contents is requested.
..:END	Terminates table text.
..:FIG "fig_info"	Creates a figure from subsequent text, controlling spacing before and after the figure, placement of figure title, and layout. Also creates an entry in the list of figures, if the table of contents is requested.
..:IDX "index_info"	Creates an index entry for a term included or implied in the preceding text.
..:HLP "[manual_text][;help_text]"	Creates alternate wording for the manual and for the HELP file.

Creating Text Source Files

This subsection describes conventions for creating a text source file for hard-copy documentation. Guidelines for producing on-line HELP files from the same files are discussed in a later subsection.

Line Length

In general, it is advisable to keep line length to a maximum of 79 characters. In unformatted copy-- such as figures, unformatted tables, or matrix tables-- line length must not exceed the page width. Length restrictions are stated in "Usage Notes" in the discussion of the macros for tables and figures.

HELP files must not include lines of greater than 79 characters. This can be accomplished by entering text according to the guidelines stated above and by limiting :HLP macro lines to 79 characters. In generating a HELP file, a warning is issued for records exceeding 79 characters. NOTE: Overstriking counts as two characters (backspace, and the overstrike character), and should be avoided in text that is to appear in HELP files.

Blocking

$TEXT assumes that text is blocked. That means that all paragraphs start on the left margin. Lists do too, unless they are lists within lists. Indentation of list text is accomplished by use of the TEXT control words .INL and .UNL.

Spacing

$TEXT produces single spaced output. Macros for headings, tables, and figures cause appropriate spacing: between headings and text, between tables and text, and between figures and text.

The user enters the TEXT control word .SPB between paragraphs. In certain cases, where a blank line is needed but a page break is undesirable, the TEXT control word .SPF can be used. For example, .SPF should be entered in these cases: after a line ending in a colon that introduces a list of items on separate lines, between lines in a table where a page break is inappropriate, and between lines in syntax formats.

Section and Subsection Headings

Documentation files are hierarchically structured. In addition to the section heading, four levels of headings are permitted. Headings created through macros are distinguished typographically (by upper- and lower-case and by underscoring) and by appropriate spacing. $TEXT also attaches specific meaning to the heading levels for hard-copy and HELP.

When outlining a new manual, both the hard-copy heads and on-line HELP topics should be considered. For example, all commands of a given processor may appear as level 2 headings with level 4 headings (such as "Format:", "Parameters:", etc.) subordinate to each command heading; the command names become topics, and "Format", "Parameters", etc. become subtopics for each command. The following discussion of level head macros illustrates this point.

Note: In CP-6 user documentation, the order of level 4 heads for command (and similar) documentation is as follows: Format, Parameters, Description, Example. This order of presentation produces the most usable HELP facility.

Level 0 Head Macro

Format:

```
..:LOH "section_name"
```

Parameters:

section_name is the 1-49 character section name.

Description:

This macro creates and formats the section name, and creates an entry in the table of contents (if it is requested at document assembly).

Example:

```
..:LOH "User Documentation/HELP"
```

creates the section name for this section.

Level 1-3 Head Macro

Format:

```
..:L{1|2|3}H "[headname][;][{X|help_topics [synonym_list]}]"
```

Parameters:

headname is the heading name. headname can be up to 58 characters for level 1, 55 for level 2, and 52 for level 3.

help_topic is the 1-31 character HELP topic name, if other than headname. help_topic must be specified even if the same as headname if a synonym list follows. help_topics must not include blanks; for example, LEVEL_1 could be the topic associated with the heading "Level 1-3 Head Macro".

synonym_list is a list of 1-31 character synonyms for the HELP topic. Synonyms must not contain blanks.

Description:

This macro creates and formats the level head, and creates an entry in the table of contents and index, if they are requested at document assembly. The macro also creates a HELP topic unless X is specified.

Example:

```
..:L2H "COPY Command;COPY C CO COP"
```

creates COPY Command as a hard-copy heading, COPY as a HELP topic, and C, CO, and COP as HELP synonyms.

Level 4 Head Macro

Format:

```
..:L4H "[headname][;{X|help_subtopic [synonym_list]}]"
```

Parameters:

headname is the head contents of up to 255 characters.

help_subtopic is the 1-31 character HELP subtopic name, if other than headname. help_subtopic must be specified, even if the same as headname, if a synonym list follows. help_subtopic names must not include blanks.

synonym_list is a list of 1-31 character synonyms for the HELP subtopic. Synonyms must not include blanks.

Description:

This macro creates and formats the level 4 head (or table entry, if the level 4 head falls between ..:MAT and ..:END macros). If X is not specified, the macro creates a HELP subtopic, provided the preceding higher level head is a HELP topic. (Level 4 headings do not appear in the table of contents.)

$TEXT transforms level 4 headnames, entered in metalanguage used to described syntax, into useable HELP subtopics. These transformations are described later in this section.

Example:

```
..:L4H "Example:"
```

creates a hard-copy heading and generates the HELP subtopic EXAMPLE.

Usage Notes:

1. If headname exceeds line width in the output produced, headname is formatted on multiple lines just as any other text, with line breaks at blank characters.

Syntax Formats

Two kinds of syntax formats are used:

- COBOL-oriented syntax formats, used in COBOL, I-D-S/II and FPL documents.
- General syntax formats, used in all other documents.

Figure 7-1 is an example of a COBOL-oriented syntax format.

```
OBJECT-COMPUTER [HIS-SERIES-60] LEVEL-66-ASCII

    [                        {WORDS     }]
    [, MEMORY SIZE integer {CHARACTERS}]
    [                        {MODULES   }]

    [                                  {alphabet-name}]
    [                                  {STANDARD-1    }]
    [                                  {NATIVE        }]
    [, PROGRAM COLLATING SEQUENCE IS {ASCII         }]
    [                                  {EBCDIC        }]
    [                                  {GBCD          }]
    [                                  {HBCD          }]
    [                                  {JIS           }]
```

Figure 7-1. COBOL-oriented Syntax Format

Notice the brace or bracket on each line of a group, and the alignment of the braces and brackets that enclose the group.

Figure 7-2 is an example of a general syntax format as it appears in a user document. Note that the OR bar is preferred to the vertical stacking of options.

```
Syntax:

PASSWORD {OLD=oldpassword[,NEW=newpassword]|NEW=newpassword}
```

Figure 7-2. General Syntax Format in a User Document

Tables

Tables of seven lines or more are appropriate to be formatted as named tables. All named tables are created through the use of macros.

The table macro allows rapid entry of table text. The following list explains the three table types and when each is useful.

- Matrix - tables consisting of rows and columns. Up to 8 columns of data are permitted. Each row of text is entered as one line, with the # character as a column separator. Figure 7-3 shows a sample matrix table.

- Formatted 2-column - tables consisting of a table item in the left column and formatted text in the right column. Each item in the first column, entered as a level 4 head, occupies a separate line in the table. Text explaining each item starts on a following line (allowing a wide second column instead of one aligned beyond the longest item in the left column; this usually results in more compact tables). Each table item and accompanying text can be an individually selectable HELP subtopic. Figure 7-4 shows a sample formatted 2-column table.

- Unformatted - tables consisting of text in unformatted mode, i.e., the TEXT control word .FIF followed by text entered exactly as it is to appear in the output. Figure 7-5 shows a sample unformatted table.

Table A-1. ASCII Character Codes

Graphic	Octal	Decimal	Hexadec.	Definition
NULL	0'000'	D'000'	X'00'	NULL of time fill character
SOH	0'001'	D'001'	X'01'	Start Of Heading
STX	0'002'	D'002'	X'02'	Start of Text
ETX	0'003'	D'003'	X'03'	End of Text
EOT	0'004'	D'004'	X'04'	End of Transmission
.				
.				
.				

Figure 7-3. Matrix Table

```
                    Table 1-2.  Set Options

Option          Meaning

ACCESS=(({ALL|accountlist})[,controllist])

                ALL     is the control list is to apply to all
                other accounts.

                .
                .
                .
ACSVEH=(vehiclelist)[,controllist])

                vehiclelist     processors that are to be given
                access permissions.
                controllist     one of the following permissions:

                     DELRECORD
                     .
                     .
                     .
```

Figure 7-4. Formatted 2-Column Table

```
..:MAT "Table 13-1.  DELTA Directives"
.FIF
.spf 2
HOUSEKEEPING |EXECUTION    |EXECUTION |MEMORY DISPLAY |MODE    |MISC.
             |CONTROL      |TRACING   |& MODIFICATION |CONTROL|
-------------|-------------|----------|---------------|--------|-------
             |             |          |               |        |
Input/Output | Procedure   | HISTORY* | Variable      | ANLZ   | END*
Control:     | Breakpoint: | PLUGH*   | Oriented      | RUM    | HELP*
             |             | TRACE*   | Directives:   |        | LIST
COPY         | AT*         |          |               |        | PROTECT
 .
 .
 .
..:END
```

Figure 7-5. Source for Unformatted Table

Purpose of :MAT Macro

Table macros perform the following functions:

- Insert and center the table title, including blank line spacing.
- Automatically create running titles for continuation pages.
- Provide standard blank line spacing to separate the table from preceding and following text.
- Make logical page breaks.

In addition, these functions are performed for matrix and formatted 2-column tables:

- Insert and position column headings, including blank line spacing.
- Automatically create running headings for continuation pages.
- Perform all the positioning for the columns.

Before the table macros terminate, they always perform an .FIN, an .ALL and an .INL 0 so that the file builder is "reinitialized".

All named tables included in a text source file are delimited by :MAT and :END macro calls as follows:

```
..:MAT "detail"
table text
..:END
```

:MAT Macro

Format:

```
..:MAT "title[;col_head1[;[#]n;col_head2]...]"
```

Parameters:

title is the 1-61 character table title.

col_head1 is the column heading for the first column.

[#]n;col_head2 specifies the position and column heading for the second column. If # is specified, n is relative to the end of the last column heading. If # is omitted, n is the absolute column position for the heading. For example, 35;Description causes the head to be placed in column 35. #5;Description causes the head to be placed 5 positions after the previous column heading. Matrix tables may consist of up to 8 columns; formatted 2-column tables must have only 2 column heads specified.

Description:

This macro labels and lays out a table consisting of the following text (up to the ..:END macro). It also includes the table name in the list of tables and figures in the table of contents (if a table of contents is requested).

Example:

```
..:MAT "Table A-1.  ASCII Character Codes;Graphic;#5;Octal;#6;
Decimal;#7;Hexadec.;#7;Definition"
```

is the macro used to create the matrix table in Figure 7-3. The first line of source text in that table is entered in the source file as follows:

```
NULL#0'000'#D'000'#X'00'#NULL of time fill character
```

Following is the macro used to create the 2-column formatted table in Figure 7-4:

```
..:MAT "Table 1-2.  Set Options;Option;15;Meaning"
```

The first lines of source text in that table are entered in the source file as follows:

```
..:L4H "ACCESS=(({ALL|accountlist})[controllist])"
ALL^^^^is the control list that is to apply to all other accounts.
.
.
.
..:L4H "ACSVEH=(vehiclelist)[,controllist])"
vehiclelist^^^^processors that are to be given access permission.
.spb 0
controllist^^^^one of the following permissions:
.spf 0
     DELRECORD
     .
     .
     .
```

Note that the text between level 4 heads is indented (as requested in the :MAT macro) and formatted.

Usage Notes:

1. For matrix tables, text is left-justfied within the columns unless blanks are explicitly entered (by use of the up-arrow character). Adjacent pound signs are entered if a field in the matrix is to be left blank.

2. For unformatted tables, no column headings are specified in the :MAT macro. Any column headings must be included with the table text.

3. For matrix tables, the column headings and text (excluding # characters) cannot exceed 76 per line. For unformatted tables, each line of table text is limited to 76 characters.

Figures

Source text files should contain all artwork, not blank space for hand-drawn diagrams. (If desired, better quality artwork can overlay the artwork stored in the file if camera-ready master copies are being created.)

All figures included in a source text file are delimited by the :FIG and :FND macro calls formatted as follows:

```
..:FIG "fig_info"
figure text
..:FND
```

:FIG Macro

Format:

```
..:FIG "title[;n]"
```

Parameters:

title is the 1-61 character figure title.

n is the estimated number of lines in the actual figure. If n is not specified, $TEXT assumes the figure belongs on a single page.

Description:

This macro labels and lays out a figure consisting of the following text (up to the ..:FND macro). It also includes the figure name in the list of tables and figures in the table of contents (if the table of contents is requested).

$TEXT uses the following rules to determine figure layout:

1. If there is enough room on the current page, the figure is printed.
2. If there is not enough room on the current page, and the figure is less than a single page in length, a break to the next page occurs and the figure is printed.
3. If the figure is longer than a single page, the figure will begin printing on the current page, and all subsequent pages of the figure are printed with a figure title at the bottom of each page.

Example:

```
..:FIG "Figure 7-3.  Matrix Table;12"
```

created Figure 7-3 in this section.

Usage Notes:

1. The figure content must be kept within 78 character positions, starting at character position 1.
2. Areas in a figure can be forced to appear together by placing a .BRN above the area to be kept together. For example:

```
.BRN 10
```

means the next 10 lines must appear on the same page. Using an .SPF 0 is another way of forcing lines in a figure to print on the same page. .SPF 0 specifies that this is an inappropriate place to break the current block of text. (Using .SPB 0 specifies that this is an appropriate place to break the current block of text.)

3. Figures are created in .FIF mode and the .TRF is set so that the up-arrow prints. When the figure macro is exited, text is in .FIN mode, and the up arrow character is reset to the "blank replace" mode.

Figure Symbols

Figure 7-6 includes standard forms to be used in the construction of symbols used in electronically reproducible art. Some symbols can be represented in two ways: in an expanded or a condensed version. The expanded symbol uses dashes for top and bottom lines instead of underscores. The expanded symbol is preferred because it allows space for more text within the figure. Within any figure, it is important to be consistent about the use of condensed versus expanded symbols. It is also recommended that consistency be maintained within sections and, if possible, within a document.

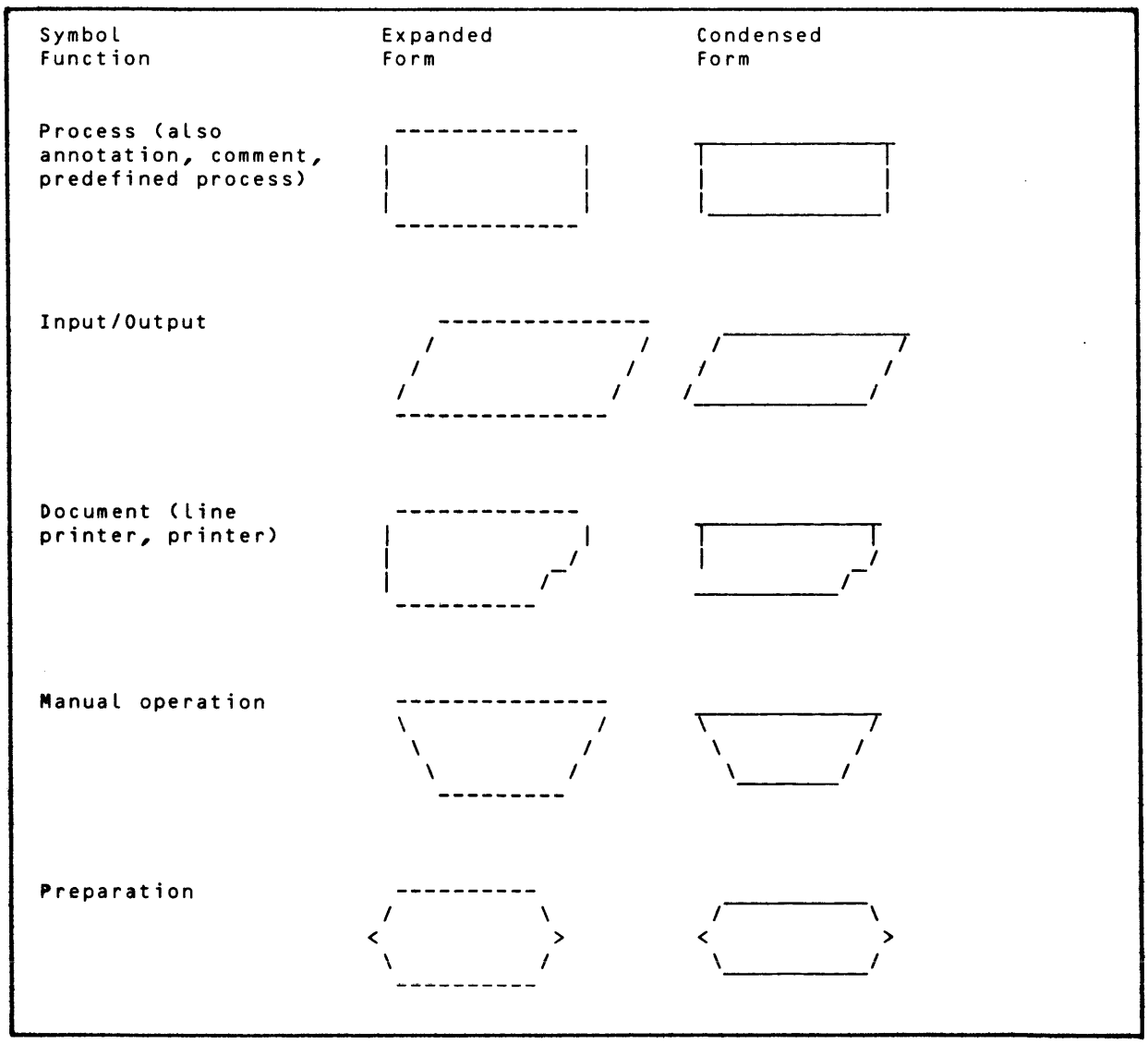


Figure 7-6. Figure Symbols (cont. next page)

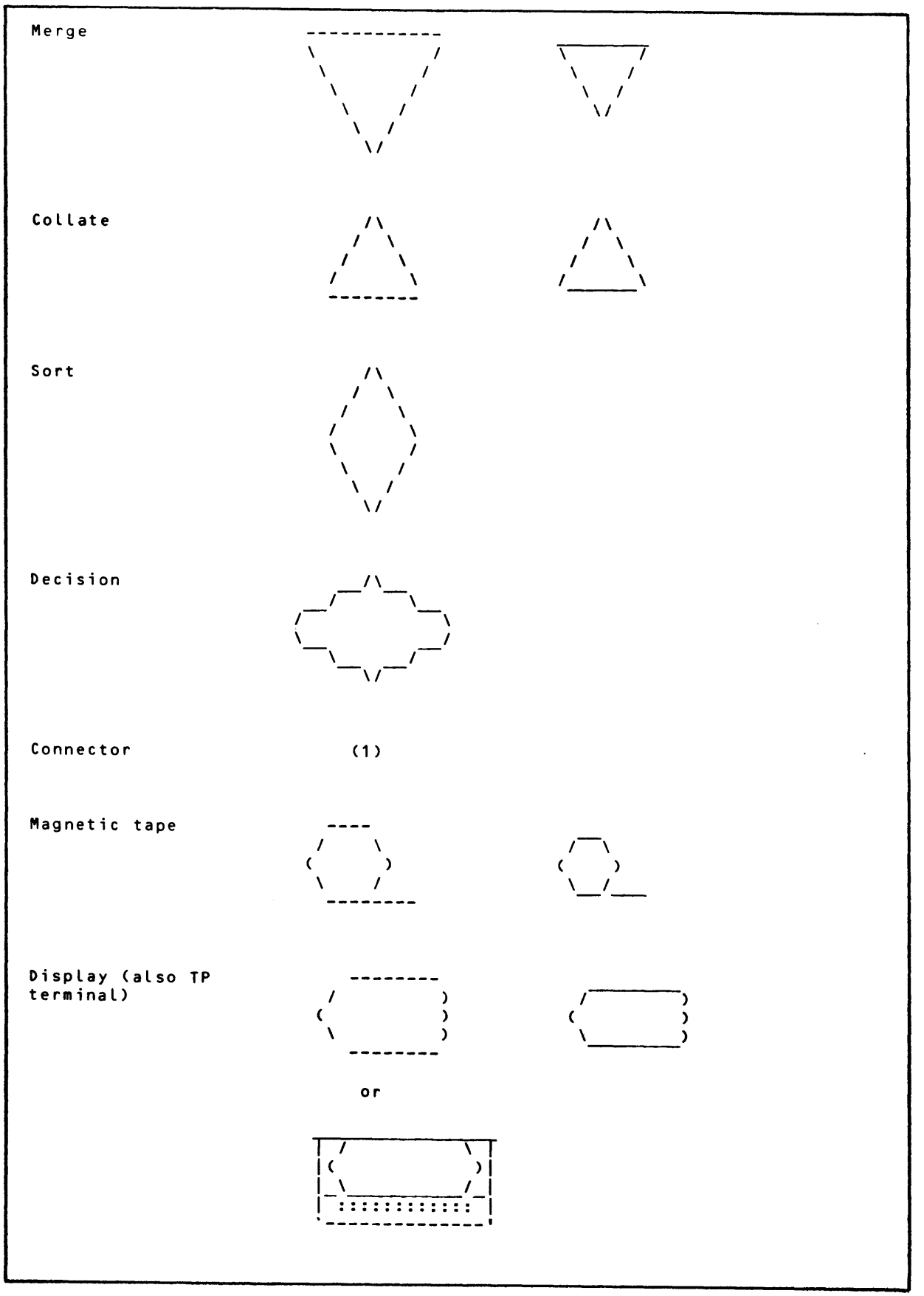


Figure 7-6. Figure Symbols (cont. next page)

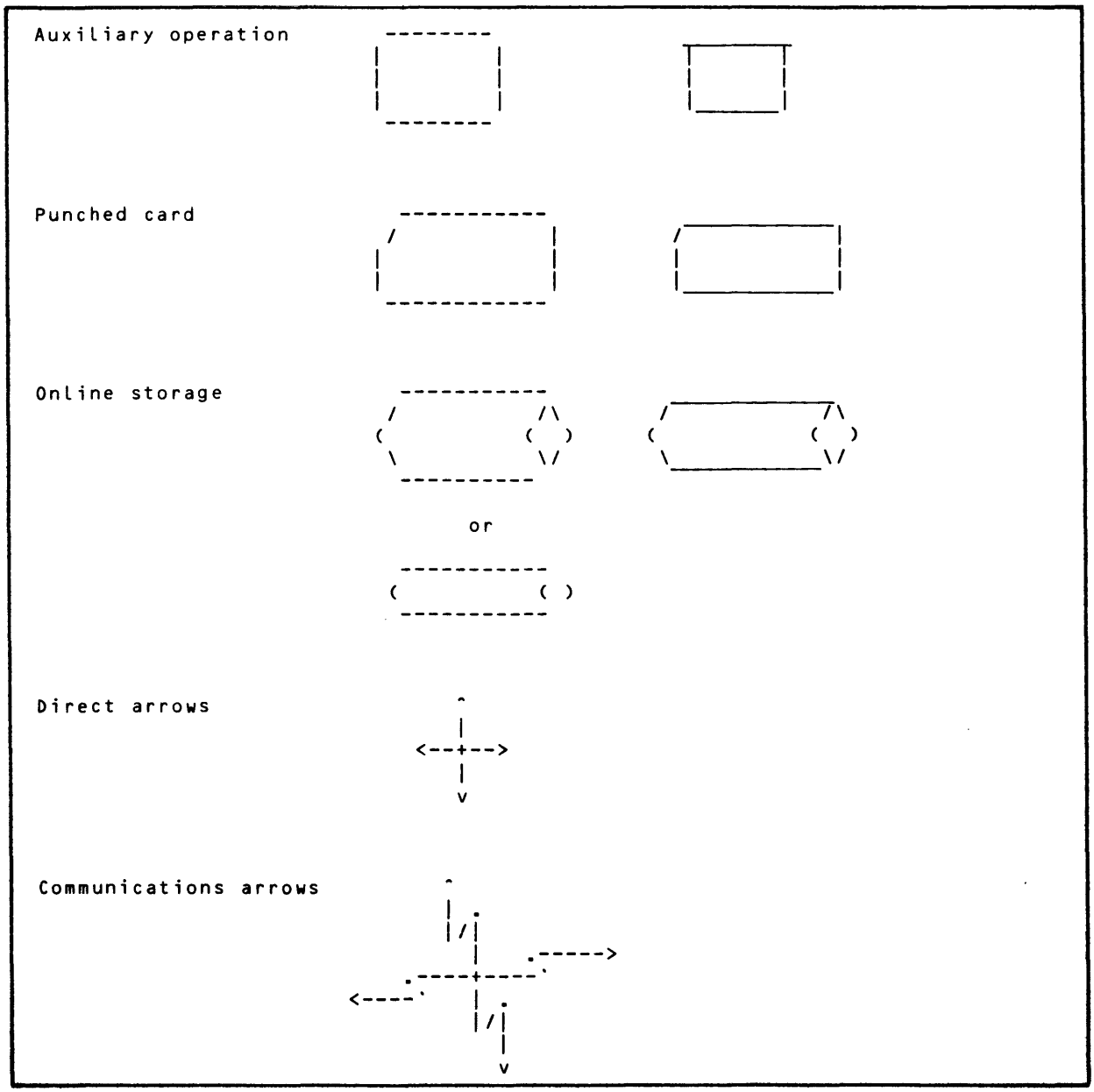


Figure 7-6. Figure Symbols

Index Entries

Indexes are created on request as part of document assembly. There is no limit to the number of items that can be included in an index. The index for a manual consists of:

- Automatic entries - all level 1, 2, and 3 heads contained in the document file.
- Specified entries - entries selected by the user by entering the :IDX macro.

:IDX Macro

Format:

```
..:IDX "term[;subterm]"
```

Parameters:

term is the term to be entered into the index.

subterm specifies that subterm is to appear in the index subordinate to term.

Description:

This macro creates an index entry. $TEXT, on request, produces an alphabetized, collated index. Upper and lowercase are considered identical; the first occurrence of a term will determine how an entry appears in the index. Thus, "ASDF" and "asdf" will be sorted together as "ASDF". Similarly "abcd" and "ABCD" will be sorted together as "abcd".

Plurals and suffixes of a term are not combined with the original term. Thus "Process" and "Processing" do not collate together.

Example:

```
..:IDX "BUILD Command"
..:IDX "BUILD Command;EDIT"
..:IDX "BUILD Command;IBEX"
```

creates a 2-level index, which could appear as follows for the terms shown above:

```
BUILD Command - 2-10
  EDIT - 3-18
  IBEX - 4-20
```

Usage Notes:

1. The :IDX macro must follow the paragraph to which it refers.
2. When a 2-level index is created by use of subterms, the term should be defined on one :IDX macro without subterms (as shown in the example).

Ending a Section

All source text files must end with the following sequence:

```
..:HLP ".BRP"
..:L1H ";X"
.*
```

This is the only time that the user is permitted to include a .BRP in a source text file.

Preparing On-line (HELP) Documentation

The CP-6 HELP facility is an on-line communication medium which provides quick information about processor capabilities. It is participatory in nature; that is, HELP issues messages in response to user requests for information. The HELP audience includes anyone who uses the CP-6 system from the novice to the experienced user. In addition to providing quick reference information, the HELP facility can be used for browsing. Through HELP browsing, the user can become familiar with the commands within a particular processor.

Reference manuals, stored as properly structured source text files, can be used to produce both hard-copy manuals and on-line documentation. $TEXT is designed to use level 1, 2, and 3 heads as HELP topics and level 4 heads as subtopics, unless otherwise directed. The task of encoding a source text file to also produce a HELP file is primarily a matter of suppressing topics that are inappropriate to HELP, supplying topic synonyms, and providing alternate wording for the on-line documentation (replacing references to other sections of the hard-copy document, for example).

Encoding a Source File

The HELP file creator uses encoding tools to transform a manual source file into a file containing a combination of manual information and HELP information. These tools enable the HELP processor to differentiate between material which is to appear in HELP and material which is to appear only in the manual. The semicolon (;) is the tool which allows HELP to make this distinction.

A semicolon within a heading informs the HELP processor that anything to the left of it is manual-only information and anything to the right of it is HELP-only information. Semicolons are always used when it is necessary to tell the HELP processor of one of the following conditions:

- A level head and its text are to be excluded from HELP.
- A HELP topic differs from the level head; also to supply synonyms for a HELP topic.
- A HLP macro is entered to supply HELP-only text.

Excluding Topics

A HELP topic is automatically made of each level 1, 2 and 3 head unless steps are taken to exclude the head as a topic. HELP makes no distinction between levels 1, 2 and 3. Any level 1, 2 and 3 headings which are to be excluded from HELP must be specifically flagged.

The following standard topics are good candidates for inclusion in HELP files:

- All statements, commands, verbs, and clauses. Their options with descriptions, syntaxes, syntax rules, usage notes and examples are subtopics.
- Lists of reserved words, verb categories and compiler options. The description of each entry in the list is a candidate for a subtopic.
- Miscellaneous subjects such as notation conventions.

Good candidates for exclusion from HELP include:

- Conceptual material
- Tutorial information
- References to Manuals, Sections, Tables and Figures

For example, a level 1 heading INTRODUCTION and all text associated with it are not to be included in HELP. The HELP creator adds ;X to the heading to indicate that INTRODUCTION is not to be included in HELP. The heading then appears as:

```
..:L1H "Introduction;X"
```

The semicolon signifies that all information to the right of it is HELP-only material. The X signifies that the heading and subsequent text are excluded from HELP. The ;X is automatically turned off when the next level 1, 2 or 3 head is encountered. If there are subsequent level 2 and 3 heads under INTRODUCTION which are to be excluded from HELP, they must also be flagged with a ;X.

Topic Names and Synonyms

A level 1, 2, or 3 head becomes a topic name by default. Any level 4 head subordinate to a level 1, 2, or 3 head included in HELP becomes a subtopic name by default. Alternate topics, subtopics, and synonyms can be provided by entering them to the right of the ;.

If a single word appears on the right side of the ;, it is the topic name (instead of the heading name). If multiple values appear on the right side of the ; the first is the topic name and the subsequent values are synonyms. Each synonym is separated from the preceding value by a space.

No single topic name, synonym or subtopic name may exceed 31 characters. The underscore is used as a connector (e.g., USAGE_NOTES).

Names

Unless changed, the name of a topic (or subtopic) is the name of the heading. Topic names appear in the automatically generated HELP TOPICS message for a HELP file. It is important that the manual heading be examined to determine its appropriateness as a listed item in the TOPIC tabular message.

Synonyms

A synonym is an alternate name for a topic or subtopic. Synonyms are assigned in two cases:

1. When a command (statement, etc.) can be abbreviated in one or more ways.

2. When a concept can be named in more than one way.

Note: Whenever a synonym is included, the topic (or subtopic) must always appear as the first word to the right of ; even if it is the same as the level head.

Command Abbreviations

For a good HELP file, each command that has abbreviations should be retrievable through any of its abbreviated forms. Messages are retrieved through that means. Therefore if a command is the topic, the HELP coding must include all valid abbreviations for the command. These abbreviations are introduced as name synonyms.

The COPYALL command has the following syntax:

```
C[OPY]A[LL]
```

The valid abbreviations for the COPYALL command are:

```
CA,CAL,CALL,COA,COAL,COALL,COPA,COPAL,COPYALL,COPYA,COPYAL
```

The HELP portions of the COPYALL heading must contain all of these combinations as synonyms for the user to be able to access the COPYALL topic using any one of the valid abbreviations. The heading appears as:

```
..:L2H "COPYALL Command;COPYALL CA CAL CALL COA COAL COALL
COPA COPYA COPYAL"
```

Notice that the list begins with the full command form "COPYALL". The full command form is always listed first because only the first form will be listed in the HELP TOPICS message.

Conceptual Synonyms

Conceptual synonyms give the user a choice of ways to access a topic which may have several different names. For example, suppose a portion of text appearing under the heading "Possible Error Conditions" is to be included in the HELP file. The heading appears in the manual as:

```
..:L1H "Possible Error Conditions"
```

One user may think of the word "errors" while another user may think of the words "warnings" or "problems". Using synonyms, the HELP creator can assign all three names to the topic. The heading is changed to appear as:

```
..:L1H "Possible Error Conditions; ERRORS WARNINGS PROBLEMS"
```

Encoding Subtopics

A subtopic is the name of a retrievable submessage associated with a HELP topic. Entering the HELP request for a command followed by a subtopic produces only the subtopic information.

Consistent organization and presentation of material is critical for successful use of subtopics. Messages in one subtopic must be the same structure as similar messages in other subtopics.

The HELP file creator must determine:

- If any subtopics need to be excluded or modified.
- Appropriate subtopic names and synonyms.
- What table entries to include as subtopics.

Creating Subtopics

Any level 4 head appearing under a higher level head that is a HELP topic is automatically a subtopic. Any level 4 head appearing under a higher level head that is not a HELP topic is automatically excluded from HELP. To create a subtopic for a subject appearing at a level 2 or 3 head requires the insertion of a level 4 head of HELP-only information. For example, suppose LITERALS appears in the manual file as:

```
..:L2H "Literals"
       .
       . text
       .
..:L3H "Nonnumeric Literals"
       .
       . text
       .
..:L3H "Numeric Literals"
       .
       . text
       .
```

The HELP creator wants to make NONNUMERIC and NUMERIC subtopics of LITERALS. The manual file is modified to appear as:

```
..:L2H "Literals"
       .
       . text
       .
..:L3H "Nonnumeric Literals;"
..:L4H ";NONNUMERIC"
       .
       . text
       .
..:L3H "Numeric Literals;"
..:L4H ";NUMERIC"
       .
       . text
       .
```

The topic LITERALS now has the subtopics NONNUMERIC and NUMERIC. Adding the semicolon at the level 3 head prevents those headings from becoming topics in themselves while preserving the text under them for the HELP file. Adding the level 4 head with the semicolon in front of the heading contents creates a HELP subtopic name which does not appear in the manual but does appear in HELP.

Subtopics Within Tables

Named subtopics are useful also for direct access to items such as compiler options. In a table of compiler options, each option is a level 4 heading. The named topic, in this case the name COMP_OPTIONS, is a level 1, 2 or 3 heading (i.e., ..:L1H ";COMP_OPTIONS" is specified immediately following the :MAT macro).

Entering only the named topic produces a list of all the options in the table. For example, to obtain a list of all compiler options for PL6, the user enters:

```
HELP (PL6) COMP_OPTIONS
```

A list of all the compiler options in PL6 is produced. The user decides to view only the NFORMAT option for the PL6 compiler and enters:

```
HELP (PL6) COMP_OPTIONS NFORMAT
```

The syntax for the NFORMAT option appears along with a brief description of the option's function.

Automatic Transformation of Subtopics

The HELP processor examines each character in the heading separately to transform the manual entries to HELP subtopics. In the above example, NFORMAT actually appears in the manual as NFOR[MAT]. Braces and brackets are removed from the heading. Any OR bars ("|") are transformed into blank spaces and anything following a "(", ":", ";", "=" or " " is deleted from HELP. Whatever remains is converted to uppercase. Table 7-3 illustrates the conversion from the source file to the HELP file.

Table 7-3. Examples of :L4H Transformation

Heading	Converted Subtopic Name
..:L4H "ABC[DE]"	ABCDE
..:L4H "{AB|CD}"	AB CD
..:L4H "{AB[C]|DE}"	ABC DE
..:L4H "ABC=n"	ABC
..:L4H "ABC:DE"	ABC
..:L4H "Example:"	EXAMPLE

:HLP Macro

Format:

```
..:HLP "[manual_text][;help_text]"
```

Parameters:

manual_text is text that is to appear in the manual, but not in the HELP file.

help_text is text that is to appear in the HELP file, but not in the manual.

Description:

This macro allows entry of alternate wording for the hard-copy manual and for the HELP file.

Example:

```
    ..:HLP "Refer to Section 3 of this manual for details;"
```

is text for the manual only, since references to Section 3 are inappropriate for the user of the HELP file.

```
    ..:HLP ";HELP messages for each option can be obtained"
    ..:HLP ";by entering HELP (PL6) COMP_OPTIONS option"
```

is text for the HELP file only since references to HELP topics are inappropriate for the manual.

```
    ..:HLP "See Table 2-2 for a list of compiler options;Enter
    HELP COMP_OPTIONS for a list of compiler options"
```

provides alternate wording for the manual and the HELP file. The example is functionally identical to:

```
    ..:HLP "See Table 2-2 for a list of compiler options;"
    ..:HLP ";Enter HELP COMP_OPTIONS for a list of compiler options"
```

Creating a HELP File

The HELP file is built through $TEXT. The HELP file creator invokes $TEXT and directs the output of processing to a file. $TEXT in turn, invokes the X account program HERMAN which creates the HELP file.

The following dialog illustrates how a HELP file can be generated from one source file.

Prompt	Action
!XEQ $TEXT.:DOCUM	Call the document assembly program.
FASTEXT B03	
Files to Format>CE29_01	Enter the file name.
In account>:MANUALS	Enter the account the manual files are in.
Device or Destination>Z0_COBOL1_HELP	Enter the name of the file to be created, Z0_COBOL1_HELP. If that exists, a prompt is issued asking if you want to write over it; respond with a "Y". By convention, this file name is of the form fcg_processorname_HELP, where fcg is the Functional Code Group (FCG). The "COBOL1" portion of the file name indicates the manual Section (1) for a HELP section file.
TEXT OPTIONS>	Enter a carriage return <CR>.
DRAFT or FINAL Format>H	Enter H for HELP.
Pagesize (type of paper)	Enter a <CR>.
Number of Copies >	Enter a <CR>.
Extra Files >I	Enter an I to get a HELP index.

```
Do you want these files for :MANUALS:

ORG TY  GRAN  NGAV     REC   LAST MODIFIED   NAME
KEY       7     0     541 11:50 NOV 23 '81  CE29_01

creating HELP source file named Z0_COBOL1_HELP
creating a readable HELP file named HELP:COBOL1:

Enter Y to send the job, C to reenter the values, or <CR> to exit.
Go for it?>Y                          Enter a Y to create a HELP file.
```

Once HELP files for several sections have been created and tested, the HELP file creator can invoke $TEXT in document assembly mode specifying multiple section files to produce a single HELP file.

Section 8

Techniques: Central System

This section discusses a number of interfaces to central system software and includes sample code.

Accessing the JIT

The Job Information Table (JIT) contains system related information about an individual CP-6 user. From logon to logoff time, the JIT accumulates and retains facts about a user's session. The information contained in the JIT is useful for programmers writing programs to run on the CP-6 system.

The JIT is the area in memory that gathers and records specific facts about the user such as the users's account, mode, name, sysid and accounting information.

The JIT is accessed in several different ways; via the Linker-built pointer BJIT, the JIT can be examined through DELTA by using linkage segment number one ($LS1) or by using $JIT. Any command processor automatically has modification access to the JIT. A run unit that resides in the :SYS account and that has been linked with JIT alteration privilege also has modification access to the JIT. Appendix A include a description JIT fields and a DRAW of the structure. Figure 8-1 illustrates a PL-6 subroutine which returns a user's logon account and user name from the JIT to a FORTRAN program.

```
!C JIT_MAIN_SIF
      PROGRAM JIT
      CHARACTER * 8 MYACCT
      CHARACTER * 12 MYUNAME
      CALL GETJIT (MYACCT,MYUNAME)
      WRITE (108,1000) MYACCT, MYUNAME
 1000 FORMAT (1X,A8,1X,A12)
      STOP
      END
!C JIT_SUB_SI6
/*M* Subroutine returns logon account and
     user name from JIT to FORTRAN program */

GETJIT: PROC (ACCT, UNAME);

DCL ACCT CHAR(8);
DCL UNAME CHAR(12);

DCL B$JIT$ PTR SYMREF;

%INCLUDE B$JIT;

ACCT = B$JIT$ -> B$JIT.ACCN;

UNAME = B$JIT$ -> B$JIT.UNAME;

RETURN;

END GETJIT;
!FORTRAN JIT_MAIN_SIF OVER *G(NLS)
FORTRAN 77 VERSION B04  AUG 20 '82
!PL6 JIT_SUB_SI6 INTO *G(SR(.:LIBRARY),NLS)
PL6 B02  here at 15:32 AUG 20 '82


    No errors detected in file JIT_SUB_SI6

!LINK *G OVER *L
*   :SHARED_COMMON.:SYS (Shared Library) associated.
*  No linking errors.
*  Total program size = 3K.
!*L

XWKPL6 876KWIK
 *STOP*
```

Figure 8-1. Accessing the JIT Using PL-6 Subroutine

Accessing the Task Control Block (TCB)

The Task Control Block (TCB) contains system related information about a running program. From fetch to rundown time, the TCB accumulates and retains facts about a user's program including information about monitor service ALTRETURNs, interrupts, faults and asynchronous events.

The TCB is accessed in several different ways; via the Linker-built pointer BTCB, the TCB can be examined through DELTA by using linkage segment number fifteen ($LS15) or by using $TCB. The layout of the TCB and the structures related to it are described in the Monitor Services Reference Manual, in the section on Exception Condition Services.

Figure 8-2 shows a PL-6 subroutine which accesses the TCB, returning a monitor service error code from the altreturn frame.

```
!C ERR_SI6
/*M* Get error code from TCB, print using M$ERRMSG */
ERRMSG: PROC MAIN;

     %INCLUDE CP_6;
     %INCLUDE CP_6_SUBS;

DCL M$DO DCB;
DCL MY_ERROR_BUF CHAR(255) STATIC;
DCL B$TCB$ PTR SYMREF;

     %FPT_ERRMSG    (FPTN=ERROR_PRINT,
                         BUF=MY_ERROR_BUF,
                         OUTDCB1=M$DO,
                         CODE=NIL);
     %B$TCB;

     %B$ALT;

          ERROR_PRINT.CODE_ = VECTOR(B$TCB$->B$TCB.ALT$->B$ALT.ERR);
          CALL M$ERRMSG (ERROR_PRINT) ALTRET (HMMM);
HMMM:     CALL M$XXX;
END ERRMSG;
!PL6 ERR_SI6 OVER *G(NLS,SR(.:LIBRARY))
PL6 B02  here at 14:01 SEP 02 '82


    No errors detected in file ERR_SI6

!RUN *G
*  :SHARED_SYSTEM.:SYS (Shared Library) associated.
*  No linking errors.
*  Total program size = 3K.
 MMP-M00606-0  Attempt to free more space than is in data segment.
  M$XXX issued by user.

Note:  MMP-M00606-0 is the error given to :SHARED_SYSTEM when it releases all
    the memory in AUTO prior to starting a PL-6 MAIN program.
```

Figure 8-2. Accessing the TCB Using PL-6 Subroutine

Break Handling

Break handling allows a time sharing user to hit the break key while a program is running and either check on the program's progress, or provide for an alternate routine or exit. To create a break handling routine, a user must provide a PL-6 subroutine which calls the M$INT Monitor Service to establish and setup an address for the break handler. The break handler subroutine must be a PL-6 asynchronous procedure (PROC ASYNC). Figure 8-3 shows a FORTRAN program with a PL-6 subroutine that handles a break by clearing the break frame from the Task Control Block (via the call to M$CLRSTK) and forcing the program to exit on a special path. Note the call to the M$TRMPRG monitor service which is included to avoid four breaks simulating CONTROL-Y; see the discussion of M$TRMPRG in the CP-6 Monitor Service Reference Manual for details.

```
!C INT_SIF
      PROGRAM INT
      CALL SETUPINT(110S)
      DO 100 I=1,10
      CALL SLEEP(10)
      OUTPUT I
  100 CONTINUE
      OUTPUT 'ALL DONE'
      CALL EXIT(0)
  110 OUTPUT 'BACK FROM BREAK'
      STOP
      END
      SUBROUTINE BREAKH(*)
      OUTPUT 'IN FORTRAN BREAK ROUTINE'
      RETURN 1
      END
!C INT_SI6
/*M* INT HANDLERS FOR EXAMPLE PROGRAM */
SETUPINT: PROC (ENTPOINT);
DCL ENTPOINT PTR;
DCL INTPTR PTR STATIC SYMDEF;
%INCLUDE CP_6;

DCL MY_INT ENTRY ASYNC;

      %FPT_INT (FPTN=MY_INT_FPT,UENTRY=MY_INT);

        INTPTR = ENTPOINT;
        CALL M$INT (MY_INT_FPT) WHENALTRETURN DO; END;
        RETURN;
END SETUPINT;
%EOD;
MY_INT: PROC ASYNC;
DCL INTPTR PTR SYMREF;
DCL BREAKH ENTRY (1);
      %INCLUDE CP_6;
      %FPT_TRMPRG (DCB=M$UC,RSTBRK=YES);

        CALL M$CLRSTK;
        CALL M$TRMPRG (FPT_TRMPRG) WHENALTRETURN DO; END;
        CALL BREAKH (INTPTR);

        RETURN;
END MY_INT;
%EOD;
SLEEP:  PROC (TIME);
%INCLUDE CP_6;
       %FPT_WAIT (FPTN=ZZZZZ);
DCL TIME SBIN;
```

Figure 8-3. Break Handling Via PL-6 ASYNC Procedure (cont. next page)

```
        ZZZZZ.V.UNITS#=TIME;
        CALL M$WAIT (ZZZZZ);
        RETURN;
END SLEEP;
!FORTRAN INT_SIF OVER *G(NLS)
FORTRAN 77 VERSION C00  NOV 03 '83
!PL6 INT_SI6 INTO *G(NLS,SR(.:LIBRARY))
PL6 B02  here at 15:56 NOV 03 '83
      No diagnostics issued in procedure SETUPINT

      No diagnostics issued in procedure MY_INT

    No errors detected in file INT_SI6.

!LINK *G OVER *L
*  :SHARED_COMMON.:SYS (Shared Library) associated.
*  No linking errors.
*  Total program size = 3K.
!*L

I = 1
I = 2
IN FORTRAN BREAK ROUTINE
BACK FROM BREAK
 *STOP*
```

Figure 8-3. Break Handling Via PL-6 ASYNC Procedure

Trap Handling

Trap control allows the user to gain control in the case of a hardware detected fault (trap). Trap handling allows the user to detect and accommodate fault conditions. Trap control is usually taken by highly generalized, widely used libraries of utility subroutines. To create a trap handling routine, a user must provide a PL-6 subroutine which calls the M$TRAP Monitor Service to establish and set-up an address for the trap handler. The trap handling routine must be a PL-6 asynchronous procedure (PROC ASYNC). See the Monitor Services Reference Manual for a complete description of M$TRAP. Figure 8-4 shows a FORTRAN program with a PL-6 subroutine that handles a trap by clearing the trap frame from the Task Control Block (via a call to M$CLRSTK) and forcing the program to exit on a special path.

```
!FORTRAN TRAP_SIF OVER *F(LS,OU)
FORTRAN 77 VERSION B04  SEP 20 '82
*     1.000>     1:          PROGRAM TRAP
      2.000>     2:          CALL SETTRAP(100S)
      3.000>     3:          OUTPUT 'GOING TO DIVIDE BY ZERO'
      4.000>     4:          CALL DIVIDE(4,0,RESULT)
      5.000>     5:          OUTPUT 'BACK FROM SUBROUTINE'
      6.000>     6:          STOP 'ME BEFORE I KILL AGAIN'
      7.000>     7:    100   OUTPUT 'GOT A ZERO DIVIDE'
      8.000>     8:          STOP
      9.000>     9:          END
 ERRORS FOUND       : 0        TOTAL ERRORS FOUND: 0

*    10.000>     1:          SUBROUTINE DIVIDE (DIVIDEN,DIVISOR,QUOTI)
     11.000>     2:          QUOTI=DIVIDEN/DIVISOR
     12.000>     3:          RETURN
     13.000>     4:          END
 ERRORS FOUND       : 0        TOTAL ERRORS FOUND: 0
```

Figure 8-4. Trap Handling Via a PL-6 Subroutine (cont. next page)

```
*   14.000>      1:          SUBROUTINE ZERODIV (*)
    15.000>      2:          OUTPUT 'WHOOPS! ZERO DIVISION ENCOUNTERED'
    16.000>      3:          RETURN 1
    17.000>      4:          END
 ERRORS FOUND       : 0        TOTAL ERRORS FOUND: 0

!PL6 TRAP_SI6 OVER *6(LS,OU,SR(.:LIBRARY))
PL6 B02  here at 15:55 SEP 20 '82
    1.000         1          /*M* SET UP TRAP CONTROL */
    2.000         2          SETTRAP: PROC (LABEL);
    3.000         3
    4.000         4          %INCLUDE CP_6;
    5.000        81     1    DCL LABEL PTR;
    5.100        82     1    DCL MY_LABEL PTR STATIC SYMDEF;
    6.000        83     1    DCL MY_TRAP ENTRY ASYNC;
    7.000        84
    7.010        85                %FPT_TRAP    (FPTN=SETUP_TRAP,
    7.020        86                                  ARITHMETIC=MY_TRAP,
    7.030        87                                  DIVIDE_CHECK=TRAP);
    7.040       110
    7.050       111     2    CALL M$TRAP (SETUP_TRAP) WHENALTRETURN DO; END;
    7.055       112     1    MY_LABEL=LABEL;
    7.060       113
    7.070       114     1    RETURN;
    7.080       115
    7.090       116     1    END SETTRAP;
    8.000       117          %EOD;

      No diagnostics issued in procedure SETTRAP

    9.000         1          MY_TRAP: PROC ASYNC NOAUTO;
   10.000         2
   11.000         3     1    DCL ZERODIV ENTRY(1);
   12.000         4     1    DCL MY_LABEL PTR SYMREF;
   13.000         5
   14.000         6          %INCLUDE CP_6;
   15.000        83
   16.000        84     1              CALL M$CLRSTK;
   17.000        85
   18.000        86     1              CALL ZERODIV (MY_LABEL);
   19.000        87
   20.000        88     1              RETURN;
   21.000        89     1    END MY_TRAP;

      No diagnostics issued in procedure MY_TRAP

    No errors detected in file TRAP_SI6

!LINK *F,*6 OVER *L
*  :SHARED_COMMON.:SYS (Shared Library) associated.
*  No linking errors.
*  Total program size = 3K.
!*L

GOING TO DIVIDE BY ZERO
WHOOPS! ZERO DIVISION ENCOUNTERED
GOT A ZERO DIVIDE
 *STOP*
```

Figure 8-4. Trap Handling Via a PL-6 Subroutine

Associating or Linking to Another Program

Associating another program with a currently running program can be accomplished by one of three monitor services:

- M$LINK passes control to another program which runs and then returns control to the original program following the call to M$LINK.
- M$LDTRC passes control to another program; context from the original program is not saved.
- M$ALIB passes control to a shared library, an alternate shared library, or a debugger.

The uses for M$LINK and M$LDTRC are straightforward. One point illustrated in Figure 8-5 is particularly important to note: setting of the command line DCBs (#1, #2, #3, #4) for a newly associated program can be performed by a call to the M$YC monitor service. Figure 8-5 illustrates how this is done in a PL-6 subroutine (which may be called within a FORTRAN program, for example).

```
DRIBBLE ON @  13:41 01/12/83
!B *2
EDIT B03 HERE
    1.000 THIS IS A TEST.
    2.000 IT IS ONLY A TEST.
    3.000
!C *2 OVER *1
 ..COPYing
!C LINK_SI6
/*M* LINK_TEST */
/*X* DMR,PLM=5,IND=3,CTI=3,SDI=3,MCL=10,CSI=0,ECI=0 */
LINK_TEST:PROC MAIN;

%INCLUDE CP_6;
%INCLUDE CP_6_SUBS;

     %FPT_YC   (FPTN=SET_1,
                     CMD=SET_1_CMD,
                     NOERR=YES);

     %FPT_LINK (FPTN=LINK_TUNA,
                     CMD=TUNA_CMD,
                     NAME=TUNA_NAME,
                     ACCT=TUNA_ACCT);
DCL 1 TUNA_CMD STATIC,
     2 * UBIN BYTE CALIGNED INIT(SIZEC('TUNA.X (LEN=79)')),
     2 * CHAR(0)   CALIGNED INIT('TUNA.X (LEN=79)');
          %VLP_NAME (FPTN=TUNA_NAME,
                          NAME='TUNA');
          %VLP_ACCT (FPTN=TUNA_ACCT,
                          ACCT='X       ');

DCL SET_1_CMD CHAR(0) STATIC INIT('!SET #1 *1');

     CALL M$YC(SET_1)
     WHENALTRETURN
     DO;
          CALL M$XXX;
          END;

     CALL M$LINK (LINK_TUNA)
     WHENALTRETURN
     DO;
          CALL M$XXX;
          END;

     CALL M$EXIT;

END LINK_TEST;
!PL6 LINK_SI6 OVER *G(NLS)
PL6 B02  here at 13:41 JAN 12 '83
    No errors detected in file LINK_SI6.

!LINK *G OVER *L
*   :SHARED_SYSTEM.:SYS (Shared Library) associated.
*  No linking errors.
*  Total program size = 3K.
!*L
!C *1
THIS IS A TEST.  IT IS ONLY A TEST.
!DONT DRIBBLE
DRIBBLE OFF @ 13:42 01/12/83
```

Figure 8-5. DCBs for Program Called by M$LINK/M$LDTRC

M$ALIB is typically used in cases such as these:

- To dynamically change the shared library in use.
- To dynamically associate an Alternate Shared Library, such as I-D-S/II, based on need rather than automatically associating the data base manager at invocation of a program.
- To dynamically associate a debugger to diagnose an error condition.

The sample subroutine shown in Figure 8-6 associates the CP-6 debugger, DELTA, to ascertain the location at which an error occurred. This example shows how to obtain the Instruction Counter (from the TCB) for the error location and return it for display by the erring program.

```
!C ALIB_SI6
/*X* DMR,PLM=5,IND=3,CTI=3,SDI=3,MCL=10,CSI=0,ECI=0 */
ALIB_TEST: PROC MAIN;

DCL SETXCON ENTRY;

DCL PTR$ PTR STATIC INIT(ADDR(ARRAY));
DCL ARRAY(0:1024) SBIN STATIC;
DCL WORD_ SBIN BASED(PTR$);
DCL X_ SBIN WORD;

     CALL SETXCON;                        /* SETUP THE XCON ADDRESS */

     DO WHILE('1'B);                      /* I KNOW THIS CODE DOESN'T WORK */
                                          /* I WANT TO FORCE A FAULT SO    */
        PTR$=PINCRW(PTR$,-1);             /* MY XCON ROUTINE WILL BE       */
        X_ = PTR$->WORD_;                 /* ENTERED                       */
        END;                              /* DO FOREVER */

END ALIB_TEST;
%EOD;
SETXCON: PROC;
DCL MY_XCON ENTRY ASYNC;
%INCLUDE CP_6;
     %FPT_XCON (FPTN=SET_XCON,
                    UENTRY=MY_XCON);

     CALL M$XCON (SET_XCON)
     WHENALTRETURN
     DO;
        END;

     RETURN;
END SETXCON;
%EOD;
MY_XCON: PROC ASYNC;
%INCLUDE CP_6;
     %FPT_XCON (FPTN=RESET_XCON);

     %FPT_ALIB (FPTN=PRINT_ADDRESS,
                    CMD=DELTA_CMD.NAME#,
                    ECHO=YES,
                    DLIB=YES,
                    LIBNAME=DELTA_NAME,
                    RETRN=YES);

          %VLP_NAME(FPTN=DELTA_CMD,
                         NAME='EVAL .000000\R');

          %VLP_NAME(FPTN=DELTA_NAME,
                         NAME='DELTA');

     %FPT_ERRMSG    (FPTN=MY_ERROR,
                         BUF=ERROR_BUF,
                         OUTDCB1=M$DO);

DCL B$TCB$ PTR SYMREF;

                                          /* BASED STRUCTURES TO LOOK AT TCB */
     %B$TCB;
     %B$XCON;
     %B$EXCFR;
```

Figure 8-6. Associating DELTA to Dump I.C. (cont. next page)

```
DCL I SBIN;
DCL J SBIN;
DCL 1 X(0:5),
       2 Z_ UBIN(3) UNAL;
DCL X_ REDEF X UBIN(18) UNAL;
DCL ERROR_BUF CHAR(140) STATIC;
DCL M$DO DCB;

                                        /* PRINT ERROR MESSAGE THAT
                                             CORRESPONDS TO ERROR CODE
                                             IN XCON FRAME ON TCB */
     MY_ERROR.CODE_ = VECTOR(B$TCB$->B$TCB.STK$->B$XCON.ERR);

     CALL M$ERRMSG(MY_ERROR)
     WHENALTRETURN
     DO;
        END;
                                        /* GET IC FROM TCB FRAME */
     X_ = B$TCB$->B$TCB.STK$->B$EXCFR.IC;

                                        /* CONVERT IC TO PRINTABLE
                                             FORM SO IT CAN BE PASSED
                                             TO DELTA */
     J=6;
     DO I = 0 TO 5;
        CALL INSERT (DELTA_CMD.NAME#,J,1,BINASC(X.Z_(I)+48));
        J=J+1;
        END;                            /* DO I=0 TO 5 */

                                        /* GO ASK DELTA TO EVALUATE
                                             THE IC */
     CALL M$ALIB (PRINT_ADDRESS)
     WHENALTRETURN
     DO;
        END;

     CALL M$XCON (RESET_XCON)
     WHENALTRETURN
     DO;
        END;

     CALL M$EXIT;
END MY_XCON;
!PL6 ALIB_SI6 OVER *G
PL6 B02  here at 13:00 JAN 12 '83
    No errors detected in file ALIB_SI6.

!LINK *G OVER *L
*  :SHARED_SYSTEM.:SYS (Shared Library) associated.
*  No linking errors.
*  Total program size = 4K.
!*L
 HFA-M00520-6   Missing Page fault
    $ALIB >EVAL .004017\R
 = ALIB_TEST :16,,.1 [ASSIGNMENT]
!DONT DRIBBLE
DRIBBLE OFF @ 13:01 01/12/83
```

Figure 8-6. Associating DELTA to Dump I.C.

Shared Data Segments

Data segments may be shared by separate programs that run independently or by programs which call one another via the M$LINK monitor service. The two types of data segment sharing are discussed below.

Sharing COMMON between M$LINKed Programs

Programs that call one another via the M$LINK monitor service may pass data in the COMMON data segment, i.e., data segment 2. To allocate the COMMON data segment, the M$GDS monitor service can be called with SEGSIZE set; this returns (via the RESULTS area) a vector framing the COMMON data segment. Once the data segment is allocated, M$GDS (with SEGSIZE=0) simply returns a vector framing the segment.

Sharing Data Segment between Independent Programs

Programs that have a need to share up to 256K of data may use a shared data segement to provide fast access to that data. By using the M$OPEN monitor service, the programs may establish or access a shared data segment. The programs can access the data in the shared data segment without performing physical I/O for each access. Parameters that must be specified on the call to M$OPEN are as follows: DCB, NAME, ACCT, FUN which are set as appropriate and ORG=RANDOM, ACS=DSn, and ASN=FILE. The file will be opened and mapped into the specified data segment.

To access the data segment, the programs use either the SYMREFed pointer BDSn or the M$GDS monitor service called specifying SEGSIZE=0 and RESULTS= a VLP_VECTOR structure. The VLP_VECTOR.SEGID field should be filled in with %DSnSID from the B_SEGIDS_C include file. The call to M$GDS returns a vector (VLP_VECTOR) framing the data segment.

The M$CLOSE and M$EXTEND monitor services may also be used to manipulate a shared data segment.

Programs that share a data segment in this way must take complete responsibility for assuring the integrity of the data. Such programs need to include locking mechanisms for this purpose (e.g., M$ENQ to assure that the item or group of items it accesses contains current information and that no updates are lost).

Virtual Data Segments

A virtual data segment is much like any other CP-6 data segment: it is an addressable portion of memory which may be used to store and access user data. Virtual data segments differ from standard data segments in several ways:

- A virtual data segment may be much larger than a standard data segment. Virtual data segments may contain up to 4 gigawords (4,294,967,296) of data each; the user is permitted to have up to 3 virtual data segments in existence at any particular time, yielding a total of 12 gigawords of address space.

- As their name implies, virtual data segments reside in memory in a virtual sense, and not necessarily in a real sense. Since the size of a virtual data segment may easily exceed the memory-usage authorization of the user (and may actually exceed the total amount of real memory available on the entire CP-6 system), only a portion of each virtual data segment actually exists in real memory at any particular time. The CP-6 hardware and monitor conceal this fact from the user's program, by bringing selected portions of the virtual segments into "real" existence whenever they are needed by the program.

- Virtual data segments can "survive" after the program which created them has ceased to exist. Each virtual segment is associated with a CP-6 keyed file, stored on disk; this file is identified by the user's program when the virtual segment is initialized, and may be cataloged, stored, backed up, and deleted in the same way that any CP-6 file is manipulated. If a virtual data segment's file is closed and cataloged in a file directory, another program may subsequently re-open the same file and access its contents as a virtual segment; the data stored in the original segment is available, intact, in the new segment.

- Special addressing techniques are necessary to access the contents of the virtual segment, if its total size exceeds 256K words.

How Virtual Segments Work

A virtual data segment is actually a way of addressing data which exists in a CP-6 keyed disk file. Virtual data segments are initially created by use of the M$OPEN service and are released by use of the M$CLOSE service; they are subject to all normal CP-6 file creation and access rules. To create a virtual segment (and its associated disk file), the user's program must issue an M$OPEN which includes a special VIRTUAL option (specifying a particular VLP_VIRTUAL data structure). The program must supply the following information:

1. The identity of a DCB which is to be used to store this virtual segment. The DCB must be assigned to an ASN=FILE, ORG=KEYED file (with a DISPosition of either NAMED or SCRATCH). The DCB may be opened with FUN=CREATE,EXIST=NEWFILE to create a new segment, or with FUN=UPDATE to access an existing virtual-segment file.

2. The virtual size of the segment (i.e., the amount of space that the user wishes to be able to access). This value is specified as a decimal number of words, and is passed through the SEGSIZE option of the VLP_VIRTUAL structure.

3. The physical size of the segment (i.e., the amount of real memory which is to be used to retain portions of the virtual data). This value is specified as a decimal number of pages (not words), and is passed through the PHYSICAL option of the VLP_VIRTUAL structure.

The CP-6 monitor opens the indicated DCB as specified. The user may then examine the field VLP_VIRTUAL.PTR$, which contains a pointer to the base (beginning) of the newly-created virtual segment in memory.

The user's program may now use the virtual data segment as it would use any other CP-6 data segment (subject to the addressing limitations discussed below). The program may store or access data at any location within the segment's virtual address limits. If the program attempts to access a portion of the virtual segment which does not currently reside in real memory, the following sequence of events occur:

1. The CP-6 hardware's memory-management logic generates a "missing page" fault.

2. The CP-6 monitor's trap handler determines that the fault occurred while accessing a virtual data segment, and calls the virtual-segment manager.

3. The virtual-segment manager looks through the real pages currently assigned to the virtual segment, trying to find one or more pages which have not been accessed recently. The pages which have not been accessed for the greatest length of time are selected for purging.

4. The virtual-segment manager purges the oldest little-used page, by writing it into the keyed file associated with this segment if the page has been modified recently. (If the page has not been updated since it was last M$READ from the keyed file, it is not re-written).

5. The virtual-segment manager issues an M$READ, to bring the virtual page needed by the program into the physical page just purged. The user's memory map is updated to reflect the change in real memory allocation.

6. The CP-6 monitor returns control to the user's program. The instruction which triggered the "missing page" fault is re-executed, and normally runs to completion.

   Note: The virtual-segment manager brings pages into real memory strictly on an "as-needed" basis, and only moves one page at a time. Thus, if an instruction accesses several pages in a virtual data segment, it may generate more than one "missing page" fault; the virtual-segment manager is called once for each fault, and brings in one page each time.

A special case of this sequence occurs during the initial use of a virtual data segment, when few virtual pages have ever been accessed. If a "missing page" fault occurs, and the virtual data segment does not yet have as many real pages allocated as are permitted, then the virtual-segment handler simply allocates a new real page to hold the as-yet-unused virtual page, and it returns control to the user's program without performing any disk I/O operations. Thus, if the user's program actually allocates enough real pages to hold that portion of a virtual segment that it really accesses, then little or no disk I/O is necessary.

A virtual data segment (and its associated disk file) is released by use of the standard monitor service M$CLOSE. The program may opt to retain the segment file, or to discard it, as follows:

- If the M$OPEN which created the segment specified DISP=SCRATCH, or if the M$CLOSE which releases it specifies DISP=RELEASE, then the segment and its associated disk file are immediately discarded. The information stored in the segment is lost, and cannot be recovered.

- If the M$OPEN which created the segment specified DISP=NAMED, and if the M$CLOSE which releases it specifies DISP=SAVE, then the segment's disk file (and the information it contains) is retained. The real pages containing portions of the segment's data are written onto the disk file before being released.

Size Limits of Virtual Data Segments

There are two aspects to the size of a virtual data segment: virtual size and real size. The virtual size of a segment is the number of uniquely addressable memory locations within the segment; the real size of a segment is the amount of physical memory which is used to hold portions of the virtual segment during processing.

The upper limit of a segment's virtual size is limited by two factors:

1. The limit of the hardware's ability to represent addresses. The DPS-8 virtual-memory hardware is capable of accessing 4 billion (actually 4,294,967,296) unique word addresses; no virtual data segment may exceed this size.
   Note: References to hardware in the following discussion refer to the DPS-8, L66, or any other hardware on which the CP-6 operating system can run.

2. Special addressing techniques are necessary to access any address within a virtual segment which lies at any address above the 256K-word boundary. These techniques are discussed in more detail later.

The upper limit of a segment's real size is dictated to a large extent by the segment's virtual size, as follows:

1. If the virtual size of a segment is 2 mega-words (2048 pages) or less, then the segment's real size may vary from 6 pages to the current virtual size (or the remainder of the user's memory authorization, whichever is less). In this situation, it is possible to have a virtual segment which is completely memory resident; no "missing page" faults occur, and no disk I/O is necessary.

2. If the virtual size of a segment exceeds 2 mega-words (2048 pages), the upper limit of the segment's real size is 256 pages. This sharp reduction in the real size limit occurs because the virtual-segment manager must use a "fragmented" page table to keep track of the segment's real memory; such a table is limited by the hardware to 256 entries.

Addressing Data within a Virtual Segment

Data within a virtual segment is addressed in much the same way as data within any data segment is accessed: through PL-6 "PTR" variables, pointing to other PL-6 variables which have been declared "BASED".

Unfortunately, some fundamental limitations of the hardware make accessing large virtual segments rather more difficult than the previous statement would imply. The hardware is designed primarily to access data which lies within segments not exceeding 256K words in size. Pointer registers (and "PTR" variables in PL-6, of course) contain an 18-bit field which contains the "word displacement from the beginning of the segment"; index registers are only 18 bits wide; addressing calculations are performed in an 18-bit modulus arithmetic; and so forth.

There are three practical ways in which the programmer may work around these hardware limitations:

1. Never use a virtual segment that exceeds 256K words in size. If this size limit is honored, a virtual segment may be addressed in exactly the same fashion as any other data segment.

2. Generate one or more secondary descriptors, which permit access to portions of a virtual segment which lie above the 256K-word address boundary. In effect, this method breaks up one large virtual segment into a number (up to 16) of smaller segments, each of which is up to 256K words in size. Each of these smaller segments has a unique "segment ID", and may be addressed as a distinct area of memory.

3. Use the hardware's "extended addressing" instructions to directly access any location within the virtual segment.

Method 1 can be performed entirely in PL-6; methods 2 and 3 each require some programming in GMAP or BMAP (DPS-8 assembler) or FORTRAN, as they involve the use of some DPS-8 instructions which the PL-6 compiler never generates.

Method 1: Small Virtual Segments

This is certainly the easiest way to use virtual segments. Using this method, a program may access up to three virtual data segments of 256K words each, for a total virtual data area of 768K words. Each of the three virtual segments may be accessed in the same fashion as a normal (non-virtual) data segment. Data in the segments may be accessed through PL-6 "PTR" variables which point to suitable "BASED" variables, and the data may be passed to other PL-6 or FORTRAN procedures through the normal parameter-passing channels.

No assembly-language code is required when using this method.

Method 2: 'Divide and Conquer'

This method operates by breaking a single (large) virtual segment up into a set of smaller segments, each of which is 256K words or less in size. In the current version of the CP-6 system, it is possible to create up to 16 of these sub-segments, thus permitting a program to simultaneously access up to 4 mega-words of virtual memory. Each sub-segment has a unique "SEGID" (segment ID number), and is treated as a completely self-contained section of memory. It is not possible to "access across" the boundary between two of these sub-segments without causing a fault to occur; each BASED structure allocated by the user must lie completely within one of these segments.

To use this method, the user's PL-6 program must call a special-purpose routine written in BMAP assembler; this routine executes the instructions necessary to create and store NSA descriptors which frame up to 16 256K-word sub-segments, and return PL-6 "PTR" variables which may be used to refer to these sub-segments.

Figure 8-7 contains a sample PL-6 routine which creates a virtual segment and calls the BMAP routine called "SHRINK". Figure 8-8 contains the "SHRINK" routine itself.

```
SETUP_SEGMENTS: PROC (NSEGS, SEG_PTRS$) ALTRET;

%INCLUDE CP_6;

DCL NSEGS SBIN;        /* input; number of 256K-word subsegments */
DCL SEG_PTRS$ (0:15) PTR;        /* output; PTRs to subsegments */

DCL I SBIN;
DCL VIRTUAL_SEGMENT_DCB DCB;

%FPT_OPEN (FPTN=OPEN_VIRTUAL_SEGMENT,
           DCB=VIRTUAL_SEGMENT_DCB,
           ASN=FILE,    ORG=KEYED,     FUN=CREATE,    DISP=SCRATCH,
           VIRTUAL=VLP_VIRTUAL);

%VLP_VIRTUAL  (FPTN=VLP_VIRTUAL,
               SEGNUM=VS1); /* could use VS2 or VS3 instead... */

DCL SHRINK ENTRY (4);

   IF NSEGS < 1 OR NSEGS > 16 THEN ALTRETURN;  /* illegal call */

   VLP_VIRTUAL.SEGSIZE# = NSEGS * 256 * 1024;    /* 256KW each */
   VLP_VIRTUAL.PHYSICAL# = NSEGS * 6;   /* minimum recommended */

   CALL M$OPEN (OPEN_VIRTUAL_SEGMENT);

   DO I = 0 TO NSEGS - 1;

      CALL SHRINK (VLP_VIRTUAL.PTR$,            /* base of VDS */
       I * 256 * 4096,           /* byte offset from VDS base */
       256 * 4096,           /* size of sub-segment, in bytes */
       SEG_PTRS$(I));      /* pointer is returned here... */

      END;

   RETURN;

END SETUP_SEGMENTS;
```

Figure 8-7. PL-6 Routine to Set up Sub-segments

```
        TTL     SHRINK
*D*     NAME:   SHRINK
*,*     CALL:   CALL SHRINK (base$, offset, size, new$);
*,*     INPUT:  "base$" points to the base of a virtual data segment;
*,*             "offset" contains a byte offset into the segment;
*,*             "size" contains the byte size of the resulting "shrink"
*,*             operation.
*,*     OUTPUT: "new$" contains a pointer corresponding to a new entry on
*,*             the argument stack, which contains the shrunken
*,*             descriptor.
*,*     DESCRIPTION: This routine is used to perform a "normal shrink"
*,*             operation on a segment framed by an NSA super-descriptor.
*,*             The shrunken descriptor is saved on the hardware argument
*,*             stack, and a pointer corresponding to the new AS entry is
*,*             returned to the user.
        USE     SHRINK,1
        ENTDEF  SHRINK
        ENTREF  X66_AUTO_4
        ENTREF  X66_ARET
X0      EQU     0
X1      EQU     1
X2      EQU     2
X3      EQU     3
X4      EQU     4
X5      EQU     5
X6      EQU     6
X7      EQU     7
PR0     EQU     0
PR1     EQU     1
PR2     EQU     2
PR3     EQU     3
PR4     EQU     4
PR5     EQU     5
PR6     EQU     6
PR7     EQU     7
*
SHRINK  TSX0    X66_AUTO_4      Set up AUTO and get parameters
        ZERO    7,0
        LDP1    5,,PR2          Pointer to new size
        LDQ     0,,PR1          Get size
        QLS     16              Shift byte count into place
        SBLQ    =0200000,DL     Make "byte count" into "byte bound"
        ORQ     W1FLGS          Add all flags & "normal shrink" info
        STQ     SHRVEC          Save in "shrink" vector
        LDP0    3,,PR2          Pointer to "base"
        LDP1    4,,PR2          Pointer to "offset"
        LDP7    0,,PR0          Get "base"
        LDEA7   0,,PR1          Set location field in descriptor
        LDD6    SHRVEC          Shrink DR7 into DR6, self-id
        SDR6    0               Push shrunken descriptor onto AS
        LDP3    6,,PR2          Get pointer to result area
        STP6    0,,PR3          Save new pointer
        TSX2    X66_ARET        Return to user
*
*               Constant data for SHRINK routine
*
W1FLGS  OCT     000000177640
*
*               Temp data area
*
        USE     SHRINK_DATA,0
SHRVEC  OCT     0,000000001777
        END
```

Figure 8-8. BMAP Utility Sample Routine 'SHRINK'

The SHRINK routine pushed new descriptors onto the hardware "argument stack". Space on this stack is quite limited, and is used for a number of purposes (including passing data to the CP-6 monitor when a monitor service routine is called). Therefore, programmers using method 2 should obey the following rule:

> Never push more than 16 descriptors onto the argument stack. If you push too many descriptors onto the stack, your program may abort with an "Argument stack is full" fault at some later time; this fault causes the entire contents of the argument stack to be discarded.

Method 3: Direct Accessing

It is possible to use the LDEA ("Load Extended Address") instruction in a way which permits a program to directly access the entire 4-gigaword address space permitted by the hardware. This technique cannot be used directly by a PL-6 program, as the PL-6 compiler has no knowledge of the LDEA instruction or of data structures larger than 256K words. Therefore, it is necessary to use BMAP (or GMAP) to access large virtual data segments in this fashion.

The LDEA instruction operates by inserting a "byte offset" value into a descriptor register. This byte offset is not subject to the usual 256K-word segment limitation, and thus "adjusts" the descriptor register to access data which lies anywhere within a virtual data segment.

Figure 8-9 shows a simple BMAP instruction sequence which might be used to access a single data word within a NON-virtual data segment. Figure 8-10 shows the corresponding sequence which can be used to access a data word within a large virtual segment.

```
        LDA     WORD_OFFSET     * get word offset
        LDP1    SEGMENT$        * get pointer to segment
        LDQ     0,A,PR1         * load data from segment/offset loc
        STQ     RESULT          * store it away
```

Figure 8-9. Accessing Data within a Standard Segment

```
        LDA     WORD_OFFSET     * get word offset
        ALS     2               * shift left 2 bits to convert
        STA     TEMP            *  to byte offset, then save it
        LDP1    SEGMENT$        * get ptr to base of VDS
        LDEA1   TEMP            * insert byte offset into DR1
        LDQ     0,,PR1          * load data from extended address
        STQ     RESULT          * save it
```

Figure 8-10. Accessing Data within a Large Virtual Segment

Further details concerning use of the LDEA instruction may be found in the "DPS-8 Assembly Instructions" reference manual (order number DH03).

Performance Considerations

The "cost" of using a virtual data segment depends on a number of factors:

- The virtual size of the segment.
- The number of real pages which are allocated to retain portions of the virtual segment.
- The locations within the segment of the data actually accessed by the program.

A virtual data segment is most efficient when most (or all) of those portions of the segment which are actually being accessed can be retained in real memory. In this situation, few or no "missing page" faults occur, and little or no disk I/O occurs; the program using the segment can run at "full speed". If the number of pages being frequently accessed by the program exceeds the number of real pages assigned to the segment, a condition known as "thrashing" may occur, and performance deteriorates drastically. One might consider "thrashing" to be the point at which the system is spending more time swapping virtual-memory pages than it is doing useful work.

At any particular time during its execution, a program usually tends to concentrate its efforts on a subset of the data available to it. This subset is generally called the program's "working set"; the working set changes with time, and may expand or contract substantially during different phases of a program's execution.

Guidelines for Virtual/Real Segment Sizing

Experience has shown that if the size of the working set is greater than three times the real memory size available, then the system will suffer from an unacceptably high rate of missing-page faults and thrashing. There are two ways to prevent a virtual-memory system from thrashing:

- Increase the amount of real memory used to hold the virtual segment.
- When allocating space within the virtual segment, cluster related data together. This tends to reduce the size of the working set.

Accounting Considerations

Use of a virtual data segment results in resource-use charges in a number of categories:

1. Real memory usage (in "page-minute" units). This charge depends on the number of pages of real memory actually used to hold the virtual segment (derived from VLP_VIRTUAL.PHYSICAL#) and on the amount of CPU time used by the program.

2. User service time. When a page fault occurs, the virtual segment handler generally issues a single M$WRITE followed by an M$READ. The time necessary to perform these services is combined as the "user service" category.

   On a DPS-C central processor (L66 high-speed model used as the "1.0 performance multiplier"), the M$WRITE/M$READ sequence generally requires between 8 and 10 milliseconds of CPU time (plus whatever "I/O wait" time is required by the disk drives; this time is not charged to the user).

3. PMME (monitor service) and disk I/O. Generally, each page fault causes the virtual segment handler to issue 2 PMMEs, and between 4 and 8 disk I/O operations.

The actual charges which a particular program incurs depend on the rates set by the CP-6 system manager. If, at an installation, memory usage is cheap and disk access is dear, the system manager can minimize charges by increasing the amount of real memory assigned to the program's virtual segments. On the other hand, if a site is memory-conscious and charges heavily for memory usage, and charges little or nothing for disk activity, then reducing the amount of real memory assigned to the virtual segments will probably cut costs (although the program will probably require more wall-clock time to execute).

Restrictions and Programming Considerations

The CP-6 monitor expects all FPTs, VLPs, buffers, etc. to be available in real memory whenever a monitor service request is called; it does not recover from a "missing page" fault when accessing any of these items. Therefore, it is NOT POSSIBLE to reliably pass portions of a virtual data segment to the CP-6 monitor, for use as I/O buffers or service parameters. To read data into a portion of a virtual data segment, the program should read the data into a temporary buffer in STATIC or AUTO memory, and then move it to the appropriate location(s) in the virtual segment; a similar technique should be used for writing data in a virtual segment. Do NOT attempt to place FPT or VLP structures in a virtual segment.

If access method 3 (LDEA Extended Addressing) is being used to address data in a large virtual segment, it is impossible to take the ADDR of data in the segment, or to pass portions of the segment to PL-6 (or any other) subroutines through the standard parameter-passing techniques.

Certain Extended Instruction Set (EIS) operations are sensitive to being interrupted and restarted (as may happen if a missing-page fault occurs). In particular, EIS instructions with a result operand that overlaps any source operand may not operate properly under these conditions. Bit- and byte-string manipulation instructions (MLR, MRL, CSL, and CSR) are particularly vulnerable to errors in situations of this sort.

As mentioned above, the performance of a virtual data segment depends very strongly on the amount of real memory assigned to the segment, and on the amount of virtual memory actually used by the program. Thus, a program which uses a virtual segment should be capable of determining the amount of real memory actually available, and should use a reasonable portion of that memory for the virtual segment. This may be done by making proper use of the M$GDDL service, which determines the number of pages of real memory which may be acquired by the program. The user may then "tune" the program's behavior by changing the value of the MEMORY option on the !RESOURCE, !ORES, or !LIMIT command.

For example:

```
%INCLUDE CP_6;

%FPT_GDDL (RESULTS=VLP_GDDL);
%VLP_GDDL;

%FPT_OPEN (FPTN=OPEN_VIRTUAL, DCB=VIRTUAL_DCB, VIRTUAL=VLP_VIRTUAL);

   CALL M$GDDL (FPT_GDDL);

   VLP_VIRTUAL.PHYSICAL# = VLP_GDDL.AVAIL_PGS# / 2;

   CALL M$OPEN (OPEN_VIRTUAL);
```

This program fragment assigns 50% of the remaining memory space to the virtual data segment.

Section 9

Techniques: Communications

This section contains descriptions of interfaces to the CP-6 communications software. This discussion is not exhaustive, but is intended to provide useful examples.

Terminal I/O Control

Although the CP-6 FEP provides a rich variety of automatic formatting features, which enable the FEP to optimize throughput to devices that have positioning and optimizing hardware built in, sometimes these features are not desired. One example of a situation where the automatic FEP line wrapping and spacing are not desired might be in an application which wishes to take full advantage of the incremental positioning of a print head on a specialized device designed for text printing. By using the TERMINAL organization on a DCB, coupled with the TRANS=YES option on M$WRITE, the application will be able to send ESCAPE sequences, carriage return/linefeed sequences, and tab characters necessary to get the desired performance from the device, and bypass the CP-6 FEP automatic formatting features.

In Figure 9-1, the ORG=TERMINAL option is used on the device open to guarantee passage of certain characters directly to the device, such as ESCAPE and other characters from the first few rows of the ASCII chart. In addition, TRANS=YES (transparency) is used to indicate that the application program also handles carriage control and intends to bypass the automatic line-wrapping at !PLATEN width. Note that the BIN=YES (binary data) option is not used. This application is passing one byte of ASCII information per host (9-bit) memory byte. With ORG=TERMINAL and TRANS=YES, the ninth bit is automatically stripped when passing the byte from the host to the FEP. The binary option, when used, would guarantee passage of all the bits in the I/O buffer to the FEP, including the ninth bit. In this case, since the data is arranged one byte per byte, ORG=TERMINAL on the M$OPEN and TRANS=YES on the M$WRITE is sufficient.

```
      %FPT_OPEN       (FPTN=OPEN_DSI_TERM,
                           ASN=DEVICE,
                           ACS=SEQUEN,
                           DCB=F$DSI,
                           DISP=NAMED,
                           FUN=CREATE,
                           ORG=TERMINAL);

      %FPT_WRITE      (FPTN=WRITE_BUFOUT,
                           BUF=BUFOUT,
                           BP=YES,
                           DCB=F$DSI,
                           TRANS=YES,
                           WAIT=YES);

/*M* DSIDUMP - contains machine-specific plotting subroutines */
/*X* DMR,PLM=5,IND=5,CTI=5,SDI=5,MCL=10,CSI=0,ECI=0,DTI=2 */
%SET LISTSUB='1'B;
                                        /**/
DSIDUMP: PROC (CHARS_, COUNT_, FIN);
                                    /*
                                         CHARS_ IS CHARACTER ARRAY
                                         COUNT_ IS CHARACTER COUNT
                                         FIN < 0 => FORCE DUMP TO
                                              TERMINAL
                                                                 */
                              /*****************************************
                                   ARGUMENTS
                               *****************************************/
DCL CHARS_ (0:0) CHAR(1) CALIGNED;
DCL COUNT_ SBIN WORD;
DCL FIN SBIN WORD;


                              /*
                                   LOCALLY NEEDED %SUBS
                                                                 */
%SUB TRUE#='1'B /*TRUE#*/;
%SUB FALSE#='0'B /*FALSE#*/;

                              /*
                                   INCLUDES
                                                                 */
%INCLUDE CP_6;
%INCLUDE CP_6_SUBS;
     %F$DCB;


                                    /*
                                         EXTERNALS
                                                                 */
DCL F$DSI DCB;
DCL F$DSIN DCB;

DCL 1 STATS SYMREF,
          2 NUMCHARS SBIN WORD,
          2 PRTCHARS SBIN WORD,
          2 PLTREADS SBIN WORD,
          2 WIPEMEMS SBIN WORD,
          2 PUSHBAKS SBIN WORD,
          2 FILEREAD SBIN WORD,
          2 FILEWRTE SBIN WORD,
          2 TERMWRTE SBIN WORD;
```

Figure 9-1. PL-6 Subroutine to Control Terminal I/O (cont. next page)

```
                                    /*
                                         LOCAL STORAGE
                                                                    */
DCL F$DSI$ PTR STATIC;

DCL CHAR_COUNT SBIN WORD STATIC ALIGNED INIT(0);
DCL CHAR_MAX  SBIN WORD STATIC ALIGNED INIT(72);
DCL ETX_OUT_ SBIN WORD STATIC ALIGNED INIT(0);
DCL ETX_ERRORS SBIN WORD STATIC SYMDEF ALIGNED INIT(0);

DCL BUFOUT CHAR(255) STATIC;
DCL BUFOUTU (0:254) REDEF BUFOUT CHAR(1) CALIGNED;

DCL HERE1CE BIT(1) ALIGNED STATIC INIT(FALSE#);
DCL FLOW BIT(1) STATIC ALIGNED INIT(FALSE#);

DCL ESC CHAR(1) STATIC CALIGNED INIT(BITASC('033'0));
DCL ETX CHAR(1) STATIC CALIGNED INIT(BITASC('003'0));
DCL PROMPT_ CHAR(1) STATIC INIT('@');
DCL READ_BUF_ CHAR(1) STATIC INIT(' ');

DCL I SBIN WORD STATIC;
DCL RECS_ UBIN WORD STATIC SYMDEF ALIGNED INIT(0);

DCL BAUD_RATES(0:15) STATIC SBIN HALF INIT (
                         -1,             /* 50 */
                         -1,             /* 75 */
                          110,
                         -1,             /* 134 */
                         150,
                         200,
                         300,
                         600,
                          -1,            /* 1050 */
                         1200,
                         -1,             /* 1800 */
                         -1,             /* 2000 */
                         -1,             /* 2400 */
                         -1,             /* 4800 */
                         -1,             /* 9600 */
                         -1    /* 19200 */);

DCL LINESPEED SBIN WORD STATIC SYMDEF INIT(-1);


%EJECT;
                                    /*
                                         FPTS
                                                                    */
     %FPT_EOM        (FPTN=SET_EOM,
                          DCB=M$UC,
                          EOMTABLE=DOO_DAH);

          %VLP_EOMTABLE(FPTN=DOO_DAH,
                          VALUES="10, 44, 0, 47, 0*12");

     %FPT_PROMPT     (FPTN=PROMPT_FLOW,
                          TRANS=YES,
                          DCB=M$UC,
                          PROMPT=PROMPT_,
                          VFC=YES);
```

Figure 9-1. PL-6 Subroutine to Control Terminal I/O (cont. next page)

```
      %FPT_OPEN       (FPTN=OPEN_FLOW,
                           DCB=F$DSIN,
                           ASN=DEVICE,
                           FUN=IN,
                           ORG=TERMINAL,
                           RES='ME');

      %FPT_CLOSE      (FPTN=CLOSE_FLOW,
                           DCB=F$DSIN);

      %FPT_READ       (FPTN=READ_FLOW,
                           DCB=F$DSIN,
                           BUF=READ_BUF_,
                           WAIT=YES);

      %FPT_OPEN       (FPTN=OPEN_DSI_TERM,
                           ASN=DEVICE,
                           ACS=SEQUEN,
                           DCB=F$DSI,
                           DISP=NAMED,
                           FUN=CREATE,
                           ORG=TERMINAL);

      %FPT_OPEN       (FPTN=OPEN_DSI_FILE,
                           DCB=F$DSI,
                           ASN=FILE,
                           ACS=SEQUEN,
                           DISP=NAMED,
                           FUN=CREATE,
                           EXIST=NEWFILE,
                           CTG=YES,
                           ORG=CONSEC,
                           REASSIGN=YES);


                                                 /**/
      %FPT_WRITE      (FPTN=WRITE_BUFOUT,
                           BUF=BUFOUT,
                           BP=YES,
                           DCB=F$DSI,
                           TRANS=YES,
                           WAIT=YES);

      %FPT_GLINEATTR (FPTN=FETCH_SPEED,
                           LINEATTR=SPEED_TABLE);

           %VLP_LINEATTR (FPTN=SPEED_TABLE);


%EJECT;

/*
      Begin  D S I D U M P  main
                                                                    */

      IF NOT HERE1CE
      THEN
        DO;
             HERE1CE = TRUE#;
             F$DSI$ = DCBADDR(DCBNUM(F$DSI));
             IF F$DSI$ -> F$DCB.ASN# = %DEVICE#
             THEN
               DO;
                    CALL M$OPEN (OPEN_DSI_TERM) ALTRET (BLEW_IT);
                    CALL M$OPEN (OPEN_FLOW) ALTRET (BLEW_IT);
                    CALL M$PROMPT (PROMPT_FLOW) ALTRET (BLEW_IT);
```

Figure 9-1. PL-6 Subroutine to Control Terminal I/O (cont. next page)

```
                    CALL M$EOM (SET_EOM) ALTRET (BLEW_IT);
                    CALL M$GLINEATTR (FETCH_SPEED) ALTRET (BLEW_IT);
                    LINESPEED = BAUD_RATES(SPEED_TABLE.LINESPEED#);
                    IF LINESPEED = -1
                    THEN
                         GOTO BLEW_IT;
                    FLOW = TRUE#;
                                              /**/
                  END;
               ELSE
                    IF F$DSI$ -> F$DCB.ASN# = %FILE#
                    THEN
                      DO;
                           CALL M$OPEN (OPEN_DSI_FILE) ALTRET (BLEW_IT);
                           FLOW = FALSE#;
                        END;
                    ELSE
                         GOTO BLEW_IT;
            END;                              /* DO IF 1ST TIME */

      DO WHILE (FALSE#);                      /* ALTRET HANDLER */
BLEW_IT: ;
            CALL M$XXX;
          END;                                /* DO WHILE ALTRET */
                                              /**/
                                              /**/
                                              /**/
      IF FIN < 0
      THEN
        DO;
             CALL WRITE;
             RETURN;
           END;                               /* DO IF DUMP REQUESTED */
                                              /**/
      IF CHAR_COUNT + COUNT_ >= CHAR_MAX
      THEN
           CALL WRITE;

      DO I = 0 TO COUNT_ - 1;

           BUFOUTU(CHAR_COUNT) = CHARS_(I);
           CHAR_COUNT = CHAR_COUNT + 1;

         END;                                 /* DUMP EACH CHARACTER IN */


      IF CHAR_COUNT >= CHAR_MAX
      THEN
           CALL WRITE;
                                              /**/
      RETURN;                                 /* FROM THIS ROUTINE */
                                              /**/
WRITE: PROC;
                                              /* INTERNAL ROUTINE "WRITE" */
                                              /**/
      IF FLOW
      THEN
        DO;
             BUFOUTU(CHAR_COUNT) = ETX;
             CHAR_COUNT = CHAR_COUNT + 1;
             ETX_OUT_ = ETX_OUT_ + 1;
                                              /**/
             WRITE_BUFOUT.BUF_.BOUND = CHAR_COUNT - 1;
             CALL M$WRITE (WRITE_BUFOUT) ALTRET(WHO_CARES);
             STATS.TERMWRTE = STATS.TERMWRTE + 1;
WHO_CARES:   ;
             CHAR_COUNT = 0;
```

Figure 9-1. PL-6 Subroutine to Control Terminal I/O (cont. next page)

```
            RECS_ = RECS_ + 1;
                                          /**/
            IF ETX_OUT_ > 1
            THEN
              DO;
READ_ACK:           ;
                    CALL M$READ (READ_FLOW) ALTRET (READALT);
                    DO WHILE (FALSE#);
READALT:                 ;
                         ETX_ERRORS = ETX_ERRORS + 1;
                       END;                 /* DO WHILE ALTRET */

                    ETX_OUT_ = ETX_OUT_ - 1;
                  END;                      /* DO IF WAITING FOR ACK */
                                            /**/
                                            /**/
            IF FIN < 0
                 AND
                 ETX_OUT_ > 0
            THEN
                 GOTO READ_ACK;
          END;                              /* DO IF FLOW CONTROL */
     ELSE
       DO;                                  /* IF TO FILE */
            WRITE_BUFOUT.BUF_.BOUND = CHAR_COUNT - 1;
            CALL M$WRITE (WRITE_BUFOUT) ALTRET(ALTWRITE2);
            STATS.TERMWRTE = STATS.TERMWRTE + 1;
ALTWRITE2:  ;
            CHAR_COUNT = 0;
            RECS_ = RECS_ + 1;
                                          /**/
          END;                            /* DO IF WRITING TO FILE */

     RETURN;
END WRITE;
                                          /**/
END DSIDUMP;
```

Figure 9-1. PL-6 Subroutine to Control Terminal I/O

Transparent I/O for Asynchronous Graphics Terminals

Sending data in transparent or non-transparent mode to asynchronous graphics terminals (such as the Tektronics 40xx series) requires a thorough understanding of the specific terminal in use as well as the particular communications network in which the terminal is to operate.

Transparency and M$WRITE

The following parameters control transparent operation at write (and read) operations: fpt.DVBYTE.BIN#, fpt.DVBYTE.TRANS#, and dcb.ORG#. These parameters affect the write operation as follows:

1. If dcb.ORG# = %UR, the characters in the buffer are run through the "unit record" translation table, which converts all non-printable characters to blanks. If dcb.ORG# = %TERMINAL, the characters are not so translated.

2. If fpt_write.V.DVBYTE.BIN# is set, the data in the buffer is sent to the FEP in binary mode. All bits in the data are significant; each double-word in the buffer (72 bits) results in the transmission of nine 8-bit ASCII characters.

   If fpt_write.V.DVBYTE.BIN# is not set, the data is sent to the FEP in ASCII mode. Each 9-bit byte in the buffer has its high-order bit stripped off, and the remaining 8 bits prepared for transmission.

3. If fpt_write.V.DVBYTE.TRANS# is set, the 8-bit characters derived from step (2) above are transmitted exactly; no output optimization, tab expansion, or cursor positioning (including VFC and CR/LF) is done. If fpt_write.V.DVBYTE.TRANS# is not set, the data is processed through the output optimizing logic, has its parity set as appropriate, and is sent.

Transparency and M$READ

M$READ behaves almost as an mirror image of M$WRITE. If a transparent-mode read is issued, the data characters received on the line are stored in the user's buffer (in ASCII mode only, BINary input from ASYNC lines is not implemented); the CP-6 input editing functions (escape sequences, backspace, DEL, etc.) and echoing are disabled. When using a transparent read, the "read complete" condition can occur from one of several causes:

- Enough bytes have been received to fill the user's buffer completely;
- An activation character has been received and the program specified (via the M$STRMCTL service, with the flag VLP_TRMCTL.V.ACTONTRN# set) that transparent-mode reads are to honor the current activation character set (which may be specified via the M$EOM service);
- or, a read timeout condition (specified by M$EOM) has occurred.

Issuing a transparent-mode read puts the terminal into transparent-input mode; the terminal remains in this mode until a non-transparent read is issued.

Performing Transparent/Non-transparent I/O

To perform I/O with a Tektronics (or HP, or any other) graphics terminal which expects to send and receive transparent data, first, open a DCB to the ME# device, specifying FUN=UPDATE and ORG=TERMINAL. This ensures that unit-record translation is not performed.

To output to the terminal, issue one or more M$WRITE requests with the fpt_write.V.DVBYTE.TRANS# bit set. Whenever possible, it is preferable to buffer large amounts of information in the host and to issue big M$WRITEs (<= 2048 bytes) to the output DCB. While short (e.g., 10-byte) writes will work, the overhead involved in performing rapid burst of short writes can significantly degrade the performance of both the host program and the FEP being used.

To perform an operation such as "return current position of graphics crosshairs", use a sequence such as the following:

1. Write any output currently buffered (see step 2 under "Transparency and M$WRITE");

2. Issue an M$TRMPRG service request, specifying that any input currently in the FEP's buffers be discarded;

3. Issue an M$EOM service, setting the read timeout to 1 ten-millisecond interval (the minimum);

4. Issue a 1-byte transparent M$READ. Expect this read to ALTRET with a "read timed out" indication; if it returns normally, go back to step 2. This process will help to ensure that any input that the user may have typed ahead will be flushed and will not interfere with the terminal-enquiry operation, and will ensure that the terminal is placed in transparent-input mode before the response to step 6 is received (which might not happen on a heavily-loaded FEP if this step were omitted).

5. If the operation being performed is one in which the terminal is supposed to answer immediately (e.g., "report terminal status"), issue an M$EOM specifying a timeout value somewhat greater than the communication network's end-to-end turnaround delay plus the time required for the terminal to respond (e.g., approximately 5 seconds for a Tektronics connected to a local FEP). If the operation being requested requires the terminal user to take manual action (e.g., "position crosshairs and hit a key, please"), issue an M$EOM specifying a timeout value of 0 (to disable timeout) or of 1 minute (for example).

6. Issue a transparent M$WRITE, sending the character sequence to make the terminal perform the necessary "answerback".

7. Issue a transparent M$READ, specifying a buffer large enough for the terminal's most verbose response. This read may ALTRET with a "read timed out" if the terminal sends fewer bytes than the buffer specified; this condition must be handled in a fashion appropriate for the terminal in question.

8. Issue an M$EOM service specifying TIMEOUT=0 to disable the read timeout.

NOTE: Performing only steps 1 and 5-8 may be sufficient for some graphics terminals. However, taking steps 1-8 ensures correct operation.

Use of Comgroups

The program shown in Figure 9-2 illustrates the use of comgroups. To make the example more interesting, two comgroups are used. As in most comgroup applications where the data coming in from the comgroup arrives in bursts, it is desirable to use no-wait IO. This allows the program to perform other tasks when there is no activity on the comgroup.

This program opens two comgroups, sets up each one to allow terminal connects but not DCB connects, and then starts a no-wait read on each comgroup. Data received from any connected terminals is written through M$LO; any terminal sending "OFF" will be disconnected from its comgroup.

As with most comgroup applications (and with most CP-6 applications), there are several ways to perform any given task. The example below is not intended to be the best solution to the problem but serves to illustrate some basic techniques.

The most important thing to be learned from this example is to avoid the temptation to place too much processing in the event routine. Most first-time no-wait I/O programmers will process the event and re-issue the read in the event routine, which can lead to disaster.

When an event occurs, the current program status is placed in the user's TCB which is treated as a push down stack. A RETURN from the event handler pops the stack, and control returns to the point where the event took place. If another read is started while still in the event routine, before the stack is popped, another event can occur and another frame can be pushed. The number of frames the TCB can hold is a LINK option, but there is a finite limit. Eventually, if the comgroup has several records in it that the read will satisfy, and a read is re-issued before the stack is popped, the stack will overflow. Even using the DO INHIBIT feature will not prevent stack overflow, because any monitor service call creates an opportunity for event processing to occur.

Therefore, the suggested course is the one shown below: save the pertinent information from the stack frame, set a flag, and return. More complicated processes will usually utilize a linked list of events; a short cut was used here because of the limited nature of the example. Since the processing to be done on any given event is limited, the read is not re-issued until after all processing is complete; thus only one event need be saved per comgroup. An information table is used, indexed by comgroup number.

The event code for both the open and the read calls is used as the index to the information tables. Note that the event code is offset by one, i.e., table index 0 is event 1. Event codes of zero have a special meaning, see the Monitor Services Reference Manual for more information.

To get the most from this example, refer to the Monitor Services Reference Manual for the defaults taken for the various FPTs. While relatively few options are specified, correct running of this program depends on several defaults in VLP_CGCP.

This program also makes use of the "anonymous queue". Comgroup terminals always write into the anonymous queue; that is why the M$READ call need not specify a station. Reviewing the M$READ, M$OPEN/SETSTA, and VLP_SETSTA defaults is essential to understanding the example.

```
/*M* CGDEMO - Comgroup demo */
CGDEMO: PROC MAIN;

%EQU CG1_READ=1;
%EQU CG2_READ=2;
%EQU CG1_OPEN=1;
%EQU CG2_OPEN=2;

DCL I UBIN;

DCL CG1 DCB;
DCL CG2 DCB;
DCL M$LO DCB;

DCL 1 CGTABLE (0:1) STATIC SYMDEF,
     2 PROCESS BIT(1) ALIGNED,          /* set when read complete */
     2 ADMIN BIT(1) ALIGNED,            /* set by admin messages from
                                           the comgroup */
     2 DCB# UBIN,                       /* This comgroups DCB */
     2 EVENT UBIN,                      /* The comgroup admin message
                                           event, not the no-wait event,
                                           valid only when ADMIN is true */
     2 BUF_ VECTOR,                     /* this comgroups buffer */
     2 ARS_UBIN,                        /* size of current record */
     2 STATION CHAR(8);                 /* name of this records originating
                                           station */

DCL CG1_BUF CHAR(140) STATIC;
DCL CG2_BUF CHAR(140) STATIC;

DCL EVENT_ROUTINE ENTRY ASYNC;

%INCLUDE CP_6;

%EQU_CG;
%FPT_EVENT (UENTRY=EVENT_ROUTINE, STCLASS=CONSTANT);

%FPT_WAIT (FPTN=ZZZZ, STCLASS=CONSTANT,
           UNITS=86399);  /* 24 * 60 * 60 -1, WAIT is mod 24 hours */

%FPT_OPEN (FPTN=OPEN_CG1,      STCLASS=CONSTANT,
           ASN=COMGROUP,
           DCB=CG1,
           EVENT=%CG1_OPEN,
           NAME=CG1_NAME,
           SETSTA=MY_STATION,
           FUN=CREATE,
           EXIST=NEWFILE,
           CTG=YES,
           SHARE=ALL,
           AU=YES);

%FPT_OPEN (FPTN=OPEN_CG2,      STCLASS=CONSTANT,
           ASN=COMGROUP,
           DCB=CG2,
           EVENT=%CG2_OPEN,
           NAME=CG2_NAME,
           SETSTA=MY_STATION,
           FUN=CREATE,
           EXIST=NEWFILE,
           CTG=YES,
           SHARE=ALL,
           AU=YES,
           IXTNSIZE=30);
```

Figure 9-2. Sample Use of Comgroups (cont. next page)

```
%VLP_NAME (FPTN=CG1_NAME, STCLASS=CONSTANT,
          NAME='CG1');
%VLP_NAME (FPTN=CG2_NAME,  STCLASS=CONSTANT,
          NAME='CG2');

%FPT_CGCTL (CGCP=VLP_CGCP);

%VLP_CGCP (DCBCONLGL=NO,       STCLASS=CONSTANT,
          MAXMC=140);

%FPT_READ (FPTN=READCG,
          WAIT=NO);



%VLP_SETSTA (FPTN=MY_STATION,  STCLASS=CONSTANT,
          MYSTATION='ME');

%FPT_ACTIVATE (DISCONNECT=YES,
          STATION=DISC_STATION);

%VLP_STATION (FPTN=DISC_STATION);

/* Establish the event handler */
     CALL M$EVENT (FPT_EVENT);

/* Open the comgroups */
     CALL M$OPEN (OPEN_CG1) ALTRET(KEEP_IT_SIMPLE);
     CALL M$OPEN (OPEN_CG2) ALTRET(KEEP_IT_SIMPLE);

/* Set up comgroup parameters */

     FPT_CGCTL.V.DCB# = DCBNUM(CG1);
     CALL M$CGCTL (FPT_CGCTL) ALTRET(KEEP_IT_SIMPLE);
     FPT_CGCTL.V.DCB# = DCBNUM(CG2);
     CALL M$CGCTL (FPT_CGCTL) ALTRET(KEEP_IT_SIMPLE);

/* Setup the CG table */
     CGTABLE.BUF_(0) = VECTOR (CG1_BUF);
     CGTABLE.BUF_(1) = VECTOR (CG2_BUF);
     CGTABLE.DCB#(0) = DCBNUM(CG1);
     CGTABLE.DCB#(1) = DCBNUM(CG2);

/* Issue the first reads. Subsequent reads are done in PROCESS */
     DO I = 0 TO 1;
          READCG.V.DCB# = CGTABLE.DCB#(I);
          READCG.BUF_ = CGTABLE.BUF_(I);
          READCG.V.EVENT# = I+1;
          CGTABLE.PROCESS(I) = '0'B;
          CALL M$READ (READCG) ALTRET(KEEP_IT_SIMPLE);
     END;

/* Loop, checking for read completes using the PROCESS flag.  If
   there is nothing do, goto sleep.  The M$WAIT will be interrupted
   by an event, the RETURN from the event routine comes back to the
   instruction following the M$WAIT PMME. */

     DO WHILE ('1'B);     /* DO FOREVER */
          DO I = 0 TO 1;
             IF CGTABLE.PROCESS(I)
                 THEN CALL PROCESS(I);
             END;
/* Since a read could have completed in the interim, test flags
   again before sleep, since it will be a very deep sleep */
```

Figure 9-2. Sample Use of Comgroups (cont. next page)

```
       DO INHIBIT;
          IF NOT CGTABLE.PROCESS(0) AND NOT CGTABLE.PROCESS(1)
               THEN CALL M$WAIT (ZZZZ);  /* A wait is always interruptable */
         END; /* Do inhibit */
      END;  /* DO WHILE */

KEEP_IT_SIMPLE:
      CALL M$MERC;   /* let the monitor print the error */


/*
      This routine does the actual work, based on the contents of the
      active table entry.
*/
PROCESS:  PROC(I);

DCL I UBIN;

DCL 1 OUTBUF STATIC,
      2 * CHAR(0) INIT('From '),
      2 STATION CHAR(8),
      2 * CHAR(0) INIT(':'),
      2 * CHAR(0) INIT (' '),
      2 EVENT CHAR(20);

DCL CHAR3 CHAR(3) BASED;

%FPT_WRITE (FPTN=WRITEL0,
            DCB=M$L0);

      OUTBUF.STATION = CGTABLE.STATION(I);
      IF CGTABLE.ADMIN(I)
      THEN DO;
            OUTBUF.STATION = CGTABLE.STATION(I);
            DO CASE(CGTABLE.EVENT(I));
                 CASE(%CG_TCON#);
                      OUTBUF.EVENT='Connected';
                 CASE(%CG_TDSC#);
                      OUTBUF.EVENT='Disconnected';
                 CASE(%CG_TBRK#);
                      OUTBUF.EVENT='Break';
                 CASE(ELSE);
                      CALL BINCHAR(OUTBUF.EVENT, CGTABLE.EVENT(I));
                 END;  /* Do case */
            WRITEL0.BUF_ = VECTOR(OUTBUF);
            CALL M$WRITE (WRITEL0);
            END;  /* Admin event */
      ELSE DO;   /* not admin */
            OUTBUF.STATION = CGTABLE.STATION(I);
            WRITEL0.BUF_ = VECTOR(OUTBUF);
            WRITEL0.BUF_.BOUND = WRITEL0.BUF_.BOUND - SIZEC(OUTBUF.EVENT);
            CALL M$WRITE (WRITEL0);
            IF CGTABLE.ARS(I) = 0 THEN WRITEL0.BUF_ = VECTOR(NIL);
            ELSE DO;
                 WRITEL0.BUF_ = CGTABLE.BUF_(I);
                 WRITEL0.BUF_.BOUND = CGTABLE.ARS(I)-1;
                 END;
            CALL M$WRITE (WRITEL0);
            IF CGTABLE.ARS(I) = 3 AND VBASE(CGTABLE.BUF_(I))->CHAR3 = 'OFF'
            THEN DO;
                 DISC_STATION.STATION# = CGTABLE.STATION(I);
                 FPT_ACTIVATE.V.DCB# = CGTABLE.DCB#(I);
                 CALL M$DEACTIVATE (FPT_ACTIVATE) WHENALTRETURN DO;
                                                  END; /* ignore ALTRET */
                 END;
            END;  /* not admin */
```

Figure 9-2. Sample Use of Comgroups (cont. next page)

```
/*
     Reset the need-to-process flag and re-issue the read
*/
     CGTABLE.PROCESS(I) = '0'B;
     READCG.V.DCB# = CGTABLE.DCB#(I);
     READCG.BUF_ = CGTABLE.BUF_(I);
     READCG.V.EVENT# = I+1;
     CALL M$READ (READCG);
     RETURN;
END PROCESS;
END CGDEMO;
%EOD;
/*
     The event routine is entered when the previously
     issued comgroup read is complete.  The TCB must be big enough
     to handle concurrent events since an event from one comgroup
     can occur during processing of another.  In this case, only
     two events can occur at once, which will fit in the default
     TCB size.
*/
EVENT_ROUTINE: PROC ASYNC;

DCL STK$ PTR;
DCL I UBIN;

DCL 1 CGTABLE (0:1) SYMREF,
     2 PROCESS BIT(1) ALIGNED,
     2 ADMIN BIT(1) ALIGNED,
     2 DCB# UBIN,
     2 EVENT UBIN,
     2 BUF_ VECTOR,
     2 ARS UBIN,
     2 STATION CHAR(8);

%INCLUDE CP_6;

DCL B$TCB$ PTR SYMREF;
%B$TCB (STCLASS="BASED(B$TCB$)");
%B$NWIO (STCLASS="BASED(STK$)");
%B$CGAURD;

     STK$ = B$TCB.STK$;
     I = B$NWIO.EVID-1;
     IF B$NWIO.CGPARM.MSGTYP# = '*AUEV'
     THEN DO;
          CGTABLE.ADMIN(I) = '1'B;
          CGTABLE.EVENT(I) = VBASE(CGTABLE.BUF_(I))->B$CGAURD.EVENT;
          END;
     ELSE CGTABLE.ADMIN(I) = '0'B;
     CGTABLE.ARS(I)=B$NWIO.ARS;
     CGTABLE.STATION(I)=B$NWIO.CGPARM.STATION#;
     CGTABLE.PROCESS(I) = '1'B;
     RETURN;
END EVENT_ROUTINE;
```

Figure 9-2. Sample Use of Comgroups

To create devices which can be used as terminals for this program, use the session shown in Figure 9-3 as an example.

```
!SUPER

*** CP-6 SUPER B03 ***

CMD*CR DEV CGDEM02
OPT*USE=CG
OPT*COMGROUP=CG/CG2.account
OPT*SNAME=STATION2
OPT*END
  STATION2 created.
CMD*END

***  NO Errors  ***
***  NO Warnings ***


Substitute the account you will run CGDEM0 in for ".account".
```

Figure 9-3. Use of SUPER for Comgroup Definition

Section 10

Shared Run Units

Advantages of Shared Run Units

CP-6 provides a sophisticated capability for sharing the unchanging portions of frequently-used run units (that is, those portions of the program which cannot be altered by the program). This sharing leads to a dual advantage:

- Total system memory usage is greatly reduced. For example, no matter how many users are accessing the FORTRAN compiler, there is only one copy of the compiler's executable code in memory.

- System response is substantially improved. After a shared run unit has been loaded into memory once, its procedure does not need to be loaded again when other users invoke it. Thus, users invoking the processor receive a much faster response; the total number of disk I/O operations decreases sharply, increasing disk throughput for other users of the same disk packset.

Shared Programs

Programs can be divided into three categories, based on their ability to be shared:

1. Programs that must be shared to operate properly. All special shared processors (debuggers, command processors, shared libraries, and alternate shared libraries) fall into this category. Processors such as these are normally placed in "shared" status when CP-6 is initialized, and remain in that status.

2. Programs that cannot be shared. Programs in this class include programs with more than one level of overlay, programs which have been LINKed with the NSHAREABLE option, and programs stored in "star" files.

3. Programs that may be shared, but need not be. Most CP-6 processors and user-written programs fall into this category.

System Configuration to Permit Sharing

The sharing process operates, based on three options specified in the TIGR deck at system initialization time:

- The SPROC option is used to specify the name, type, and overlay count for a processor which is to be "always shared" (category 1). Generally, this option is used to list those command processors (CP), debuggers (DB), shared libraries (LI), and alternate shared libraries (AS) necessary for normal operation of the system. It is possible to use this option to specify that a standard processor (SP) should be "always shared", but this is not recommended.

- The SPSPACE option may be used to reserve space in the shared-processor tables for additional category-1 processors, which are to be installed at a later time by use of the SPIDER processor. This option may be used to permit installations to develop, install, and test new shared libraries, etc. without requiring that the system be rebooted each time the new libraries are changed.

- The SPAUTOSPACE option is used to reserve space in the shared-program tables for ordinary, shareable run units (category 3).

Auto-Sharing Process

The normal run-unit sharing process is performed automatically by the CP-6 monitor; it is essentially invisible to the users of a shared run unit, and requires no intervention by either the system operator or the system manager. Programs are placed in shared status when users invoke them, and remain in this status until there is no further use for them.

Programming Considerations

Almost all programs written for the CP-6 system are capable of being shared. Shared programs can acquire and release memory and DCBs, access files and issue all but a very few monitor service requests without having to take the program's "shared" status into consideration. A shared program can (if necessary) issue an M$UNSHARE monitor service request, which creates a new, unshared copy of the program's procedure on behalf of the user currently running the program.

Programs written in a high level language (FORTRAN, PL-6, PASCAL, etc.) are generally very good candidates for sharing, as all executable instructions and constants generated by these languages' compilers are automatically placed in "read/execute/no-write" memory.

Programs or subroutines which are being written in assembler (GMAP or BMAP) should be coded with data stored separately from procedure and without self-modifying procedure in order to take advantage of the auto-sharing process.

To ensure that a large, overlaid program can be shared in an effective manner, care should be taken when the program's overlay structure is designed. Specifically:

- The program may have only one level of overlays; that is, an overlay cannot have other overlays subordinate to itself.

- The program should not have a large number of overlays at level 1. Each overlay in the program requires one entry in the shared-program table; a program with (for example) 200 overlays might not fit into the table at all, or might fit only if most other shared programs were purged from the table.

  If it is necessary to run programs with large numbers of overlays, the programmer should request that the system manager increase the SPAUTOSPACE entry in the TIGR deck to accommodate the large programs. It is also important to note that not all overlays of an overlaid program which is shared need to be in memory at once.

Usage Considerations

Under normal circumstances, a CP-6 user accessing a shared run unit need not be concerned about the program's shared status. Differences in behavior appear only when a shared program is being debugged via the DELTA debugger (or some other, user-written debug tool).

The major difference between debugging a shared program and debugging an unshared program concerns changes to the program's "procedure" memory pages. These pages may be accessed by the program, but can never be modified by the program itself. The DELTA debugger is permitted to modify data in the procedure pages of an unshared program, but cannot modify information in the procedure pages of a shared program (because any such modifications would affect every user associated with the program!). This restriction has several implications:

- The user cannot use DELTA's MODIFY command to change instructions or constant data stored in the procedure portion of a shared program.

- The user cannot set procedure breakpoints (via DELTA's AT) in a shared program's procedure.

- The user cannot set data breakpoints (via DELTA's WHEN command) when executing a shared program, or when executing an unshared program which is associated with a shared library.

- The user cannot step through a shared program (via DELTA's STEP, STEP ONE CALL, or] commands).

The user may work around this restriction in one of several ways:

- The user may choose to invoke the program via the IBEX command "!START rununit UNDER DELTA" (or "!U" followed by "!rununit"). When a run unit is invoked in this fashion, the auto-sharing process is bypassed; the user receives an unshared copy of the program, and may proceed with normal debugging operations.

- If the user associates DELTA after the program has begun executing (via a control-Y "!!DELTA" sequence) or after the program aborts, DELTA informs the user that a "shared program associated - use UNSHARE to set breakpoints or modify procedure". At this point, the user must issue the DELTA command "UNSHARE" before setting instruction breakpoints or modifying the program's procedure; the user must issue the command "UNSHARE ALL" before setting data breakpoints.

- Some sophisticated programs make use of the M$ALIB monitor service to associate and pass commands to DELTA, to display data or modify the programs' procedure pages or perform other interesting tasks. If programs modify constant data or procedure, it is necessary for the programs to to issue an M$UNSHARE request before issuing the initial M$ALIB, to remove themselves from "shared" status and thus permit DELTA to modify their procedure.

Section 11

Special Shared Processors

A special shared processor is a run unit that may co-exist and interact with the user program. It differs from shared run-time libraries in that it resides in a working space other than that of the user; it is not LINKed with the user program.

The CP-6 system recognizes three types of special shared processors:

- Alternate Shared Libraries
- Debuggers
- Command processors.

Only one processor of a given type may be associated with a user at any time. The standard CP-6 system includes I-D-S/II as the supplied alternate shared library, DELTA as the debugger, and IBEX as the command processor. Installations may provide additional special shared processors for their own use in any of the three categories.

All special shared processors of a given type reside in a dedicated working space quarter concurrently with one another. Working space 6 is dedicated to Alternate Shared Libraries, working space 5 is dedicated to debuggers, and working space 4 is dedicated to command processors. The page table for a special shared processor is also contained within its working space.

A special shared processor run unit must consist of only pure procedure; there can be no DCBs or static data. Special shared processors may use the M$GETDCB monitor service to acquire DCBs in the user's Read-only Segment and use dynamic data segments for all non-constant data. A data segment area is provided in each user's own working space for each type special shared processor. One type of special shared processor does not have access to the data segments of another special shared processor, nor does the user have access to the data segments of any of the special shared processors. Refer to the map of user virtual memory in the Monitor Services Reference Manual, Appendix E.

It is, then, the procedure portion of a special shared processor that resides in its own working space. The procedure portion is shared by all users associated with the processor.

Entry to a special shared processor causes a change of domain, thereby changing the areas of memory to which the processor has access. The areas of user memory to which the processor has access are determined by its type and are described in the discussion of each type of special shared processors.

Guidelines for All Special Shared Processors

The following discussion presents information needed to create a special shared processor. Information that is common to the three types of special shared processors is included in this subsection; information that is specific to each type of special shared processor is presented later in this document.

Special Shared Processor Initialization

Because special shared processors do not contain static data or DCBs, initialization for all three types of special shared processors is similar. The following subsections discuss interfaces and techniques available for special shared processor initialization.

Processor Initialization Area (PIA)

Sixteen words of static in the user working space are reserved for use by special shared processor initialization routines. This area is shared by all special shared processors associated with the user program, so any process that references that area should be inhibited to avoid being interrupted by another special shared processor that may also modify the contents of the PIA. This space is framed by a descriptor in the special shared processor's Linkage Segment. A pointer to this area, BPIA, is SYMDEFed in the module B_USRPTRS_D.:LIBRARY. If the special shared processor is not to be LINKed with this module, a pointer to this area may be built as follows:

```
%INCLUDE B_SEGIDS_C.:LIBRARY;
DCL PIA BIT(36) CONSTANT INIT(%PIASID);
DCL PIA$ REDEF PIA PTR;
```

Also, this segment may be referenced in BMAP modules as follows:

```
SEGREF   PIASID
LDPn     PIASID,DL
```

Initial Entry and Obtaining AUTO Storage

When a debugger or alternate shared library is entered, it must first determine if this is the first time it has been called so that it may perform any initialization functions that may be required. The module that contains the special shared processor entry address must be written in BMAP, because on the first entry the AUTO segment must be obtained and initialized and on subsequent entries the AUTO stack must not be obtained, but must be re-initialized to its initial (empty) values.
NOTE: Command Processors do not have this requirement as the data segments for a command processor are released on every M$CPEXIT service request. A command processor's entry module may be written in PL-6 by declaring that procedure as the "MAIN" procedure. A call to X66_MAUTO will be generated by the compiler.

A simple test to determine if this is an initial entry to a special shared processor is to determine if an AUTO segment has been allocated. This may be accomplished as follows:

```
       SEGREF  AUTOSID
       SEGREF  PIASID
*
       LDI     =04010          Set HEX & OVRFL mask
       LDP2    AUTOSID
       LDP1    PIASID
       INHIB   ON
       STD2    0,,1            Store the AUTO descriptor in the PIA
       LXL1    0,,1            Load the descriptor access flags
       INHIB   OFF
       ANX1    =0100600,DU     Test Read, Not empty, Present
       ERX1    =0100600,DU
       TZE     NOTFIRST        Segment previously allocated, not first
```

AUTO storage for a special shared processor must be in a data segment. A special shared processor may establish the AUTO stack by making a call to the appropriate routine in X6U$CSEQU which is an object unit in the :LIBRARY account. If the need is only to initialize AUTO for possible future use, X66_MSTATIC should be called. If, however, the calling BMAP module needs AUTO storage, X66_MAUTO should be invoked. In either case the call is: TSX0 X66_Mxxx. The following word is assumed to be a data word. For X66_MAUTO it must contain the size of the desired frame in the left half of the word following TSX0, and must specify an even number of words:

```
       ENTREF  X66_Mxxx
*
       TSX0    X66_Mxxx
       ZERO    frame size,0
```

In either case return is: TRA 1,X0.

Now that the AUTO Stack has been allocated, the BMAP entry module may call PL-6 subroutines to complete any initialization functions.

The AUTO data segment may be re-initialized on subsequent entries to the special shared processor as follows:

```
NOTFIRST
       LDP2    AUTOSID,DL      Point to base of AUTO
       LDX0    -5,DU
       SXL0    0,,2            Set current frame offset
       AWD     4,,2            Point to first frame
```

A special shared processor must obtain its own DCBs. See the description of M$GETDCB in the CP-6 Monitor Services Reference Manual (CE33) for details. Some entries in the DCB table are reserved for special shared processors. These are mentioned in the discussion of each type of special shared processor.

Use of Data Segments

Because a special shared processor has no static data, it makes use of Data Segments in the user working space obtained via either the M$GDS monitor service described in the CP-6 Monitor Services Reference Manual or the use of the AREADEF attribute of the PL-6 compiler as described in the CP-6 PL-6 Reference Manual. The data segments for each type of special shared processor while residing within the user working space are separate from the user data segments and are not directly accessible by the user. The M$GDS automatically allocates the appropriate segments based on which domain issued the M$GDS request.

Once obtained, data segments remain allocated until the current user program is run down, with the exception of data segments for command processors which are released on every M$CPEXIT service request.

Exceptional Condition Processing

Exceptional condition processing works basically the same way for special shared processors as it does for user programs. Any differences are discussed in the description of exceptional conditions in Section 6 of the CP-6 Monitor Services Reference Manual. An author of a special shared processor should take note of:

- M$INT - "Entry to M$INT Procedure"
- M$EVENT - "Event Conditions and Domains"
- M$XCON - "Exit Control and Domains"

Also of interest are monitor services for exiting exceptional condition procedures that are available only to special shared processors:

- M$INTRTN - Return from M$INT procedure
- M$XCONRTN - Return from M$XCON procedure

Taking Snapshot Dumps

Taking and analyzing snapshot dumps is discussed in the following subsections.

Calling M$SCREECH

A special shared processor may cause an entry to CP-6 recovery for the purpose of creating a snapshot dump by using the M$SCREECH monitor service. One of the parameters of this service is the recovery code which allows the special shared processor to indicate what portions of the user memory and the special shared processor memory are to be written to the dump file. Refer to the following in Section 8 of the Monitor Services Reference Manual:

- M$SCREECH - RECOVERY
- VLP_SCODE

Special Shared Processor Data in Dump Files

The ANLZ processor may be used to examine data belonging to a Special Shared Processor. Complete information on ANLZ is contained in the CP-6 System Support Reference Manual (CE41). The following ANLZ commands may be useful:

- To examine the Special Shared Processor's Task Control Block:

```
TCB(ssp) nnn
```

- To examine the Special Shared Processor's Data Segments:

```
DU $LS4->0 USING nnn, ssp
DU $LS5->0 USING nnn, ssp
DU $LS6->0 USING nnn, ssp
      .
      .
DU $LS11->0 USING nnn, ssp
```

  where nnn is CUN for the current user, or is a specific user number and ssp is ICP, ASL, or IDB to identify the type of special shared processor.

Debugging of Special Shared Processors with XDELTA

This subsection discusses XDELTA and describes its use for the debugging of Special Shared Processors (debuggers, command processors, and alternate shared libraries). Specified here is also some general information pertaining to the use of XDELTA for debugging of user programs and the CP-6 Monitor.

XDELTA is a standalone entity which does not depend on the CP-6 Monitor for control or services. XDELTA uses the mini-I/O system in AARDVARK for all its input and output requirements. This is the same I/O system used at boot-time and by recovery. With this system XDELTA can read commands (usually patches) from cards, tape, or the console at boot-time, and can output to the console or the boot-time printer. The control of XDELTA at boot-time for patching is described in the Operations Reference Manual (CE34) under System Startup and Recovery. The other function of XDELTA is its use as an interactive debugger. XDELTA may be used to debug the Monitor, any user program, or any special shared processor (command processor, debugger, or alternate shared library). When being used as a debugger, XDELTA uses the mini-I/O system for reading commands (from the console or card reader) and for outputting to the console and the printer. XDELTA reads and writes only through the IOM-connected system console, and will print only on the IOM-connected printer specified at boot time.

XDELTA is the same as user DELTA in its general operation and command repertoire. The major difference is noticed when debugging entities other than the CP-6 Monitor when none of the program symbols are available. When debugging the Monitor, the linker defined symbols (ENTDEFs and SYMDEFs) and a small subset of debug schema are always available along with as much additional schema as was specified at boot time. (See Operations Reference Manual, CE34, System Start-up and Recovery.) For the same reason, the @ and # symbols are not defined except in the Monitor. To reference the patch space defined by LINK's IPATCH and DPATCH options, one must DEFINE symbols for the locations B_PATCHI and B_PATCHD respectively. These symbols can be found in the LINK map. Other general restrictions are that XDELTA has no provisions for handling overlaid programs and does not have a data breakpoint (WHEN) capability.

Using XDELTA

When using XDELTA to debug any domain other than the Monitor, it is important to remember that XDELTA references ASL, ICP, and IDB domains through some user's linkage segment and page table. The USE command discussed in detail later establishes which domain is to be examined. The most frequently used form of the USE command is as follows:

```
USE USER[#nn], domain
```

All users of the same ICP will cause XDELTA to reference the same actual pages of memory when referencing the ICP's instruction segment ($LSO), but all other references will be unique to each user. For this reason, it is usually best to debug a command program, ASL, or debugger which has only one user associated with it. In general it is best to debug these domains on a quiescent system if at all possible. All special shared processors are always shared; therefore, if a breakpoint is planted in a special shared processor via the address domain of one user, any other user of the same special shared processor can trigger the same breakpoint.

Debugging IDBs (Debuggers)

The following technique is useful for initially planting a breakpoint in a debugger:

1. Enter XDELTA via the DELTA keyin.

2. Place a Monitor breakpoint at SSU$DELTAGO.

3. GO

4. Start up the user which associates the debugger to be tested.

5. When the breakpoint is reached, the debugger is about to be entered. The USE command UU,IDB will give access to the debugger for the purpose of planting breakpoints. Using the LINK map for the debugger, DEFINE appropriate symbols for the modules to be debugged and specify the wanted breakpoints.

Debugging ICPs (Command Processors)

The following technique is useful for initially planting a breakpoint in a command processor:

1. Enter XDELTA via the DELTA keyin.

2. Place a Monitor breakpoint at SSC$ACPENT.

3. GO

4. Start up the user which associates the command processor to be tested.

5. When the breakpoint is reached, the command processor is about to be entered. The USE command UU,ICP will give access to the command processor for the purpose of planting breakpoints. Using the LINK map for the command processor, DEFINE appropriate symbols for the modules to be debugged and specify the wanted breakpoints.

Debugging ASLs (Alternate Shared Libraries) and User Programs

The following information is useful when debugging ASLs and user programs with XDELTA.

Alternate Shared Libraries are called directly from the user domain without involving the Monitor. For this reason it is necessary to first "catch" the user program before planting breakpoints in the ASL. On a quiescent system this may be accomplished by the following technique:

1. Enter XDELTA via the DELTA keyin.

2. Place a Monitor breakpoint at FMM$ASSMRG.

3. GO

4. Cause the user program which will call the ASL to be fetched.

5. When the breakpoint is reached, both the user program and the ASL (if any) will be accessible. To access the user program enter the USE command UU. To access the ASL enter the USE command UU,ASL.

Note: If the system is not sufficiently quiescent to "catch" the appropriate user, use SPY.X to find out what user number the user program will run as. Then use an IF S_CUN = .nn clause on the Monitor breakpoint specified in Step 2.

Debugging ASLs at Recovery

When debugging the portion of an ASL that is called at the time of System Recovery, use the following steps to set breakpoints in the ASL:

1. Entry to XDELTA is made following the "DO YOU WANT DELTA ?" question, which comes out to the system console just as the system is rebooted. Just answer yes to the question and the system will go to XDELTA.

2. From GHOST1's link map, find the ENTDEF ASL$RECOVER. A link map for GHOST1 can be obtained by using the linker MAP command on the GHOST1 run unit supplied with the CP-6 release tape.

3. Place a breakpoint in GHOST1 at this address. (GHOST1 is the current user at this time so just specify UU to address it.)

4. When the breakpoint is hit, specify UU,ASL and plant whatever breakpoints you need to debug the ASL.

It should be remembered that all breakpoints planted in the user by the debugger under test will be first reported to XDELTA. The GOTRAP command should be used to pass these faults on to the user program.

Operational Considerations When Using XDELTA

Initial entry to XDELTA may be caused by the keyin DELTA on the system console. This is usually adequate. If, however, this does not work or it is undesirable to have the KEYIN system ghost as the current user, then XDELTA may be entered by typing CNTRL-Y, CNTRL-Y, RETURN (or ESC, ESC, EOM on some older consoles.)

Entry to XDELTA may also be made by causing the machine to perform an EXECUTE fault via the maintenance panel or DPU. This procedure is different for each type of CPU and is described fully in the Operations Reference Manual, CE34, under System Start-up and Recovery. (N.B. It is not always possible to proceed correctly from XDELTA if it is entered via an EXECUTE fault.) The EXECUTE fault may also be used to interrupt XDELTA from a lengthy command such as DUMP. In this case there is no problem with continuing.

Addressing with XDELTA - Domain Specification

The USE command specifies to XDELTA what is to be addressed for all memory references. Depending on what this command specifies, XDELTA will reference real memory locations, virtual locations through a specified page table, or a domain of virtual space specified by some combination of linkage segments and page tables. When XDELTA is entered, a default USE command is assumed which sets addressability to the currently executing domain. This can be the Monitor, a user program, or an ICP, IDB, or ASL executing on behalf of a particular user. When this default addressability is established, XDELTA will address anything in the current process's domain including segments framed by descriptors on the argument and parameter stacks, as well as those framed by the current process's descriptor registers.

USE REAL[,AARDVARK] USE REAL[,XDELTA] USE REAL[,.offset] e.g. UR	Addresses real memory locations directly. Optional argument adds appropriate bias for AARDVARK, XDELTA, or whatever is specified.
USE PT@.pta e.g. UPT@.120400	Address through any page table. pta is the 18 bit page table address as it appears in WSPTD (word address modulo 64).
USE MON e.g. UM	Address through Monitor's linkage segment and Monitor's and current user's page tables.
USE USER,USER e.g. UU	Address current user through his linkage segment and page table. ",USER" is assumed if omitted.
USE USER#.nn,USER e.g. UU#.21	Address user number .nn through his linkage segment and page table.
USE USER@.pta,USER e.g. UU@.102030	Address user whose page table is at page table address .pta through that page table and his linkage segment. (.pta is the 18-bit word address of the page table modulo 64.)
USE USER[#.nn],ASL USE USER[@.pta],ASL e.g. UU,A	Address ASL through page table and appropriate linkage segment of the specified user.
USE USER[#.nn],IDB USE USER[@.pta],IDB e.g. UU#.33,ID	Address debugger through page table and appropriate linkage segment of the specified user.
USE USER[#.nn],ICP USE USER[@.pta],ICP e.g. UU,IC	Address command processor through page table and appropriate linkage segment of the specified user.
USE USER[#.nn],MON USE USER[@.pta],MON e.g. UU#.53,M	Address the Monitor through its page table and linkage segment and the page table of the specified user.
USE e.g. U	Resets addressing to that of the executing process at the time of entry to XDELTA.

Note: XDELTA allows data access to memory through type 1 descriptors. This means that the special symbols $LSR, $ASR, $PSR, and $SSR may be used as pointers in a pointer qualified reference such as:

```
$LSR->.26
```

XDELTA allows modification of memory through descriptors that have only read access to memory, and onto pages that are write protected.

Control of XDELTA's Input and Output

The commands which affect XDELTA's input and output are READ, OUTPUT, ECHO, and to some extent DUMP. XDELTA has only one input stream, which is used for reading commands. This input stream is directed by the READ command. The available sources for commands are the system console, the card reader, and, during the boot process only, the patch deck. The patch deck is a collection of input from the boot tape, the card reader, and the system console, under control of commands to AARDVARK. (Reference the Operations Reference Manual, CE34, System Start-up and Recovery.) During debug sessions, commands may only come from the system console or the card reader. The card reader must be the IOM-connected card reader known to AARDVARK at boot time.

The READ command:

R[EAD] P[ATCH]	Read commands from AARDVARK controlled patch deck.
R[EAD] C[ARD]	Read commands from the card reader. At End-Of-Deck input reverts to the console.
R[EAD] M[E]	Read commands from the system console.

XDELTA has two output streams. One, called the echo stream, directs the output of echoed commands and the output of the DUMP command. (Commands are echoed whenever they are read from a device other than the system console.) This stream is controlled by the ECHO command. The other output stream is called the output stream and is used for all other output (except command prompts) and is controlled by the OUTPUT command. The only options for these commands are LP, ME, and NO. ME is the system console and LP is the IOM-connected line printer known to AARDVARK at boot time. NO turns off the stream. The ECHO stream defaults to LP, and the OUTPUT stream defaults to ME.

EC[HO] M[E]	Echoed commands and output of DUMP command go to the system console.
EC[HO] L[P]	Echoed commands and output of DUMP command go to the line printer.
EC[HO] N[O]	Throw away output of command echoing and the DUMP command.
OU[TPUT] M[E]	Direct output of all commands except DUMP to the system console.
OU[TPUT] L[P]	Direct output of all commands except DUMP to the line printer.
OU[TPUT] N[O]	Do not print the output of commands.

The DUMP command may optionally specify an output device, LP or ME, which is effective only for the single command.

```
    DU[MP] ..... O[N] M[E]
or  DU[MP] ..... O[N] L[P]
```

The LP option on these commands specifies the printer known to AARDVARK at boot time. AARDVARK exercises some degree of control over this device in that if AARDVARK is given the skip option, "S", at boot time in response to its request to ready the printer, AARDVARK sets a flag indicating that all subsequent output to that device through mini-I/O is to be ignored. If XDELTA printed output is desired and this skip condition has been set, the following sequence must be followed to reset the flag:

```
    >MINI
AARDVARK HERE
  ? OU LP
  ? GO
    >DUMP .....
```

Control of Faults

During normal system operation, all program faults are handled by the system fault handler in the Monitor without XDELTA's awareness. In order to allow XDELTA to function, faults must be given to XDELTA whenever one or more breakpoints are set. This function is handled automatically by communication between XDELTA and the system fault handler. In order that the user of XDELTA have control over the handling of faults other than breakpoints, XDELTA has the KEEP command. The KEEP command specifies which classes of faults XDELTA is to report (KEEP) and which to give back to the system fault handler for normal processing. XDELTA will keep control of just Monitor SCREECHes, just Monitor faults, or all faults. When debugging the Monitor it is usually desirable to have XDELTA report any Monitor faults, but not user faults. When debugging any other domain, however, it is usually necessary to have XDELTA report on all faults. For this reason it is recommended that debugging of these domains take place on a relatively quiescent system. Whenever XDELTA reports a fault it brings the entire system to a halt and prompts for commands. XDELTA can be directed to return control of a reported fault to the system for normal handling with the GOTRAP command.

KE[EP] S[CREECH]	Instructs XDELTA and the Monitor to give control to XDELTA just before any SCREECH occurs. This option alone does not cause any system overhead for fault handling, all faults are directly handled by the system fault handler without XDELTA's involvement. Note: When XDELTA reports a SCREECH, the current environment is that of the SCREECH call unless the SCREECH was caused by a Monitor fault in which case the current environment is that at the time of the fault. This permits, in the case of a Monitor fault, the commands "GO" or "GO location" to continue execution without allowing the SCREECH to occur. When a SCREECH with a severity of 5 (snap) is reported to XDELTA, the GO command causes execution to resume without creating the snapshot dumpfile. In order to allow any SCREECH to continue normally, XDELTA must be given a QUIT command. (N.B. If XDELTA is given a QUIT command at any other time it causes a DISK boot!)

KE[EP] M[ON]	Instructs the system to give fault control to XDELTA and instructs XDELTA to report only those faults caused by the Monitor. User faults, including those caused by any special shared processor, are handed back to the system for normal processing.
KE[EP] A[LL]	Instructs the system to give fault handling to XDELTA and instructs XDELTA to report all faults.
G[O] T[RAP]	Instructs XDELTA to return control of the currently reported fault to the system for normal fault processing. This is only valid if entry to XDELTA was caused by a program fault.

Inactivation of Breakpoints by XDELTA

Some times while debugging domains other than the Monitor, XDELTA will set certain breakpoints in user or special processor domains inactive. This is due to the fact that the user through which the breakpoint was specified has been run down by the Monitor and the pages which contain the breakpoints are no longer accessible through that user. When this occurs, KILL the breakpoints and start over. This will usually mean having to SPIDER in a new copy of the special shared processor, as the old copy will have breakpoints (Derail instructions) left in it.

Guidelines for Command Processors

A command processor is to control the conditions under which a terminal session or a batch job takes place. Through command processor commands, the user controls resources, files, devices, comgroups, and terminals and causes user programs and standard shared processors to be executed.

User-written command processors reside in the command processor working space concurrently with IBEX and with one another. Only one command processor can be associated with a given user at a given point in time.

The following guidelines are provided for the system programmer who creates a command processor.

Entry to Command Processor

A command processor becomes associated with a user whenever:

- The command processor has been specified via the SUPER user authorization CPROC option, or
- One command processor issues an M$CPEXIT monitor service request specifying that another command processor be associated with a user.

Once associated with a user, a command processor is entered at its start address under any of the following conditions:

- The user is at Job Step (as is the case on the initial entry to the command processor).
- The user has aborted and is about to be run down.
- A time-sharing user has typed a Control-Y sequence on his terminal.
- A user program has issued an M$YC monitor service request.

There is no direct linkage between the command processor and the user program; instead, the interface is in the monitor.

The CP-6 monitor communicates the reason for entry to the command processor via bit settings in B$JIT.CPFLAGS1 as follows:

CP_JSTEP#	The user is at Job Step.
CP_RUND#	The user is about to be run down.
CP_YC#	The time-sharing user typed a Control-Y sequence.
CP_YCPMME#	The user program issued an M$YC.

Another interesting bit in B$JIT.CPFLAGS1 is CP_LOGOFF#. This bit is set whenever the system detects a line hang-up of a time-sharing terminal or an operator abort of a user. This bit may be set in conjunction with any of the other bits mentioned above. When set, it indicates to the command processor that no more job steps are allowed.

Command Processor Capabilities

The privileges afforded the command processor domain are:

- the ability to issue an M$CPEXIT monitor service request
- the ability to issue an M$FINDPROC monitor service request
- the ability to issue an M$ACCT monitor service request
- the ability to issue an M$OCMSG monitor service request
- the ability to issue an M$SCREECH monitor service request
- the ability to issue an M$XCONRTN monitor service request
- write access to the JIT
- write access to the *A file to effect DCB assignments (See "Effecting DCB Assignments" later in this section.)

- write access to the *S file for accounting purposes (See M$ACCT in Section 9 of the CP-6 Monitor Services Reference Manual and Appendix A of the CP-6 System Support Reference Manual, STARACC$KEY and ACCT_KEY.)

DCBs for Command Processor

There are three DCBs reserved for Command Processors. Because the first command processor written was called IBEX, these DCBs are called M$IBEX, M$IBEX1 and M$IBEX2. These DCBs should be referred to by their EQUs in CP_6_SUBS.:LIBRARY, i.e., M$IBEX#, M$IBEX1#, and M$IBEX2#, respectively.

A command processor may also use M$DO and M$UC as appropriate. Additional DCBs may be acquired by issuing an M$GETDCB monitor service request, unless the command processor is executing on behalf of an interrupted run unit and has used all of the DCB slots.

The CP-6 system generally recognizes that M$LL is used for listing purposes by the command processor. This DCB is acquired dynamically, or by passing one of the reserved DCB numbers through FPT_GETDCB.DCBNUM_ when calling M$GETDCB.

If acquiring M$LL, first attempt to acquire a dynamic DCB. This is because an interrupted run unit may already have an M$LL defined. The command processor should use that DCB if possible.

The M$UC DCB is only useful for interacting specifically with the user terminal of timesharing users (not the normal command stream). At other times it is tied to the "bit bucket" or NO# device.

The M$DO DCB is used for diagnostic output. If a command processor is effecting a DCB assignment for a user DCB that is also being used by the command processor (i.e. M$DO or M$LL), the command processor should close that DCB and reopen it to insure that the new assignment takes effect immediately.

Effecting DCB Assignments

A special form of the OPEN FPT, with the desired options set, is to be written into the *A file. The record is keyed by the name of the DCB.

The OPEN FPT should be invoked with PFMT="PTR". This changes all the vectors to pointers. Any necessary VLPs should be contained in the record and should be pointed to by the appropriate vector name.

DCBs #1 through #4 require some special processing. These DCBs are reserved for specification of files on the run unit invocation. There are four bits in the JIT that indicate the specification of these DCBs. They are B$JIT.PRFLAGS.SI, .UI, .OU and .LS. If records for these DCBs are written to the *A file, then the bit in B$JIT.PRFLAGS should be set. Upon re-entry to the command processor, if at job step (B$JIT.CPFLAGS1# & %CP_JSTEP# are true), these flags and the corresponding DCBs should be reset.

Refer to the following items in Section 14 of this guide:

- Standard Run Unit Invocation Format for Compilers
- DCB Usage Conventions

Addressing User Memory from Command Processor

The user's Job Information Table (JIT) is of primary interest to the command processor. The command processor may also receive commands in the user's command buffer via the M$YC monitor service.

User's JIT

A command processor has write access to the user's JIT through the JIT descriptor in the command processor's Linkage Segment.

The fields in the JIT are described in detail in Appendix A of this manual. Those that are of particular interest to the creator of a command processor are listed below. The command processor should take care that all other fields in the JIT remain intact.

CPFLAGS1

This word is used by the monitor and the LOGON processor to communicate job step information to the default command processor. Some of the flags are to be set by the default command processor to communicate information to the logoff command processor. This word may also be used by the command processor to communicate information across job step.

PRFLAGS

This set of flags is used to pass information from the command processor to standard shared processors or user programs based on the run unit invocation line or other commands.

CCBUF, CCARS and CCDISP

These fields should be set to reflect the latest run unit invocation.

USRERR and USRDCB

This word contains the error code that describes why the previous run unit aborted. JIT.USRERR.CODE is non-zero when there is an error to report. If CP_EXIT# is set in JIT.CPFLAGS1 then the command processor should not report it now (the exiting processor should have done so), but hold it for processing later, usually in response to a question mark command. JIT.USRDCB contains the DCB number that was associated with this error.

USRRNST and USRIC

USRRNST, along with the RNST masks, is used to determine the final status of the previous run unit. USRIC contains the address of where the user program terminated.

JSLEV, PSLEV and SSLEV

These three fields are used to control what level of statistics is to be reported to the user when at jobend (JSLEV), a proprietary processor is invoked (PSLEV) or at jobstep (SSLEV).

MODE

This field specifies the mode in which the user is running: M_INT# (interactive, online), M_BATCH# (batch), M_GHOST# (ghost) or M_TP# (transaction processing).

NEXTCC

This specifies the source of the next command stream read.

User Parameters for M$YC

In general, a command processor is not granted access to memory belonging to a user program. However, if the command processor is being entered as a result of an M$YC monitor service request, the monitor executes LTRAS to invoke the command processor, making the user's M$YC parameters available through the command processor's Parameter Stack.

Parameter 0 - frames the command which may consist of a TEXT string of up to 256 characters. The "bound" of this parameter (byte count - 1) is stored in B$JIT.YCOSZ.

Parameter 1 - frames the V area of the M$YC FPT. Bit 0 is set if ECHO = YES was specified. Bit 1 is set if NOERR = YES was specified.

The pointers to these parameters are defined in the module B_USRPTRS_D. These pointers are SYMDEFed as B$PS0$ and B$PS1$.

Exit from a Command Processor

The Command Processor communicates the action to be taken for this user via the various options of M$CPEXIT. This monitor service is used to:

- Initiate execution of a user program or shared processor.
- Resume execution of an interrupted program (following Control-Y or an M$YC service call).
- Associate a debugger with a (possibly interrupted) user program.
- Remove a user from the system.

Before a Command Processor issues an M$CPEXIT, it must close and release (M$RELDCB) all its DCBs and, in general, clean up. Data Segments obtained via M$GDS are released unless CP_KEEPDS# is set in B$JIT.CPFLAGS1.

The M$CPEXIT monitor service and its options are explained in the CP-6 Monitor Services Reference Manual, Section 8.

Guidelines for Debuggers

A debugger can be used to monitor and/or control the execution of a program in the user domain.

Entry to the Debugger

The debugger is entered under any of the following conditions:

1. Initial entry to the debugger:

    a. A program is started under the debugger.
    b. The debugger is invoked at Job-Step with no run unit associated.
    c. The debugger is invoked after a user program is in execution by striking Control-Y and asking for the debugger.

2. An overlay of the user's program is loaded.

3. A program is put into execution via M$LINK or M$LDTRC or an M$LINKed to program is restored.

4. An Exceptional Condition occurs other than:

    a. Line hang-up
    b. Operator !X key-in
    c. Bad call, and the user specified ALTRET.

5. A user is exiting an Exceptional Condition processing procedure.

6. The debugger is associated via an M$ALIB Service Request. If the program making the request is not executing under control of the debugger (see Item 1), DELTA is thereafter entered only on subsequent M$ALIB requests or for requests to be put under control of the debugger. Note that this may or may not be the initial entry to the debugger.

When the debugger is entered a standard Exceptional Condition frame containing a copy of the user's Safe-Store frame is placed in the debugger's TCB. The Exceptional Condition Code (B$EXCFR.ECC), Sub-code (B$EXCFR.SUBC) and Event ID (B$EXCFR.EVID) uniquely identify the condition that caused the entry to the debugger. The %SUB_EXC and %SUB_ECCDELTA macros from the system macro library provide string substitutions for the values of these fields as indicated in Tables 11-1 and 11-2.

Table 11-1. ECCs for Debugger

ECC	SUBC	Reason for Entry
ECC_DELTA#	SC_STARTU#	User program started under the debugger.
	SC_JOBSTEP#	Debugger was invoked at Job-Step time. No Run unit associated.
	SC_YC#	Post association of the debugger while the user program is in execution. B$EXCFR.EVID contains one of the following values: EVID_USER# EVID_AUTOS#
ECC_OLAY#	Contains the Node#	Overlay has been loaded. B$EXCFR.EVID contains one of the following values from the M$OLAY FPT: EVID_CANCEL# EVID_ENTER# EVID_NOPATH#
ECC_LINK#	SC_MLINK#	User program entered via M$LINK.
	SC_MLDTRC#	User program entered via M$LDTRC.
ECC_LRTN#	-	M$LINKed to program has been restored.
ECC_ALIB#	-	Debugger was invoked via M$ALIB. B$ALIB[F].CMDSZ contains the byte size of the command. B$ALIB[F].REPLYSZ contains the byte size of the reply area. B$ALIB[F].WHO is set as follows: SC_AUSR# - User Program SC_AASL# - Alternate Shared Library SC_ASHR# - Standard Shared Processor SC_EXUO# - Execute-only Run unit
ECC_EXCRTN#		Exit from a user's exceptional Condition procedure. B$EXCFR.EVID contains the address of the call.
	SC_TRTN# SC_MERC# SC_MERCS# SC_RETRY# SC_RETRYS# SC_XCONXIT#	M$TRTN M$MERC M$MERCS M$RETRY M$RETRYS Final Exit from Exit Control. In this case B$EXCRTN.TYP has the following values: XCON_EXIT# - M$EXIT XCON_ERR# - M$ERR XCON_XXX# - M$XXX
ECC_DBRK#	-	Data Break Point

Table 11-2. ECCs for User Exceptional Condition

ECC	SUBC	Reason for Entry
ECC_TIMER#	-	M$STIMER specified interval expired.
ECC_EVENT#	As specified by user	Event over which the user has requested control occurred.
ECC_INT#	SC_BRK#	Time-sharing terminal break key.
	SC_BYC#	The debugger request from the Command Processor when the debugger is already associated.
ECC_XCON#	See CE33 Section 6	User exit condition, normal or abnormal.
ECC_PMME#	See CE33 Section 6	Error on Monitor Service request. No ALTRET specified on user's call.
ECC_ARITH#	See CE33 Section 6	User caused an Arithmetic fault.
ECC_PROG#	See CE33 Section 6	User caused a Programmed fault.
ECC_ERROR#	See CE33 Section 6	User caused an Error class fault.

When the debugger is entered because of a user's Exceptional Condition, the ECC and the remainder of the frame reflects what would have been placed on the user's TCB had the user not been running under the debugger and had established control of the specific condition. No determination is made as to user specified Exceptional Condition control requests. The Exceptional Condition frame is moved to the user's TCB and the procedure to handle the condition is entered only when this action is specified via options of the M$DRTN FPT. (Refer to the description of the SETECC and ECC options below.)

Word 1 of the TCB frame is non-zero if the ASL was in control upon entry to the debugger. Note that this can happen only if the user has hit Break and the ASL has not requested break control or if the user has hit Control-Y and invoked the debugger while the ASL is in control.

Debugger Capabilities

The power of the debugger comes from its position of control between the monitor and the user program for all exceptional conditions. The special privileges afforded a debugger are:

- the ability to set the data breakpoint software flag in the user's page table. See M$SSC in Section 8 of the Monitor Services Reference Manual.
- the ability to write on user Instruction Segment and user dynamic data segment pages.
- the ability to use the M$DRTN monitor service
- the ability to issue an M$SCREECH monitor service request
- the ability to issue an M$XCONRTN monitor service request.

DCBs for Debugger

The DCB Table entry for DCB 9 is reserved for use by the debugger. This DCB should be referred to as M$DELT#; its EQU is in CP_6_SUBS.:LIBRARY. Any other DCBs used are not protected from use by the user program.

Addressing User Memory from Debugger

The debugger has access to the user's Working Space through descriptors stored in the Special Descriptor Access descriptor slots in the debugger's Linkage Segment. The following pointers (which are DEFed in B_USRPTRS_D) may be used to access the user's area:

```
B$SPCL1$ ->   the user's Safe-Store frame
B$SPCL2$ ->   the user's Linkage Segment
B$SPCL3$ ->   the user's Argument Segment
B$SPCL4$ ->   the user's Parameter Segment
B$SPCL5$ ->   the user's Instruction Segment
```

The first four of these descriptors are type 1; the user's Instruction Segment descriptor is type 0. The Special Access Descriptors 2 through 5 are a copy of those from the user's Safe-Store frame. Unless the debugger is being entered as a result of an M$ALIB from an ASL, shared processor or execute-only run unit, the Page Table write control bit for procedure pages in the user's ISR is set prior to entry to the debugger; it is reset when the debugger returns to the monitor via M$DRTN.

The monitor normally enters the debugger via the LTRAD instruction. However, if the debugger is being entered as a result of an M$ALIB request, the monitor executes LTRAS to invoke the debugger making the user's M$ALIB FPT available through the debugger's Parameter Stack:

```
Descriptor 0 - frames the Debugger name.
Descriptor 1 - frames the area containing the command.
Descriptor 2 - frames the area where the debugger may return a reply.
Descriptor 3 - frames the V area of the M$ALIB FPT.  FPT$ALIB_V may be used to
               define the based structure of this area.
```

The pointers to these parameters are defined in the module B_USRPTRS_D and are SYMDEFed as B$PS0$, B$PS1$, B$PS2$ and B$PS3$.

Ten words of user data space are set aside for use by debuggers. These words are inserted by the linker and are found by a pointer which is located in the second word of the user's first procedure page. The sixth through tenth words of this space are reserved for use by XDELTA.

Data Breakpoints

A debugger may use the M$SSC service to cause a data breakpoint flag to be set for specified page(s) owned by the user (see M$SSC in the Monitor Services Reference Manual, Section 8).

Once this flag is set, the monitor causes the debugger to be entered if the user causes a fault because of trying to modify data on a page for which this flag is set. ECC is set to ECC_DBRK# in the TCB frame upon entry to the debugger. The debugger can cause the data breakpoint flag to be reset by again using the M$SSC service or by the M$DRTN DBRK option (see M$DRTN in the Monitor Services Reference Manual, Section 8).

Exit from a Debugger

The debugger returns control to the user program via the M$DRTN service request. This monitor service is used to:

- Change the registers and/or IC of the user program
- Resume execution of the user program
- Terminate execution of the user program
- ALTRET to the user's M$ALIB monitor service request
- Disassociate the debugger from the user program

The M$DRTN monitor service and its options are explained in the CP-6 Monitor Services Reference Manual, Section 8.

Guidelines for Alternate Shared Libraries

An Alternate Shared Library (ASL) is a special shared processor that can be called directly from the user program via the CLIMB instruction. Its primary use is for the implementation of a data base manager; it may also be used to implement other applications such as I/O graphics packages, etc.

Associating an ASL with the User

An ASL is associated with a user whenever that user begins execution of a run unit that has an ASL specified in the HEAD record or when the ASL is the object of an M$ALIB monitor service request.

There are two methods of associating an ASL with a user run unit. One method is to automatically associate it when the run unit is invoked. This happens if the head record of the run unit contains an ASL name. The head record contains an ASL name if either the ALTSHARELIB option is specified in the link of the run unit, or one of the object unit head records that comprise the run unit contains an ASL name (COBOL associates I-D-S/II by using the latter method). In either case, the LINKer uses the ASL to satisfy the references to the symbols that correspond to the function being provided by the ASL.

An ASL may also be associated dynamically at run-time by a user program that issues a call to the M$ALIB monitor service. In this case the ASL is not present until the M$ALIB is issued; no ALTSHARELIB on the LINK command or ASL name in the head record is necessary. Instead the ASL should be included in the LINK command as if it were a normal object unit. This causes the LINKer to satisfy references to the functional entry points of the ASL run unit without forcing it to be automatically associated.

Defining the Function Codes of the ASL

The creator of the ASL must define symbolic function names to distinguish the various operations the ASL performs. This is necessary since there is only one actual entry point into the ASL. The symbols used to identify the functions are defined in a module, SYMDEFed, and that module is linked as part of the ASL.

Because it is the symbolic values that define the functions, and not the contents of the memory locations associated with the symbols, the symbols are usually equated to specific sequential constants in a BMAP routine.

For example, for an ASL capable of performing function "X" and function "Y", the BMAP module contains:

```
        SYMDEF   ASL$FUNCTION_X
        SYMDEF   ASL$FUNCTION_Y

ASL$FUNCTION_X   EQU      2
ASL$FUNCTION_Y   EQU      3
```

Note that the value of 1 is skipped. This is because the value of 1 is reserved to indicate ASL recovery and should only be used for that function.

User Calls to an ASL

The CP-6 Operating System provides a CLIMB instruction interface for communication between the user and an associated Alternate Shared Library. A slot has been reserved in every user's Linkage Segment for the ASL Entry Descriptor. The descriptor is built from information contained in the ASL run unit. When a user program wishes to cause entry to an ASL, it issues a CLIMB instruction in which the S,D field is coded to reference the Linkage Segment descriptor that points to the ASL.

This CLIMB instruction is generated by the PL6 compiler when the user calls a procedure that is defined using the CONV type 2 attribute of the ENTRY declaration.

If no parameters are to be passed:

```
DCL asl_function_name ENTRY CONV(2,0)
CALL asl_function_name;
```

If parameters are to be passed:

```
DCL asl_function_name ENTRY(1) CONV(2,code1)
CALL asl_function_name(asl_vector_list);
```

where:

code1 = the number of parameters to be passed.

The "asl_function_name" is placed in the object unit as a SYMREF rather than an ENTREF. The SYMREF is satisfied by the LINKer from the ASL associated with the run unit as described above. This is the function code that is passed from the user program to the ASL in the X0 register.

The "asl_vector_list" contains a vector framing each parameter that is to be passed to the ASL. This is similar to the FPT passed to the monitor on a monitor service request.

For more information on the descriptors, parameter stacks, and CLIMB instructions, see DPS 8 Assembly Instructions reference manual (DH03).

A user program that calls an Alternate Shared Library relinquishes control until the library returns control to the user. User-established break control, timer run-out, and event reporting are deferred until the ASL returns control to the user program.

Building an ASL System File

A programmer who is creating an ASL should provide other users with an INCLUDE file that contains the ENTRY declarations for each service the ASL performs. This INCLUDE file could also contain macros to generate the vector lists for those services that require parameters.

As explained above the INCLUDE file contains the DCLs that correspond to the symbolic function codes for the ASL entry points:

```
DCL ASL$FUNCTION_X ENTRY CONV(2,0);
DCL ASL$FUNCTION_Y ENTRY CONV(2,3);
```

These declare statements define the two ASL functions and indicate that they require zero and three vectors respectively.

The macros to generate the vector list for the ASL functions can be built in the same manner as those contained in the CP_6 system file. That is, the structure for each function contains a list of vectors followed by a level 2 sub-structure that is framed by the first vector in the structure. The level 2 sub-structure contains the additional parameters necessary for the function. This whole structure is generated within a MACRO that can specify default options and parameters. An example of ASL$FUNCTION_Y is as follows:

```
%MACRO ASL$VECTORS_Y(NAME=ASL$VECTORS_Y,
         OPTION1(YES='1'B,NO='0'B)='1'B,
         OPTION2(RED=1,ORANGE=2,YELLOW=3,GREEN=4,BLUE=5,INDIGO=6,VIOLET=7)=0,
         INPUT=NIL,
         OUTPUT=NIL);
DCL 1 NAME STATIC,
         2 V_ VECTOR INIT(VECTOR(NAME.V)),
         2 INPUT_ VECTOR INIT(ADDR(INPUT)),
         2 OUTPUT_ VECTOR INIT(VECTOR(OUTPUT)),
         2 V DALIGNED,
            3 OPTION1# BIT(1) INIT(OPTION1) UNAL,
            3 OPTION2# UBIN(3) INIT(OPTIONT2) UNAL;
%MEND;
```

The user who then wishes to access the ASL builds a program that contains the following statements:

```
A:       PROC MAIN;
%INCLUDE ASL$SYSTEM_FILE
         %ASL$VECTORS_Y(NAME=FUNCTION_Y,INPUT=IN,OUTPUT=OUT,OPTION1=YES);

DCL IN CHAR(15) STATIC INIT('Hello there ASL');
DCL OUT CHAR(15) STATIC;

         CALL ASL$FUNCTION_Y(ASL$VECTORS_Y);

END A;
```

Refer to the CP-6 PL-6 Reference Manual (CE44), Section 10 for details on the CONV attribute and Section 12 for MACRO definition. Calls from other languages could be done through a BMAP interface.

Entry to ASL

The main routine of the ASL should be a BMAP routine. This is necessitated by two facts. First, the function code associated with this call is placed in Register X0 by the CLIMB instruction. Second, the BMAP routine has to determine whether the ASL has been entered before or not and set up an AUTO Stack as described earlier in this section.

The routine makes use of an initialized data segment. Any segment besides AUTO may be used. The following example uses Dynamic segment 8. Thus the BMAP routine assumes the following PL-6 declarations are part of the ASL code:

```
DCL 1 ASL_STATUS STATIC AREADEF(DS8SID) ALIGNED,
        2 INITED UBIN WORD INIT(0),
        2 BREAK  UBIN WORD INIT(0);
DCL 1 ASL_STATUS$ PTR CONSTANT SYMDEF INIT(ADDR(ASL_STATUS));
```

The entry module for the example program would be coded as follows:

```
*
        ENTDEF   ASL$ENTRY          The ASL start address is ASL$ENTRY
*
*       Define the symbols that correspond the the functions
*       supported by the Alternate Shared Library
*
        SYMDEF      ASL$RECOVERY
        SYMDEF      ASL$FUNCTION_X
        SYMDEF      ASL$FUNCTION_Y
*
ASL$RECOVERY      EQU   1
ASL$FUNCTION_X    EQU   2
ASL$FUNCTION_Y    EQU   3
*
*      References to the ASL subroutines written in PL-6
*
       ENTREF  ASL$INIT
       ENTREF  ASL$RECOVERY
       ENTREF  ASL$ROUTINE_X
       ENTREF  ASL$ROUTINE_Y
*
       ENTREF  X66_MSTATIC       and external BMAP routines
*
*      References to external segments
*
       SYMREF  B$PIA$          We will link with B_USRPTRS_D
       SYMREF  B$AUTO$
       SYMREF  B$IS$
*
*        References to the AREADEF structure.
*
       SYMREF  ASL_STATUS$
INITED EQU     0
BREAK  EQU     1
*
*       CP-6 monitor service macros
*
ALTRETFLG   BOOL     400000
INTRTN      EQU      17
XXX         EQU      3
*
M$INTRTN    MACRO
*       This macro generates a monitor service call that returns
*       control to the user domain just as if the normal ASL
*       return were issued.  In addition it causes the operating
*       system to cause a BREAK event to be reported on the
*       user.  Thus the ASL can transfer breaks it receives
```

```
*         to the user program when it is ready.
          PMME      17+ALTRETFLG
          ENDM
M$XXX     MACRO
*         This macro generates a standard M$XXX.
          PMME      XXX
          ENDM
*
ASL$ENTRY   NULL
          EAX7      0,0                 Copy function code to X7
          TMOZ      BADFCN              If legal codes are all positive
          CMPX7     ASL$FUNCTION_Y      Highest defined function code
          TPL       BADFCN              Out of range
          LDI       =04010              Set HEX and OVRFL mask
*
*         See if this is the initial entry
*
          LDP7      ASL_STATUS$         Pointer to AREADEF structure
          LDA       INITED,,7           Have we been entered before
          TNZ       SETAUTO             Yes
          LDA       1,DL                Indicate have been entered
          STA       INITED,,7           And save
*
*       Initialize an AUTO stack
*
          TSX0      X66_MSTATIC
          ZERO      0,0
*
*         Call any ASL initialization routines.
*
          TSX1      ASL$INIT
          ZERO      0,0
          TRA       FUNCTION
*
*         Set PR2 to current top of auto stack
*
SETAUTO   LDP2      B$AUTO$             Make sure base pointer set up.
          LXL2      0,,2                Offset to top AUTO frame
          SWDX      1,2,2               Compute pointer to top.
*
*       Now go do the requested function
*
FUNCTION  LDX1      CHKBREAK,DU         For PL-6 returns
          TRA       *+1,7
          TRA       BADFCN              0 is not used in this example
          TRA       ASL$RECOVERY        1 is reserved for recovery
          TRA       ASL$ROUTINE_X
          TRA       ASL$ROUTINE_Y
*
*       RET is the common exit point
*
*         It is the ASL's responsibility to save BREAK events received
*         when it is in control and report them to the user.  The
*         ASL must set up an ASYNC routine to intercept breaks, and
*         must set the flag STATUS.BREAK to be non-zero.  This
*         interface will check that flag and report the break event
*         to the user if it is set.
*
CHKBREAK  NULL
          INHIB     ON
          LDP7      ASL_STATUS$         Set pointer to status block
          LDA       BREAK,,7            Check for break during ASL
          TZE       RET                 None.  Normal return
*
*         Break hit during ASL execution.  Report event to user.
          LDA       0,DL                Clear old flag
          STA       BREAK,,7
```

```
         M$INTRTN
*
RET      NOP
         EXIT
         INHIB    OFF
*
*      Issue M$XXX if the user passes a bad function code
*
BADFCN   M$XXX
         END      ASL$ENTRY
```

ASL Capabilities

An ASL has the following special abilities:

- the ability to access the *I file
- the ability to issue an M$SCREECH monitor service request
- the ability to issue an M$XCONRTN monitor service request
- the ability to issue an M$INTRTN monitor service request.

DCBs for ASL

There are no special DCBs for ASLs. However, an ASL may use the DCBs defined by the user, or acquire his own DCBs using the M$GETDCB monitor service. The XONLY option can be used to disallow user programs from accessing the ASL's DCBs. See the discussion of M$OPEN in the Monitor Services Reference Manual for details.

Addressing User Memory from ASL

Each parameter passed by the call to the ASL corresponds to a vector specified as part of the user's call. When the ASL is entered, the hardware builds a descriptor on the parameter stack associated with each of these vectors. Thus to access the first parameter, the ASL uses a pointer with the SEGID referencing parameter stack entry zero. The pointers for up to nine parameters are defined in the module B_USRPTRS_D. These pointers are SYMDEFed as B$PS0$, B$PS1$, B$PS2$ ... B$PS8$.

Thus in the example above, to access the memory framed by ASL$VECTORS_Y.INPUT_, B$PS1$ would be used. To access ASL$VECTORS_Y.V.OPTIONS1, B$PS0$ would be used. In the second case a structure must exist that defines the V area of the ASL$VECTORS_Y structure, without the additional vectors. (The pointer B$PS0$ will point to ASL$VECTORS_Y.V; using this pointer in combination with the ASL$VECTORS structure will not work). This can be automatically generated by the X account tool FPTCON which converts the V areas of FPT-like structures for use after a CLIMB has been issued.

If the creator of an ASL defines functions that require more than nine parameters, pointers to these parameters may be built as follows:

```
DCL B$PS9 UBIN CONSTANT INIT(9);
DCL B$PS9$ REDEF B$PS9 PTR;
```

Exit from an ASL

When the ASL has completed its functions, it must return to the user. This is done by performing an outward CLIMB, which restores the user environment and returns control to the user program. Note that this is done in the example above. When the PL-6 routine ASL$ROUTINE_X or ASL$ROUTINE_Y simply returns, control transfers to 'RET' in the setup routine which performs the outward CLIMB.

ASL Recovery

The key to ASL recovery is the *I file. This file can only be accessed from the ASL's domain. The existence of the *I file causes GHOST1 to associate and call the ASL during the recovery procedure with a function code of 1. It is the ASL's responsibility to write records to the *I that are useful during the recovery process. As an example, I-D-S/II writes before-images of updates to the *I and uses them to back out of partial updates if a deadlock, abort or recovery occurs.

As part of the recovery process, the JIT for each user is written to the dump area of the system disk. After the system has re-booted itself, the system ghost, GHOST1, is entered where further recovery functions are performed. One of these functions is to access the user JITs from the dump area to determine if ASL recovery entry is indicated.

If the disk address of the FIT for the *I file in the user's JIT is non-zero, that address is moved to the *I entry of GHOST1's B$JIT.STAR.DA and the file is opened (test mode) in order to locate the FPARAMS. The PROCATTR information in the FPARAM table (if any) is assumed to contain the TEXTC name of the ASL that was associated with the user at the time of the Screech.

This name is then used as the NAME parameter of an M$ALIB request. If the ASL has been established as a Special Shared Processor via the SPROC command to TIGR, the ASL is now associated with GHOST1.

GHOST1 then causes entry to the ASL via the CLIMB instruction. No parameters are passed; X0 is set to 1. The routine to handle recovery must, therefore, correspond to a function code of 1. At this point the ASL may do whatever is required for closing the data base.

Debugging an ASL

An ASL may be debugged using XDELTA as described earlier in this section. However, since the interface between the user program and an ASL is through the hardware CLIMB interface and not through the monitor, considerable debugging may be accomplished by linking the ASL object units and the user program object units into one run-unit and using DELTA to debug.

The modules that make up the ASL may be linked with the modules of the user program that calls that ASL to form a single run-unit that is to be executed in the user domain under DELTA. The calls from the user program to the ASL need to be changed from the inter-domain CLIMB via the ASL entry descriptor to an intra-domain CLIMB through the user's Instruction Segment descriptor. The difference between an inter-domain CLIMB and an intra-domain CLIMB is as follows. The IC is loaded from the 18-bit entry location contained in the Entry descriptor if the CLIMB is inter-domain, and from the address field of the CLIMB instruction if intra-domain.

In order to make an intra-domain call to an ASL, the actual CLIMB instruction must be coded in BMAP. Because of this fact, the PL-6 routines that call an ASL must be modified to call a BMAP interface module which will then issue the intra-domain CLIMB instruction.

In order to facilitate switching from an inter-domain interface to an intra-domain interface, it is recommended that all the calls to the ASL be issued from a BMAP module. That is, the ASL system file would contain the macros to generate the vector lists, and ENTRY declarations for the BMAP routine names for each function of the ASL. The BMAP routine names cannot be the same as the ASL function names although they should be similar enough to identify the functions they represent. The ASL system file would then contain:

```
%MACRO ASL$VECTORS_X( ...

%MEND;
%MACRO ASL$VECTORS_Y( ...

%MEND;
DCL A$FUNCTION_X ENTRY(1);
DCL A$FUNCTION_Y ENTRY(1);
```

The PL-6 routine wishing to call the ASL contains statements as follows:

```
%INCLUDE ASL$SYSTEM_FILE;
         %ASL$VECTORS_X;

         CALL A$FUNCTION_X(ASL$VECTORS_X);
         CALL A$FUNCTION_Y;
```

The BMAP routine then issues the CLIMB to the ASL. It is a simple matter to code the BMAP routine to issue both types of the CLIMB instruction depending on a patch being applied. Thus only one version of the interface is necessary to user either a real production ASL or a debug version.

Once the type of ASL has been decided upon and called, it is no longer valid to switch types. That is, if an inter-domain call is issued to the ASL, it is not valid to modify the BMAP interface to start issuing intra-domain calls and vice versa. The reason is that the two different types will actually call two different instances of the ASL (one that is a real ASL and one that is linked into the user's run unit).

The following is an example of a BMAP interface routine. This routine is coded to issue the inter-domain CLIMB by default. If the instruction at DEBUG is modified to a NOP, the routine issues intra-domain CLIMBs. (Of course it is also necessary to LINK the ASL code into the run unit as mentioned previously, for the intra-domain CLIMB to work).

```
*
*         Define the entry points to this routine
*
          ENTDEF    A$RECOVERY
          ENTDEF    A$FUNCTION_X
          ENTDEF    A$FUNCTION_Y
*
*         Define patchable location for ease of access
*
          ENTDEF    DEBUG
*
*         Reference the ASL function codes
*
          SYMREF    ASL$RECOVERY
          SYMREF    ASL$FUNCTION_X
          SYMREF    ASL$FUNCTION_Y
*
*         Reference to the ASL main entry point
*
          ENTREF       ASL$ENTRY
*
*         Other external references that are required
*
          ENTREF    X66_AUTO_1            Auto references
          ENTREF    X66_ARET
          SYMREF    B$AUTO$

          SEGREF    ISSID                 For intra-domain CLIMB
          SEGREF    ASLENTSID             For inter-domain CLIMB
*
*         Now the actual code
*
A$RECOVERY     NULL
          TSX0      X66_AUTO_0               Set up 0 parameters
          ZERO      4,0
          LDX0      ASL$RECOVERY,DU          Put function code in X0
          LDX1      0,DU                     Set for no vectors
          TRA       DEBUG
A$FUNCTION_X   NULL
          TSX0      X66_AUTO_1               Set up 1 parameter
          ZERO      4,1
          LDX0      ASL$FUNCTION_X,DU        Put function code in X0
          LDP0      ,0                       PR0 now points to the vectors
          LDX1      3,DU                     Set for 3 vectors
          TRA       DEBUG                    Go issue the CLIMB
A$FUNCTION_Y   NULL
          TSX0      X66_AUTO_0               Set up 0 parameters
          ZERO      4,1
          LDX0      ASL$FUNCTION_Y,DU        Put function code in X0
          LDX1      0,DU                     Set for no vectors
          TRA       DEBUG
   .
   .
   .
DEBUG     TRA       INTER,1                  Default to issue inter-domain
          TRA       INTRA,1                  Issue intra-domain CLIMB
INTRA     TRA       INTRA0                   Intra-domain.  Zero vectors.
          TRA       INTRA1                   Intra-domain.  One vector.
          TRA       INTRA2                   Intra-domain.  Two vectors.
          TRA       INTRA3                                  Three vectors.
*
*         The ENTER generates a CLIMB instruction.  Since an entry
*         descriptor is not being used in the intra-domain version,
*         the ASL entry point must be specified.  The X0 indicates
*         that the function code is already in X0 and should not
*         be modified.
*
```

```
INTRA0   ENTER    ISSID,0,(ASL$ENTRY,0)
         TRA      FIXAUTO
INTRA1   ENTER    ISSID,1,(ASL$ENTRY,0)
         TRA      FIXAUTO
INTRA2   ENTER    ISSID,2,(ASL$ENTRY,0)
         TRA      FIXAUTO
INTRA3   ENTER    ISSID,3,(ASL$ENTRY,0)
         TRA      FIXAUTO
*
*        An intra-domain climb destroys the SEGIDs of all the
*        pointers in the pointer registers.  Since PL-6 assumes
*        that PR2 always points to the top of auto, it is necessary
*        to restore PR2 before returning.
FIXAUTO  LDP2     B$AUTO$                 Pointer to the beginning of AUTO
         LXL2     0,,2                    Negative current size.
         SWDX     1,2,2                   Reset PR2 to correct value
         TRA      RETURN
*
*        The ENTER generates a CLIMB instruction.  The symbol
*        ASLENTSID references an entry descriptor that describes
*        the ASL and already contains the start address of the ASL.
*        The ENTER therefore does not specify the ASL entry point.
*        Again, X0 indicates that the function code is already
*        in X0.
*
INTER0   ENTER    ASLENTSID,0,(0,0)
         TRA      RETURN
INTER1   ENTER    ASLENTSID,1,(0,0)
         TRA      RETURN
INTER2   ENTER    ASLENTSID,2,(0,0)
         TRA      RETURN
INTER3   ENTER    ASLENTSID,3,(0,0)
         TRA      RETURN
*
RETURN   TSX2     X66_ARET
*
         END
```

The ASL entry module must also be changed to recognize the different conditions under which the ASL might be called. In the case of an intra-domain CLIMB the ASL should use the existing AUTO segment instead of initializing its own. In addition it is sometimes useful to know within the ASL if it has been called by an inter-domain or intra-domain CLIMB.

To facilitate the latter case, a field can be defined in the structure ASL_STATUS as follows:

```
DCL 1 ASL_STATUS STATIC AREADEF(DS8DIS),
        2 INITED UBIN WORD INIT(0),
        2 BREAK  UBIN WORD INIT(0),
        2 INTRA  UBIN WORD INIT(0);
```

Where ASL_STATUS.INTRA is non-zero if an intra-domain CLIMB is issued. The following is an updated version of this interface:

```
*
         ENTDEF   ASL$ENTRY          The ASL start address is ASL$ENTRY
*
*        Define the symbols that correspond the the functions
*        supported by the Alternate Shared Library
*
         SYMDEF       ASL$RECOVERY
         SYMDEF       ASL$FUNCTION_X
         SYMDEF       ASL$FUNCTION_Y
*
ASL$RECOVERY      EQU   1
ASL$FUNCTION_X    EQU   2
```

```
ASL$FUNCTION_Y    EQU   3
*
*       References to the ASL subroutines written in PL-6
*
        ENTREF  ASL$INIT
        ENTREF  ASL$RECOVERY
        ENTREF  ASL$ROUTINE_X
        ENTREF  ASL$ROUTINE_Y
*
        ENTREF  X66_MSTATIC       and external BMAP routines
*
*       References to external segments
*
        SYMREF  B$PIA$          We will link with B_USRPTRS_D
        SYMREF  B$AUTO$
        SYMREF  B$IS$
*
*         References to the AREADEF structure.
*
        SYMREF  ASL_STATUS$
INITED  EQU     0
BREAK   EQU     1
INTRA   EQU     2
*
*         CP-6 monitor service macros
*
ALTRETFLG   BOOL    400000
INTRTN      EQU     17
XXX         EQU     3
*
M$INTRTN    MACRO
*         This macro generates a monitor service call that returns
*         control to the user domain just as if the normal ASL
*         return were issued.  In addition it causes the operating
*         system to cause a BREAK event to be reported on the
*         user.  Thus the ASL can transfer breaks it receives
*         the the user program when it is ready.
          PMME    17+ALTRETFLG
          ENDM
M$XXX     MACRO
*         This macro generates a standard M$XXX.
          PMME    XXX
          ENDM
*
ASL$ENTRY   NULL
          EAX7    0,0               Copy function code to X7
          TMOZ    BADFCN            If legal codes are all positive
          CMPX7   ASL$FUNCTION_Y    Highest defined function code
          TPL     BADFCN            Out of range
          LDI     =04010            Set HEX and OVRFL mask
*
*         See if this is the initial entry
*
          LDP7    ASL_STATUS$       Pointer to AREADEF structure
          LDA     INITED,,7         Have we been entered before
          TNZ     SETAUTO           Yes
          LDA     1,DL              Indicate have been entered
          STA     INITED,,7         And save
*
*         Check for which type of CLIMB
*
          LDP1    B$PIA$            Get pointer to the PIA
          LDP2    B$IS$             Load Instruction Segment pointer
          INHIB   ON
          STD2    0,,1              Store the descriptor in the PIA
          LDA     0,,1              Get the Working Space Register #
          INHIB   OFF
```

```
         ANA       =0160,DL          Mask out junk
         ARS       4                 Shift into place
         CMPA      6,DL              Is it from inter-domain CLIMB
         TZE       CKAUTO            Yes.  Leave flag as is.
         LDA       1,DL              Non-zero value
         STA       INTRA,,7          Set flag in ASL_STATUS
*
CKAUTO   LDP2      B$AUTO$
         INHIB     ON
         STD2      0,,1              Store AUTO descriptor in the PIA
         LXL1      0,,1              Load the descriptor access flags
         INHIB     OFF
         LXL1      0,,1              Load the descriptor access flags
         ANX1      =0100600,DU       Test Read, Not empty, Present
         ERX1      =0100600,DU
         TNZ       NOAUTO            None yet.  Initialize one.
*        Use existing AUTO segment
         LXL2      0,,2              Offset to top AUTO frame
         SWDX      1,2,2             Compute pointer to top.
         TRA       INIT
*
*      Initialize an AUTO stack
*
NOAUTO   TSX0      X66_MSTATIC
         ZERO      0,0
*
*        Call any ASL initialization routines
*
INIT     TSX1      ASL$INIT
         ZERO      0,0
         TRA       FUNCTION
*
*        Set PR2 to current top of auto stack
*
SETAUTO  LDP2      B$AUTO$           Make sure base pointer set up.
         LXL2      0,,2              Offset to top AUTO frame
         SWDX      1,2,2             Compute pointer to top.
*
*      Now go do the requested function
*
FUNCTION LDX1      CHKBREAK,DU       For PL-6 returns
         TRA       *+1,7
         TRA       BADFCN            0 is not used in this example
         TRA       ASL$RECOVERY      1 is reserved for recovery
         TRA       ASL$ROUTINE_X
         TRA       ASL$ROUTINE_Y
*
*      RET is the common exit point
*
*        It is the ASL's responsibility to save BREAK events received
*        when it is in control and report them to the user.  The
*        ASL must set up an ASYNC routine to intercept breaks, and
*        must set the flag STATUS.BREAK to be non-zero.  This
*        interface will check that flag and report the break event
*        to the user if it is set.
*
CHKBREAK NULL
         INHIB     ON
         LDP7      ASL_STATUS$       Set pointer to status block
         LDA       BREAK,,7          Check for break during ASL
         TZE       RET               None.  Normal return
*
*        Break hit during ASL execution.  Report event to user.
         LDA       0,DL              Clear old flag
         STA       BREAK,,7
```

```
         M$INTRTN
*
RET      NOP
         EXIT
         INHIB    OFF
*
*      Issue M$XXX if the user passes a bad function code
*
BADFCN   M$XXX
         END      ASL$ENTRY
```

Linking an ASL

Once coded an ASL must be linked. The following command illustrates linking of a production ASL:

```
!LINK ;
   B_SSPSL_D.:LIBRARY,;
   ASL$ENTRY,;
   ASL$PL6$SUBROUTINES,;
   B_USRPTRS_D.:LIBRARY,;
   X6U$CSEQU.:LIBRARY ;
      OVER ASL_LIVE (ASLIB,NOSHARELIB)
```

The ASL then needs to be installed in memory. This can be done on a temporary basis using the INSTALL command under the SPIDER processor, and on a permanent basis using TIGR as mentioned above.

The user program is linked using the command:

```
!LINK ;
   USR$MODULE,;
   ASL$BMAP_INTERFACE ;
     OVER USR_PROGRAM(ALTSHARELIB=ASL_LIVE)
```

To link a debug version of the ASL and user program that issues the intra-domain climbs, only one link is necessary as illustrated by the following command:

```
!LINK
  USR$MODULE,;
  ASL$BMAP_INTERFACE,;
  B_SSPSL_D.:LIBRARY,;
  ASL$ENTRY,;
  ASL$PL6$SUBROUTINES,;
  B_USRPTRS_D.:LIBRARY,;
    OVER USR_PROGRAM_DEBUG(NOALT)
```

This link produces a warning that indicates the presence of multiple start addresses for the run unit. This is caused by the fact that both the user's module USR$MODULE and the ASL module ASL$ENTRY are main procedures. This warning can be ignored. Referencing the user's module first in the LINK command causes the run unit to start execution at its main procedure and not the ASL's.

The following RUM can then be applied to change the CLIMB to the the intra-domain type:

```
!RUM USR_PROGRAM_DEBUG
M DEBUG NOP 0
END
```

If the RUM is not applied and the program is executed, an IPR fault occurs because the user program tries to issue an inter-domain CLIMB to an ASL that is not associated with the run unit.

Section 12

Run-Time Libraries

Shared Libraries

A shared library is an executable image of a set of shareable sub-programs which is installed in the :SYS account and is permanently memory resident. A shared library is constructed by LINK in two parts: shareable procedure and static data. The shareable procedure is always biased at octal 700000 or 224K. Since a user's instruction segment is limited to 256K, the procedure portion of a shared library is limited to 32K. The data portion of the shared library is biased at octal 34 and may be as large as is required. When a user's program is linked specifying a shared library (implicitly or explicitly), the library static data space is allocated first in the user's static data space. All other user static data follows the end of the shared library's data space. The total for the procedure and constant data is prohibited from exceeding 224K.

Link Time Association of Shared Libraries

A shared library can be associated at link time by specifying the SHARELIB option which causes the specified library to be associated with the run unit. An alternate method is to build an object unit that has the two fields B$OUHEAD.LNAMSIZ and B$OUHEAD.LNAM filled in appropriately with the desired library name size and text of the name. In this case the linker will automatically associate the specified library from the :SYS account. If multiple library names are specified in the object units making up the program, the linker will associated the library that has the highest precedence. The following is a list of shared libraries in order by precedence (high to low):

:SHARED_COMMON - for programs compiled or interpreted by FORTRAN, APL, or BASIC

:SHARED_COBOL - for programs compiled by COBOL

:SHARED_RPG - for programs compiled by RPG

:SHARED_SYSTEM - for programs compiled by PL6.

Run Time Association of Shared Libraries

When a user's program is linked with a shared library, the user's run unit only contains the space reserved for the library's static data. When the library is associated at run time (by either fetch or a M$ALIB monitor service), the initialized library data will be copied into the space reserved for the library data. At that point, the procedure portion of the shared library will also be mapped in the user's working space. Note that the procedure portion of the library is shared among all users and that each user has a copy of the data portion.

Building Shared Libraries

In order to build a shared library the SLIB option must be specified on the LINK command. The linker will then produce a run unit that represents the shared library. Since the shared library is a self-contained entity, all external references must be resolved when the library is built.

Since the shared library is a self-contained entity, the user may only refer to the externally known entry points in the library. This ensures that when new versions of the library are made, the user's program does not need to be relinked to interface with the new shared library. The ability to create an invariant interface for the user is made possible with the VECTOR option to LINK when the library is built. This option will cause the linker to automatically build a transfer vector at the beginning of the shared library.

The ENTRIES sub-option of the VECTOR option allows the builder of the shared library to specify those entry points that are to be included in the transfer vector built by the linker. Once a transfer vector is built, it must be used as the basis for all libraries built thereafter.

The linker performs this task by changing the external entry point name to refer to a location in the transfer vector. It then maps the entry point name concatenated with an underscore to refer to the original entry point. The following example should illustrate the concept of a transfer vector. If a routine that is to be included in the library starts as follows:

```
        ENTDEF  ABC
ABC     TSX0    X66_AAUTO
```

and the following linker options are specified

```
VECTOR(ENTRIES(ABC))
```

a transfer vector will be built as follows:

```
        ENTDEF ABC
        ENTDEF ABC_
ABC     TRA    ABC_
ABC_    TSX0   X66_AAUTO
```

The RF option informs the linker that the specified rununit file is a shared library and that the transfer vector in it should be used as the starting point in building a new transfer vector. The linker will insure that the virtual address of the external entry points does not change which allows a new library to be associated without requiring the users to relink. Any new entry points will be added to the end of the transfer vector from the previously built library.

The REMOVE SYMDEF and the REMOVE ENTDEF commands cause the linker to build the shared library such that only the legal external entry points (those in the transfer vector) will be available when a user links his program with a shared library. This automatically keeps a user from accidentally referencing either a data location or a procedure entry point that is not necessarily invariant from one version of the library to the next. Using the above example and specifying REMOVE ENTDEF the following represents the resultant library:

```
        ENTDEF  ABC
ABC     TRA     ABC_
ABC_    TSX0    X66_AAUTO
```

Subroutines Included in Shared Libraries

The reason for packaging subroutines into a shared library is twofold: convenience and efficient use of system resources. The convenience can be achieved by packaging the subroutines in a LEMUR library. Thus the principal motive for a shared library is system efficiency. This manifests itself in several ways:

1. Reduced linking time

2. Reduced time to fetch a program for execution

3. Reduced use of memory within a system due to sharing of library procedure.

However, it is important to note that any program which has a shared library associated will have all of the static data of the shared library in it. Thus subroutines to be included in a shared library should minimize the amount of static data used in order to maximize the benefit of system efficiency. This factor should also be considered for purposes of possible use of subroutines asynchronously, e.g., by break routines.

There are several things which cannot be done by subroutines which are to be in a shared library:

1. Explicit references to DCBs other that M$UC and M$DO. Others must be acquired at execution time.

2. SYMREFs or ENTREFs to symbols not contained in the shared library.

3. SYMDEFs of data to be referenced in programs linked with the library.

4. Use of the AREADEF or AREAREF compile-time data segment is not allowed.

Also, special attention should be paid to the considerations for DELTA vis-a-vis shared libraries as specified in Section 15.

User Installation of Shared Libraries

A library may be installed as a shared library temporarily (until system shutdown or crash) via the SPIDER processor. This is done via the INSTALL command of SPIDER, using a type designation of LIB. For further information on the SPIDER processor, see the System Support Reference Manual (CE41).

A library may be installed as a shared library permanently, via the Boot process. This is done by including the library on the labeled portion of the PO tape, via the INCLUDE command of the DEF processor, and by including a MON command, in the TIGR commands of the Boot instructions, specifying the fid of the library and having a flag designation of LI.

Section 13

Library Functions

The term "library services" refers to a set of utility routines which may be used by a PL-6 program to perform common, complex, and/or repetitive tasks in a simple, standardized fashion. Most library services are designed to simplify the job of getting data into a program (prompting the user, reading input, parsing commands, reporting errors, etc.) or getting data out of a program (formatting data, writing object units, etc.). These services thus provide the PL-6 programmer with the same sort of I/O capabilities available to programmers working in other, applications-oriented languages such as FORTRAN, COBOL, or PASCAL. Most library services are used extensively by various CP-6 components (e.g. PCL and EDIT) and/or processors (e.g. FORTRAN). User-written programs which use these library services can be made to resemble familiar CP-6 products, and can take advantage of future enhancements in the service routines.

"Library services" are similar to "monitor services" in many ways, but differ in several fundamental respects. For example:

- Like monitor services, all library services are supported by Honeywell. STARs may be submitted against library services, using the subject name "SHARED_SYSTEM".

- Both library and monitor services are accessed via the PL-6 "CALL" statement. Neither set of services is directly supported by any other CP-6 language.

- Library services are treated as standard PL-6 subroutines. Normally, such services are accessed from within the standard PL-6 run-time library known as :SHARED_SYSTEM; however, a user may wish to actually LINK the service routines into a program.

- Library services can interact with the user's program in complex ways; monitor services cannot. For example, many library service routines accept an optional PL-6 EPTR variable, which holds the ENTADDR of an error-processing routine written by the user; if an error condition is detected by the library service, the error-processing routine will be CALLed to perform any appropriate action.

Library service routines generally have the following characteristics:

- A copy of each service routine is available in the :SHARED_SYSTEM shared library, which is normally associated with each PL-6 program. By using this shared copy, the programmer may make full use of the routine's capabilities without increasing the size of the program.

- A copy of each routine is available in the :LIB_SYSTEM unshared library. This library is automatically searched (and the necessary routine(s) loaded) if a program contains any FORTRAN or COBOL routines (and thus requires use of the :SHARED_COMMON or :SHARED_COBOL libraries).

- A copy of each routine is available in the :LIBRARY account. These unshared copies may be explicitly LINKed into the user's program as necessary (if, for example, it is not desirable to associate or search any system library file).

- Each library service has a name beginning with the letter "X". Some library services are named "X$servicename" (e.g. X$PARSE); others are named "XUx$servicename" (e.g. XUU$READ, XUR$GETCMD).

- Library service routines are designed to be called from a PL-6 procedure. Information is normally passed to/from such routines via PL-6 data structures. These data structures may be generated by copying a PL-6 pre-processor file stored in the :LIBRARY account (by coding an appropriate %INCLUDE statement), and invoking the appropriate %MACRO.

Library service routines can be classified in three general categories: input services, output services, and miscellaneous utility services. For additional information on many of these services, see the CP-6 Monitor Services Reference Manual, Library Services section.

Input Services

Services in this class are used to give the programmer the ability to receive data from an external source (timesharing terminal, disk file(s), batch command stream, etc.) in an easy, standardized fashion.

Table 13-1. Input Library Services

Service	Description
X$PARSE	A general-purpose recursive-descent parser.
XUR$GETCMD	Command processing service to fetch a command from the user, echo it if necessary, parse it, print diagnostics and error messages, give the user HELP information, etc.
XUU$READ	A utility to process source, update, and %INCLUDE records on behalf of a CP-6 compiler.
XSA$FSF	The "fast sequential file" package. This set of routines may be used to read CP-6 keyed and consecutive files in a strictly sequential fashion, at speeds of about twice that of the normal M$READ method.
XUE$EVAL	A general-purpose "expression evaluation" routine, which may be used to evaluate arithmetic expressions within commands, and/or to provide an IBEX-like "pre-processing" capability.

Output Services

Services in this class are designed to provide the PL-6 programmer with a way of easily formatting and writing output from a program, in useful and/or standardized ways.

Table 13-2. Output Library Services

Service	Description
X$FORMAT	A set of routines which provide the programmer with the ability to format and present information in a flexible fashion (not unlike that provided by the FORTRAN formatted-write feature).
X$HELP	A routine which provides a simplified interface to the monitor service M$HELP. Programs which call X$HELP can provide their users with access to all of the features of the CP-6 !HELP command.
XSA$FSF	The "fast sequential file" package may be used to write CP-6 consecutive files, roughly twice as fast as the standard M$WRITE method.
XUO$BUILD	A set of routines which may be used by compilers and other utilities to create a standard CP-6 object unit file, which can then serve as input to the LINK process.

Miscellaneous Utilities

Service routines in this class perform useful functions not related to input/output.

Table 13-3. Miscellaneous Library Services

Service	Description
X$ALLOCATE	A general-purpose routine for managing space within a large block of memory.
XUM$LRU	A routine which manages a "least-recently used" list.
XUW$WILDCARD	A routine to perform general-purpose wildcard pattern matching.

Section 14

Compilers and Language Utilities

Conventions for Language Processors

Conventions specifically for language processors are described in this subsection.

Standard Run Unit Invocation Format for Compilers

Shown following is the standard compiler invocation format of the Honeywell-supplied compilers. All run unit invocation commands are invoked at the IBEX command level with a command of this format.

```
                              [{ON  }                                        ]
                              [{TO  }                                        ]
!rununit [source[, update]][{OVER}[object][, listout]][(optionlist)]
                              [{INTO}                                        ]
```

where

runununit is any valid disk fid. If no account name is specified, special fetch rules apply, as follows. If the file name is specified without a trailing period, the file is fetched from :SYS. If a trailing period does follow the file name, the file is fetched from the user's current file management account.

source is any valid fid. If this fid is omitted, the compiler will process source text entered from the CR device.

update is any valid fid. There is no default.

object is any valid fid. *G file is the default.

listout is any valid fid. The LO device is the default.

optionlist contains rununit specific options, separated by commas.

Source, update, object, and listout are also frequently referred to respectively as fid1, fid2, fid3, and fid4, and are collectively referred to as the positional fids.

This invocation performs the following functions for the invoked run unit:

1. The run unit's designated source DCB is assigned to the source fid if that fid is present on the command line; the flag B$JIT.PRFLAGS.SI is set by IBEX.

2. The run unit's designated update DCB is assigned to the update fid if that fid is present on the command line; the flag B$JIT.PRFLAGS.UI is set by IBEX.

3. Similarly, the run unit's designated object and listout DCBs are assigned to the specific object and listout fids (if present), subject to the implications of the preposition preceding these fids, as follows:

   ON causes IBEX to abort the command if either the object or listout file currently exists.

   OVER directs that the object and listout files are to over-write an existing file, if any. Specifically, the M$OPEN options FUN=CREATE, EXIST=NEWFILE are added to the assignments.

   INTO directs that if the object or listout file exists, it is to be updated; otherwise, new files are to be created. This corresponds to the M$OPEN options FUN=CREATE, EXIST=OLDFILE.

   The flags corresponding to these fields are B$JIT.PRFLAGS.OU and B$JIT.PRFLAGS.LS.

These actions occur prior to entry of the invoked run unit. There is more on this subject in the CP-6 Programmer Reference Manual on LINK, where the LINK options governing DCB associations are described, and in the CP-6 Monitor Services Reference Manual where DCBs and the services connected with them are described in detail.

Consider the example:

```
NEWCOMP A,B ON C,LP (SR(.ALPHA),XR)
```

Upon entry to NEWCOMP, the following assignments have been merged into NEWCOMP's DCBs:

```
M$SI = A (disk file); B$JIT.PRFLAGS.SI='1'B.
M$UI = B (disk file); B$JIT.PRFLAGS.UI='1'B.
M$OU = C (disk file; B$JIT.PRFLAGS.OU='1'B.
M$LO = LP (line printer); B$JIT.PRFLAGS.LS='1'B.
```

Note that the invocation, by its use of positional fids, implicitly specified the following options: UI, OU, LS. These options need not be specified explicitly in the options field of the invocation assuming the the processor examines B$JIT.PRFLAGS. Note that the corresponding flags in the B$JIT are only modified by the invocation command; there is no link between the flags and any particular DCB name.

DCB Usage Conventions

A processor designed to be invoked by its users via a standard invocation may have an interface which allows the user great flexibility in specifying the DCBs through which I/O to the positional fids takes place. If it is not desired to provide this flexibility, the processor's programmer may find it convenient to require the user to employ the standard DCB default associations taken by the LINKer, as described in the LINK section of the CP-6 Programmer Reference Manual.

When a CP-6 processor is LINKed, specific DCBs may be associated with the source, update, object, and listout fields of the Command Processor Invocation. To conform to the standard conventions, these DCBs should be:

- Source DCB = M$SI
- Update DCB = M$UI
- Object DCB = M$OU
- Listout DCB = M$LO

The functions associated with these DCBs should also be common:

M$SI reading of source, base, or command input.

M$UI reading of update input, to be applied to the base input that is read through M$SI.

M$OU writing of the generated object unit resulting from the compilation; M$OU should also be used for workspace files, e.g., in APL or BASIC.

M$LO writing of the generated print file resulting from the compilation.

Other common DCBs and related functions are:

M$SO writing of the merged update and source files to a new file.

M$DO writing of any error or warning messages; all standard processors will have an M$DO DCB.

M$ME reading and writing of information which is to be dispatched to a mode-appropriate device. For instance, EDIT would like its output to appear on a terminal if the user is on-line and on the line printer if the user is in batch mode. This behavior is effected by assigning the special name 'ME' to the RES# field of the DCB.

Compiler Options Usages and Conventions

Options are included in compiler-invoking commands to affect the operation of the compiler. Presumably, writers of processors which provide options similar to those used by the Honeywell supplied compilers will want to use the same terms for these options, or at least will want to avoid using these terms in a manner inconsistent with their use by the Honeywell supplied compilers, and will want to follow similar practices concerning default options. Accordingly, Table 14-1 provides definitions of the options considered standard for new compilers. Note that in Table 14-1 the prefix N indicates NO (i.e., do not perform the option). The explicit use of the N prefix to an option supersedes any implicit assignment of the option.

Table 14-1. Descriptions of Standard Compiler Options

Option	Description
BC({ALL|number[, number] ... })	Specifies the sequential number of each procedure in the source file to be included as a compile unit. ALL requests that all procedures be included.
[M|N]DM[AP][(option[, option] ...)]	Requests that a data map listing of the compilation object unit be written to the device that is associated with the position 4 DCB. Defaults, if no contrary action is taken, are: the position 4 DCB is M$LO, the associated device is the LO device. Options may consist of the following data types: AU[TO], BA[SED], ST[ATIC], SY[MREF]. The prefix M requests a mini-map that consists of the first level of a structure only.
[M|N]PM[AP]	Requests that a procedure map indicating the relative locations of external entry points, local subroutines, and labels be written to the device that is associated with the position 4 DCB. Defaults, if no contrary action is taken are: the position 4 DCB is M$LO, the associated device is the LO device. The prefix M requests that the statement locations be omitted from the map.
[M|N]SC[HEMA]	Specifies that debugging schema records are to be written to the file that is associated with the position 3 DCB. Defaults, if no contrary action is taken, are: the position 3 DCB is M$OU, the file is that named in the third positional fid of the compiler invocation line. Default file name is *G. The prefix M causes schema records to be written only for referenced, external, or SYMDEFed items.

Table 14-1. Descriptions of Standard Compiler Options (cont.)

Option	Description
[M|N]XR[EF]	Requests that a cross-reference listing of the compiled object units be written to the device that is associated with the position 4 DCB. Defaults, if no contrary action is taken, are: the position 4 DCB is M$LO, the associated device is the LO device. This listing contains a dictionary of symbol definitions including all occurrences of all references to the definition. The prefix M generates a cross reference for used references only.
[N]SYS	Specifies that, if an INCLUDE statement or directive is encountered, the :LIBRARY account is to be searched if the file is not found in any of the accounts in the SRCH list. SYS is the default.
[N]LO	Specifies that the symbolic object listing is to be written to the device that is associated with the position 4 DCB. Defaults, if no contrary action is taken, are: the position 4 DCB is M$LO, the associated device is the LO device.
[N]LS	Specifies that all source lines are to be written to the device that is associated with the position 4 DCB. Defaults, if no contrary action is taken, are: the position 4 DCB is M$LO, the associated device is the LO device. If LS is not specified, only source lines with errors are listed.
[N]LU	Specifies that the update file is to be listed. Defaults, if no contrary action is taken, are: the position 4 DCB is M$LO, the associated device is the LO device.
[N]OU	Specifies that an object unit is to be generated, and written to the file that is associated with the position 3 DCB. Defaults, if no contrary action is taken are: the position 3 DCB is M$OU, the file is that named in the third positional fid in the compiler invocation line. Default file name is *G.
[N]SO	Requests that a new source file with updates merged is to be written through the M$SO DCB, to the file associated with that DCB.

Table 14-1. Descriptions of Standard Compiler Options (cont.)

Option	Description
[N]UI	Specifies that update source code is to be read from the file that is associated with the position 2 DCB. Defaults, if no contrary action is taken, are: the position 2 DCB is M$UI, the file is that named in the second positional fid in the compiler invocation line.
[N]UR[EF]	Requests that a list of unused data references be written to the device that is associated with the position 4 DCB. Defaults, if no contrary action is taken, are: the position 4 DCB is M$LO, the associated device is the LO device.
[N]WA[RN]	Requests that all warning messages generated by the compiler be written through M$DO.
SR[CH] (list)	This option augments the specification of the accounts to be searched if a language processor encounters an INCLUDE statement or other directive which specifies a file only by file name. The list is a list of accounts, possibly qualified by packset, separated by commas. Each account designation in the list must have a leading period. The accounts are searched in the order specified by the list. If the file is found in more than one place, the first instance found is the one that is included. A maximum of eight accounts may be supplied in the list. If the file is not found in any of the accounts in the list, the :LIBRARY account and the user's running account will then be searched, in that order. Note that a search of the :LIBRARY does not take place if the NSYS option is specified.

Specifying positional fids in the invocation of a standard compiler will implicitly specify certain options, as follows:

- Specifying the second positional fid (the update fid) constitutes an implicit specification of the UI option.
- Specifying the third positional fid (the object fid) constitutes an implicit specification of the OU option.
- Specifying the fourth positional fid (the listout fid) constitutes an implicit specification of the LS option.

If no option list is specified on the invocation line of a standard compiler, the compiler will take, in addition to the implicit options implied by use of positional fids, the following standard defaults:

LS, OU, BC(ALL), MSCHEMA, NWARN, SYS

If an option list is specified, a standard compiler will supply only three of these defaults, as follows:

- Unless WARN was specified, NWARN will be assumed.
- Unless NSYS was specified, SYS will be assumed.
- Unless BC(list) was specified, BC(ALL) will be assumed.

Suppose that a compiler called NEWCOMP, conforming to these conventions, is in place, and consider the following examples.

```
!NEWCOMP
```

Sets the options LS, OU, BC(ALL), MSCHEMA, SYS, and NWARN (the defaults assumed by the compiler). Source will be read through the M$SI DCB, the default is the CR device.

```
!NEWCOMP A,B ON C,D
```

Sets the options UI, OU, LS, BC(ALL), MSCHEMA, SYS, and NWARN. Note that OU always implies MSCHEMA unless the user specifies otherwise.

```
!NEWCOMP A,B ON C,D (NSCHEMA)
```

Sets the options UI, BC(ALL), SYS, and NWARN.

```
!NEWCOMP A,B (LS)
```

Sets the options UI, LS, BC(ALL), SYS, and NWARN.

Compiler Error Handling

Compilers should behave consistently when dealing with errors in the specification of options; specifically, the compiler should give an appropriate diagnostic and then error the step (issue an M$ERR monitor call). Errors covered by this general rule include specification of an illegal option, repeated options, and inconsistent options.

If the compiler uses a column-flag method of pointing to errors, i.e., placing a special character beneath the offending statement at the position where the error was detected, the following conventions should be observed:

1. The character used should be a caret or up-arrow '^' (octal 136 on the ASCII chart).
2. The offending statement and the flag should both be written through M$DO, unless the statement has already been written to the same print destination as M$DO. This is to ensure that the error flag is not printed out of context.

Object Unit Conventions

The severity level assigned to an object unit by CP-6 compilers should conform to the following values (decimal):

0 no error or warning messages

4 warning messages issued during compilation

7 errors were detected which may be sufficient to cause execution failure

11 fatal error; the object unit contains flaws which will almost certainly prevent proper execution

These values should be stored in the B$OUHEAD.SEVLEV field of the head record of the object unit.

Compiler Output Control Via IBEX

The CP-6 system uses two IBEX-level commands to control generation of compiler output. These commands are:

!COMMENT or !DONT COMMENT

!LIST or !DONT LIST

These commands cause flags in the JIT to be set or reset (!DONT case); the corresponding flags are B$JIT.PRFLAGS.COMMENT and B$JIT.PRFLAGS.LIST. The meaning to compilers should be consistent with the following:

```
COMMENT flag    = '0'B;  skip all writes through the M$DO DCB.

                = '1'B;  perform all writes directed through M$DO DCB.

LIST flag       = '0'B;  skip all M$LO writes

                = '1'B;  perform all M$LO writes.
```

These flags should be checked for every write through the M$DO and M$LO DCBs, since the user may set them at any time. Note that the bits control only the specified DCB; the compiler may choose to skip an M$DO write, regardless of the COMMENT-bit value, when M$DO and M$LO are both assigned to the same thing (assuming the diagnostic is written through M$LO as well as M$DO).

The COMMENT and LIST flags are reset by IBEX when a job step terminates; thus the DONT-case resetting will only hold through the end of the next program/processor invocation.

Source Update Services

The Source Update Package provides input management (XUU) services for language processors which access source input from multiple files. The source update services obtain input from:

- A base source file
- An update file
- One or more files referenced by a "read source library file" directive (e.g., the PL-6 %INCLUDE directive).

For detailed information on these services, refer to the Monitor Services Reference Manual, Library Services section.

Section 15

Interlanguage Calling

This section discusses the format of calling sequences in the CP-6 system. The use of standard calling sequences means that the programs may freely call or be called, regardless of the language in which they are written. Thus programs may be created from routines written in COBOL, FORTRAN, PL-6, RPG-II, PASCAL, or any other language with a compiler that uses the standard calling sequences.

In addition to describing calling sequence conventions for user procedures and subroutines, this section discusses calls for monitor services and the Alternate Shared Library.

The programmer does not normally need detailed knowledge of the calling sequence conventions. However, information in this section is useful in the special circumstances listed below:

- To call the monitor or ASL easily via PL-6 when this capability is not provided by run-time support for a language.
- To write special-purpose subroutines in machine language.
- To call another procedure from machine language.
- To debug by examining in detail a calling or receiving sequence, a monitor call, or an ASL call.

Thus while the dynamics of a subroutine call might suggest that this discussion be presented in the order: calling sequences, receiving sequences, return sequence, a different order is chosen to present the most useful information first. Topics are presented in the following order:

- Receiving sequences
- Return sequences
- UNWIND Routines
- Automatic Storage Layout
- Calling Sequences for External Routines
- Arguments
- Calls for Monitor and the Alternate Shared Library
- Sample Programs

The entry and exit routines described in this section are all included in the file X6U$CSEQU in account :LIBRARY.

The discussion of calling and receiving sequences refers to the registers listed below. For further information on hardware instructions and registers, refer to the DPS8 Assembly Instructions (DH03) manual.

PR0	Pointer Register 0
PR1	Pointer Register 1
PR2	Pointer Register 2 (DR2,AR2)
DR2	Descriptor Register 2
AR2	Address Register
X0-X7	Index Registers 0-7
A	Arithmetic Register
Q	Arithmetic Register
E	Arithmetic Register

Receiving Sequences

The receiving sequences for all PL-6 subroutines will be one of the following forms depending upon the form of storage used and the type of procedure.

Note that the receiving sequences documented here are those that will be used by PL-6, COBOL, and BMAP programs. Other languages, while using the same basic calling sequences, may alter the receiving sequences as required by the requirements of the language.

Table 15-1. Procedure Entry Routines

Procedure Type	Using Auto	Using STATIC (NOAUTO)
MAIN	X66_MAUTO	X66_MSTATIC
ASYNC	X66_AAUTO	X66_ASTATIC
Callable - 0 Args Expected	X66_AUTO_0	X66_STATIC_0
Callable - 1 Args Expected	X66_AUTO_1	X66_STATIC_1
Callable - 2 Args Expected	X66_AUTO_2	X66_STATIC_2
Callable - 3 Args Expected	X66_AUTO_3	X66_STATIC_3
Callable - 4 Args Expected	X66_AUTO_4	X66_STATIC_4
Callable - 5 Args Expected	X66_AUTO_5	X66_STATIC_5
Callable - N Args Expected	X66_AUTO_N	X66_STATIC_N

Each of the procedure entry routines listed in Table 15-1 is entered upon procedure activation with the following sequence:

```
    TSX0    X66_xxx
    ZERO    frameinfo, numargs
```

where:

frameinfo defines the procedure activation frame. If the routine is an AUTO routine, frameinfo is the required size of the AUTO frame rounded up to the nearest even number. If the routine is a NOAUTO routine, frameinfo is the doubleword address in the instruction segment where return address and parameters are to be stored. frameinfo must always be an even value.

numargs is the number of arguments expected by the routine.

Upon return, the required automatic data is allocated, the argument pointers given in this call are moved to the AUTO or NOAUTO frame, and the alternate return address is stored. Note that the value specified for frameinfo must include the words required for the frame header (see below) and the parameter pointers. PR2 is updated to locate the new stack frame.

Registers Used

All of the routines involving Automatic Data assume that Pointer Register 2 (DR2, AR2) frames all of the automatic storage and locates the current external automatic frame. Thus all programs should leave PR2 undisturbed. In addition, DR0, DR1, X0, X1, X2, X3, X4, A , and Q are used by the receiving routines. All other registers are not disturbed.

Note that a subroutine that requires no automatic storage, receives no parameters, and calls no other routines need not call any receiving routine. It may simply not alter X1 and execute TRA 1,X1 for normal return or TRA 0,X1 for alternate return.

Return Sequences

Table 15-2 defines the exit sequences from various types of procedures:

Table 15-2. Procedure Return Routines

Procedure Type	Using AUTO	Using STATIC (NOAUTO)	Registers Used
MAIN RETURN	TRA X66_MARET	TRA X66_MSRET	N/A
ASYNC RETURN	TRA X66_AARET	TRA X66_ASRET	N/A
Callable RETURN	TRA X66_ARET	LDX1 frame TRA 1,X1	Q
Function RETURN	TRA X66_FARET	LDX1 frame TRA 1,X1	X1, X3
MAIN ALTRETURN	TRA X66_MAALT	TRA X66_MSALT	N/A
ASYNC ALTRETURN	TRA X66_AAALT	TRA X66_ASALT	N/A
Callable ALTRETURN	TRA X66_AALT	LDX1 frame TRA 0,X1	Q
Function ALTRETURN	TRA X66_FAALT	LDX1 frame TRA 0,X1	X1, X3

In all cases where TRA is recommended to go to a return sequence, TSX2 is a recommended alternative. This can frequently leave useful information for debugging.

UNWIND Routines

The following table defines the routine names to be entered for the execution of an UNWIND statement for various cases. In all cases A and Q are assumed to be pre-loaded with an Unwind Variable (A has Auto Pointer for frame to be unwound to, Q has EPTR destination).

Table 15-3. Procedure UNWIND Routines

Procedure Type	Using AUTO	Using STATIC (NOAUTO)	Registers Used
MAIN	X66_MAUNWIND	X66_MSUNWIND	N/A
ASYNC	X66_AAUNWIND	X66_ASUNWIND	N/A
Callable	X66_AUNWIND	X66_SUNWIND	N/A

Automatic Storage Layout

The layout of automatic storage which is assumed by the calling/receiving sequences is as follows:

1. DR2 frames all of Automatic Storage.

2. AR2 locates the current external frame which may include several internal frames.

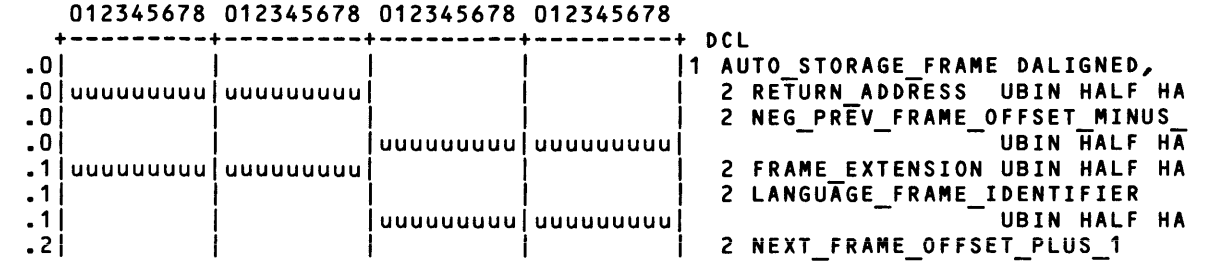
3. The following structures define the required form of each AUTO frame header and of the base of AUTO storage. Also shown is a frame for a typical PL-6 subroutine with internal procedure.

```
   DCL 1 AUTO_STORAGE_FRAME DALIGNED,
/*
         This structure defines the AUTO frame as used by PL-6.
         The first three words of the frame must also be adhered to
         by all users of AUTOmatic storage. Further storage in the
         frame is managed at the discretion of the language.
*/
      2 RETURN_ADDRESS UBIN HALF HALIGNED,
/*
         This field contains the address of the location after the TSX1
         instruction in a CALL statement, i.e. the location where
         an ALTRET will return.  For MAIN and ASYNC procedures
         (no return address), this field contains zero (see loc
         of entry field below).
*/
      2 NEG_PREV_FRAME_OFFSET_MINUS_1 UBIN HALF HALIGNED,
/*
         This field contains the complement of the value (offset of
         previous frame + 1). The field is used to set the AUTO pointer
         when returning from a procedure.
*/
```

```
      2 FRAME_EXTENSION UBIN HALF HALIGNED,
/*
        This field is used to record the original end of an AUTO frame
        when a frame is extended.
*/
      2 LANGUAGE_FRAME_IDENTIFIER UBIN HALF HALIGNED,
/*
        This field is used to identify the language of the procedure
        to which this frame belongs.
        Valid values and their meanings are as follows:

                  -1        PL6
                  -2        FORTRAN
                  -3        RFU
                  -4        RFU
                  .
                  .
                  -16       RFU
                  else      PL/I(field identifies static father - possibly 0)
*/
      2 NEXT_FRAME_OFFSET_PLUS_1 UBIN HALF HALIGNED,
/*
        This field contains the offset to the next frame +1. The +1
        value is stored to avoid overflow when the complement value
        is loaded from the field.
*/
      2 LOC_OF_ENTRY_MAIN_OR_ASYNC UBIN HALF HALIGNED,
/*
        If the return address field contains zero (MAIN or ASYNC)
        then this field will contain the location of the entry
        to the procedure +1.
*/
/*
        The format of the frame through this point is mandatory for
        all frames. The remainder of the frame as described is for
        PL-6 and BMAP procedures.
*/
      2 PARAMS,
/*
        The following words contain PTR's to the arguments passed
        to this procedure. As many as required are reserved.
*/
        3 POINTER1 PTR,
        3 POINTER2 PTR,
        3 POINTER3 PTR,
*/
      2 AUTO_VARIABLES_AND_TEMP(0:3) BIT(36)
/*
        The remainder of an AUTO frame is used for storage of AUTO
        variables and temps generated by the compiler.
*/
;
```

AUTO_STORAGE_FRAME



```
.2|uuuuuuuuu|uuuuuuuuu|         |         |                     UBIN HALF HA
.2|         |         |         |         |  2 LOC_OF_ENTRY_MAIN_OR_ASYNC
.2|         |         |uuuuuuuuu|uuuuuuuuu|                     UBIN HALF HA
.3|         |         |         |         |  2 PARAMS,
.3|ppppppppp|ppppppppp|ppppppppp|ppppppppp|    3 POINTER1       PTR,
.4|ppppppppp|ppppppppp|ppppppppp|ppppppppp|    3 POINTER2       PTR,
.5|ppppppppp|ppppppppp|ppppppppp|ppppppppp|    3 POINTER3       PTR,
.6|         |         |         |         |  2 AUTO_VARIABLES_AND_TEMP(0:3)
.6|bbbbbbbbb|bbbbbbbbb|bbbbbbbbb|bbbbbbbbb|                     BIT(36);
  Z         Z         Z         Z         Z
  +---------+---------+---------+---------+
   012345678 012345678 012345678 012345678   .12-0-0 total length
```

```
   DCL 1 AUTO_STORAGE_BASE DALIGNED,
/*
        This structure defines the base of the automatic storage segment.
        The items contained are used to control AUTO as a whole, and
        to supply values for initializing each frame.
*/
      2 INDETERMINATE UBIN HALF HALIGNED,
      2 NEG_CURR_FRAME_OFFSET_MINUS_1 UBIN HALF HALIGNED,
/*
        This field contains the complement of the value(offset of current
        frame +1). This field is used for initializing the 'previous
        frame' field in the auto frame. It is also used
        to locate the current frame when ASYNC procedures are entered.
*/
      2 MBZ1 UBIN HALF HALIGNED,
/*
        This field is used to initialize the frame extension field.
*/
      2 PL6_FLAG UBIN HALF HALIGNED,
/*
        This field contains the value of the language flag which is proper
        for PL6.
*/
      2 LIMIT_OFFSET UBIN HALF HALIGNED,
/*
        This field contains the value of the last legal offset which
        a frame may reach before further allocation is needed.
*/
      2 MBZ2 UBIN HALF HALIGNED,
      2 MBZ3 UBIN;
```

AUTO_STORAGE_BASE

```
    012345678 012345678 012345678 012345678
   +---------+---------+---------+---------+DCL
 .0|         |         |         |         |1 AUTO_STORAGE_BASE DALIGNED,
 .0|uuuuuuuuu|uuuuuuuuu|         |         |  2 INDETERMINATE   UBIN HALF HAL
 .0|         |         |         |         |  2 NEG_CURR_FRAME_OFFSET_MINUS_1
 .0|         |         |uuuuuuuuu|uuuuuuuuu|                    UBIN HALF HAL
 .1|uuuuuuuuu|uuuuuuuuu|         |         |  2 MBZ1            UBIN HALF HAL
 .1|         |         |uuuuuuuuu|uuuuuuuuu|  2 PL6_FLAG        UBIN HALF HAL
 .2|uuuuuuuuu|uuuuuuuuu|         |         |  2 LIMIT_OFFSET    UBIN HALF HAL
 .2|         |         |uuuuuuuuu|uuuuuuuuu|  2 MBZ2            UBIN HALF HAL
 .3|uuuuuuuuu|uuuuuuuuu|uuuuuuuuu|uuuuuuuuu|  2 MBZ3            UBIN;
   +---------+---------+---------+---------+
    012345678 012345678 012345678 012345678    .4-0-0 total length
```

PL6_FRAME_WITH_INTERNAL

```
     012345678 012345678 012345678 012345678
    +---------+---------+---------+---------+ DCL
  .0|         |         |         |         |1 PL6_FRAME_WITH_INTERNAL DALIGNE
  .0|uuuuuuuuu|uuuuuuuuu|         |         |  2 RETURN_ADDRESS  UBIN HALF HAL
  .0|         |         |         |         |  2 NEG_PREV_FRAME_OFFSET_MINUS_1
  .0|         |         |uuuuuuuuu|uuuuuuuuu|                    UBIN HALF HAL
  .1|uuuuuuuuu|uuuuuuuuu|         |         |  2 FRAME_EXTENSION UBIN HALF HAL
  .1|         |         |         |         |  2 LANGUAGE_FRAME_IDENTIFIER
  .1|         |         |uuuuuuuuu|uuuuuuuuu|                    UBIN HALF HAL
  .2|         |         |         |         |  2 NEXT_FRAME_OFFSET_PLUS_1
  .2|uuuuuuuuu|uuuuuuuuu|         |         |                    UBIN HALF HAL
  .2|         |         |         |         |  2 LOC_OF_ENTRY_MAIN_OR_ASYNC
  .2|         |         |uuuuuuuuu|uuuuuuuuu|                    UBIN HALF HAL
  .3|         |         |         |         |  2 PARAMS,
  .3|ppppppppp|ppppppppp|ppppppppp|ppppppppp|    3 POINTER1      PTR,
  .4|ppppppppp|ppppppppp|ppppppppp|ppppppppp|    3 POINTER2      PTR,
  .5|ppppppppp|ppppppppp|ppppppppp|ppppppppp|    3 POINTER3      PTR,
  .6|         |         |         |         |  2 AUTO_VARIABLES_AND_TEMP(0:3)
  .6|bbbbbbbbb|bbbbbbbbb|bbbbbbbbb|bbbbbbbbb|                    BIT(36),
    Z         Z         Z         Z         Z
 .12|         |         |         |         |  2 INT_FR_A,
 .12|         |         |         |         |    3 RETURN_ADDRESS
 .12|uuuuuuuuu|uuuuuuuuu|.........|.........|                    UBIN HALF,
 .13|         |         |         |         |    3 PARAMS,
 .13|ppppppppp|ppppppppp|ppppppppp|ppppppppp|      4 POINTER1    PTR,
 .14|ppppppppp|ppppppppp|ppppppppp|ppppppppp|      4 POINTER2    PTR,
 .15|ppppppppp|ppppppppp|ppppppppp|ppppppppp|      4 POINTER3    PTR,
 .16|         |         |         |         |    3 AUTO_VARIABLES_AND_TEMP(0:
 .16|bbbbbbbbb|bbbbbbbbb|bbbbbbbbb|bbbbbbbbb|                    BIT(36),
    Z         Z         Z         Z         Z
 .22|         |         |         |         |  2 INT_FR_B,
 .22|         |         |         |         |    3 RETURN_ADDRESS
 .22|uuuuuuuuu|uuuuuuuuu|.........|.........|                    UBIN HALF,
 .23|         |         |         |         |    3 PARAMS,
 .23|ppppppppp|ppppppppp|ppppppppp|ppppppppp|      4 POINTER1    PTR,
 .24|ppppppppp|ppppppppp|ppppppppp|ppppppppp|      4 POINTER2    PTR,
 .25|ppppppppp|ppppppppp|ppppppppp|ppppppppp|      4 POINTER3    PTR,
 .26|         |         |         |         |    3 AUTO_VARIABLES_AND_TEMP(0:
 .26|bbbbbbbbb|bbbbbbbbb|bbbbbbbbb|bbbbbbbbb|                    BIT(36);
    Z         Z         Z         Z         Z
    +---------+---------+---------+---------+
     012345678 012345678 012345678 012345678   .32-0-0 total length
```

The basic structure of automatic storage illustrated applies to all users of automatic storage. Users may apply more structure to the frame beyond the first three words.

A BMAP subroutine accesses the parameters passed to it by first loading the parameter pointers into pointer registers. For example, to load the first parameter, an aligned word, into the Q register, a routine can use the following instructions if an AUTO storage entry routine is used:

```
   LDPn   3,,2  The 3rd word of AUTO is the pointer to the 1st argument
   LDQ    0,,n
```

or the following instructions if a NOAUTO entry routine is used:

```
   LDPn   STADDR+1
   LDQ    0,,n
```

Calling Sequences for External Routines

All calls to external unknown routines must use the following formats. The basic form of the call is as follows:

```
   EPPR0    LOC (pointerlist)              Required if arguments present

   EPPR1    LOC (descriptorlist)           Required

   TSXI     XXX                            XXX is called routine


   {TRA     YYY}                           Control returned here if XXX
   {TSXn    YYY}                              ALTRETURNs
   {           }
   {NOP        }
   .                                       Normal return
   .
   .
```

where:

pointerlist is the list of NSA pointers to actual arguments being passed. This list of pointers is made up as necessary depending on the complexity of the call. When all arguments being passed are in STATIC or CONSTANT storage, the list should be compiled as a block of literals in constant storage. The list must be word aligned.

descriptorlist is described in the following structure:

```
   DCL 1 ARG_DESCRIPTOR_LIST ALIGNED,
/*
         This structure defines the argument descriptor list which
         must accompany each CALL. This list is located by
         PR1 when the CALL is executed. The list should normally
         be compile time constant and thus should be located in
         CONSTANT storage.
*/
      2 NUMBER_OF_ARGS UBIN HALF HALIGNED,
/*
         This field contains the number of arguments being passed.
*/
      2 V BIT(1),
/*
         V=0 specifies that the list is a normal argument list
         as specified here.

         V=1 specifies that the list is non-dense with implied ADDR(NIL)
         for all arguments not passed. This form is not accommodated by the
         setup routines described here.
*/
      2 * BIT(1),
      2 NUM_DESC_WORDS UBIN(16) UNAL,
/*
         This field contains the total number of words in the following
         list not including this word.
*/
      2 DESC_WORDS(0:NUMBER_OF_ARGS-1) ALIGNED,
/*
         The following words define each of the arguments being passed.
*/
        3 DATA_TYPE UBIN HALF HALIGNED,
/*
         The data type of the associated argument. See Table 15-4 for
         valid data types.
*/
```

```
        3 F BIT(1),
/*
        F=0 specifies that the argument is a data item.
        F=1 specifies that the argument is a subroutine or function
        address.
*/
        3 I BIT(1),
/*
        I=0 specifies that the ARG_SIZE field contains the actual size
        of the argument in units appropriate to DATA_TYPE.
        See data type list.
        I=1 specifies that the ARG_SIZE_OFFSET field contains the
        word offset from the beginning of the descriptor extension
        list (DESC_EXT). That location contains further information
        about the specification of the actual size.
*/
        3 A BIT(1),
/*
        A=0 specifies that the argument is being passed as a scalar
        variable.
        A=1 specifies that the argument has an array description
        located by ARG_SIZE_OFFSET.
*/
        3 S BIT(1),
/*
        S=0 specifies that the argument is an elementary data item.
        S=1 specifies that the argument has a structure description
        located by ARG_SIZE_OFFSET.
*/
        3 * BIT(1),
        3 ARG_SIZE UBIN(13) UNAL,
/*
        If I=A=S=0 and the size of the argument is <2**13 units, then
        this field contains the size of the data item. See data type list.
*/
        3 ARG_SIZE_OFFSET REDEF ARG_SIZE UBIN(13) UNAL,
/*
        If I or A or S = 1, then this field contains the offset to
        the descriptor extension word further defining the argument.
        This offset is from the beginning of the descriptor extension
        words. Thus the descriptor extension is located at
        PR1->NUMBER_OF_ARGS+1+ARG_SIZE_OFFSET.
*/
      2 DESC_EXT(0:NUM_DESC_WORDS-NUMBER_OF_ARGS-1) ALIGNED,
/*
        The following words are present only for the exception cases
        specified above.
*/
        3 I BIT(1),
/*
        I=0 specifies that LARGE_SIZE contains the actual size of the
        argument.
        I=1 specifies that the size is located elsewhere as specified
        below.
*/
        3 A BIT(1),
/*
        A=0 specifies that the actual size is contained in the word
        located by STATIC_LOC_OF_SIZE.
        A=1 specifies that the actual size is contained in the word
        in the caller's AUTO frame located by AUTO_OFFSET_OF_SIZE.
        Note that A is ignored if I=0.
*/
        3 * BIT(7),
        3 LARGE_SIZE UBIN(27) UNAL,
/*
        Actual size of the argument.
```

```
*/
        3 STATIC_LOC_OF_SIZE REDEF LARGE_SIZE UBIN(27) UNAL,
/*
         The location within the Instruction Segment which contains
         the actual size.
*/
        3 AUTO_OFFSET_OF_SIZE REDEF LARGE_SIZE UBIN(27) UNAL
/*
         The offset within the caller's automatic frame which contains
         the actual size.
*/
;
```

ARG_DESCRIPTOR_LIST

```
    012345678 012345678 012345678 012345678
   +---------+---------+---------+---------+DCL
 .0|         |         |         |         |1 ARG_DESCRIPTOR_LIST ALIGNED,
 .0|uuuuuuuuu|uuuuuuuuu|         |         |   2 NUMBER_OF_ARGS  UBIN HALF HAL
 .0|         |         |4        |         |   2 V               BIT(1),
 .0|         |         | 2       |         |   2 *               BIT(1),
 .0|         |         |  1421421|421421421|   2 NUM_DESC_WORDS  UBIN(16) UNAL
 .1|         |         |         |         |   2 DESC_WORDS(0:NUMBER_OF_ARGS-1
 .1|uuuuuuuuu|uuuuuuuuu|         |         |     3 DATA_TYPE     UBIN HALF HAL
 .1|         |         |4        |         |     3 F             BIT(1),
 .1|         |         | 2       |         |     3 I             BIT(1),
 .1|         |         |  1      |         |     3 A             BIT(1),
 .1|         |         |    4    |         |     3 S             BIT(1),
 .1|         |         |     2   |         |     3 *             BIT(1),
 .1|         |         |      1421|421421421|     3 ARG_SIZE     UBIN(13) UNAL
 .1|         |         |         |         |     3 ARG_SIZE_OFFSET REDEF ARG_S
 .1|         |         |      1421|421421421|                     UBIN(13) UNAL
 .2|         |         |         |         |   2 DESC_EXT(0:NUM_DESC_WORDS-NUM
 .2|         |         |         |         |                     ALIGNED,
 .2|4        |         |         |         |     3 I             BIT(1),
 .2| 2       |         |         |         |     3 A             BIT(1),
 .2|  1421421|         |         |         |     3 *             BIT(7),
 .2|         |uuuuuuuuu|uuuuuuuuu|uuuuuuuuu|     3 LARGE_SIZE    UBIN(27) UNAL
 .2|         |         |         |         |     3 STATIC_LOC_OF_SIZE REDEF LA
 .2|         |uuuuuuuuu|uuuuuuuuu|uuuuuuuuu|                     UBIN(27) UNAL
 .2|         |         |         |         |     3 AUTO_OFFSET_OF_SIZE REDEF L
 .2|         |uuuuuuuuu|uuuuuuuuu|uuuuuuuuu|                     UBIN(27) UNAL
   +---------+---------+---------+---------+
    012345678 012345678 012345678 012345678   .3-0-0 total length (variable)
```

Table 15-4. Data Types for Arguments

Data Type	Bits/Unit	Description
0	-	Type not specified
1	1	Binary fixed point single (SBIN, INTEGER, COMP-6)
2	1	Binary fixed point double precision
3	1	Binary(Hex exp) float single (REAL)
4	1	Binary(Hex exp) float double (DOUBLE PRECISION)
5	1	Complex binary fixed point single
6	1	Complex binary fixed point double
7	1	Complex binary(hex exp) float single(COMPLEX)
8	1	Complex binary(hex exp) float double(DOUBLE COMPLEX)
9	4+1/2	Packed decimal fixed, lead ASCII sign (COMP-4)
10	4+1/2	Packed decimal float
11	4+1/2	Complex packed decimal fixed,lead ASCII sign
12	4+1/2	Complex packed decimal float
13	36	Pointer (18 word, 2 byte, 4 bit, 12 segid)
14	18	Offset
15	36	Label
16	-	Entry
17	1	Structure (aggregate)
18	-	Area
19	1	Bit string
20	1	Varying bit string
21	9	Character string
22	9	Varying character string
23	-	File
24	1	Unsigned binary fixed point single (UBIN)
25	4+1/2	Packed decimal fixed, trail ASCII sign (COMP,COMP-4)
26	9	Adjustable character string
27	1	Adjustable bit string
28	36	Entry pointer (EPTR)
29	18	16 bit signed integer (2 bytes) (COMP-1)
30	36	32 bit signed integer (4 bytes) (COMP-2)
31	4+1/2	Packed decimal fixed, trail EBCDIC sign (COMP-3)
32	36	INDEX-1
33	36	INDEX-2
34	1	Fortran EVERY
35	1	Fortran LOGICAL
36	-	Fortran ANY (never passed)
37	-	Fortran LABEL
38	1	Fortran UCB
39	1	Intrinsic constant
40	4+1/2	Packed decimal fixed, no sign (COMP,COMP-3,COMP-4)
41	9	Unpacked decimal fixed, no sign
42	9	Unpacked decimal fixed, lead sign
43	9	Unpacked decimal fixed, trail sign
44	9	Unpacked decimal fixed, lead overpunched sign
45	9	Unpacked decimal fixed, trail overpunched sign
46	-	Adjustable structure
47	72	Vector
48	72	Remember
49	72	Descriptor
50-59	-	Reserved
60-63	-	Used in debug schema of object unit

DELTA Interaction with Shared Libraries

When DELTA is tracing the flow of execution which includes calls to shared libraries, the following assumptions are made about TSX instructions which reference the shared library:

- TSX0 is always followed by a non-executable word of information.
- TSX1 is always followed by a instruction to be executed when an alternate return is taken (generally a NOP, TRA, or TSXn).
- TSX2 will not return and is functionally identical to TRA.

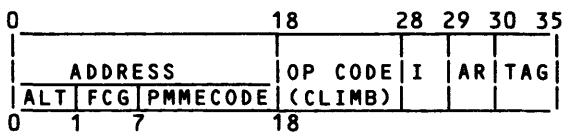
Calls to the Monitor and Alternate Shared Library

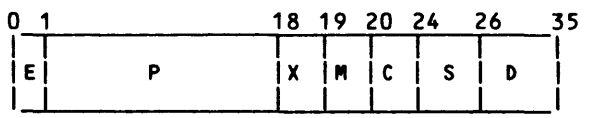
Transfer of control from one domain to another via the CLIMB instruction. The user may call on the monitor by using the PMME form of the CLIMB. He may CLIMB to the Alternate Shared Library directly. This subsection describes the standard calling sequence to be used to enter another domain.

Monitor-User Interface

A monitor service is invoked by the PMME form of the CLIMB instruction. Associated with each montior service is a unique Function Parameter Table (FPT) that supplies the monitor with user-specific information to be used in processing the service request.

The format of this machine instruction is as follows:





ADDRESS Identifies the unique service request and an optional error return:

Bit 0 When set indicates that the instruction following the CLIMB is the alternate return. It is to be executed if the monitor cannot successfully complete the service request. If successfully completed, control is returned skipping this instruction.

When Bit 0 is reset, there is no alternate return address. In this case the monitor aborts the user if the service cannot be completed successfully.

Bit 1-17 Uniquely identify the type of service request. The code defining each monitor service (FCG and PMME code) as well as the structure of each FPT is defined in the INCLUDE file CP_6.

S, D = 1760 Indicates that this instruction is a PMME.

C = 00 Indicates an Inward CLIMB (CALL). Allows for descriptors to be prepared and placed on the Argument Stack. A new Parameter and Argument Segment are framed, and the processor state is saved. Other values of C indicate other types of CLIMB which are not relevant to the present discussion.

X = 1 Specifies loading of the effective address of the instruction into Index Register 0 after the context is pushed into the Safe Store STack.

E - E = 0 Indicates that the service call does not require any parameters. E=1 if parameters are to be passed to the monitor on the Parameter Stack. If E=1, DR0 must contain a descriptor framing the parameters to be passed.

P Specifies the number of parameters minus 1. P is ignored if E=0.

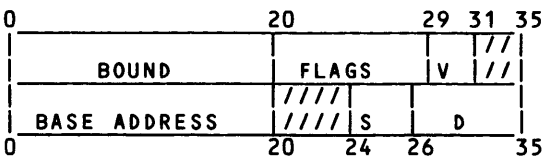
Details concerning the parameters that must be specified are described in the CP-6 Monitor Services Reference manual. The FPT contains all information required for building the monitor's Parameter Stack and for performing the service for the user.

There are two possible sections to an FPT:

1. Vectors framing arguments, expressed as addresses in the user's area. This section of the FPT defines the monitor's Parameter Stack. If the FPT contains value parameters, the first of these vectors frames the data block that contains these values.

2. A data block containing the value parameters.

Each vector in the FPT is of the following format:



V = 01 Indicates a request for a normal shrink of the descriptor defined by S,D.

S, D Identifies the descriptor to be shrunk.

FLAGS Should all be set. The shrunken descriptor will have the same permissions as the descriptor in the Linkage Segment.

BOUND Specifies the byte size minus 1 of the area being passed as a parameter.

BASE ADDRESS Specifies the byte offset into the segment defined by the descriptor that is specified by S,D.

If the hardware is to prepare the monitor's Parameter Stack, PR0 must be loaded with a descriptor prior to executing the PMME CLIMB instruction. PR0 locates a vector list which is used to shrink descriptors to be placed on the stack.

A typical monitor service calling sequence is then:

1. Generate the value parameters as required by the service call.

2. Generate vectors framing areas of memory to build descriptors that the monitor expects to find on the Parameter Stack.
   NOTE: Steps 1 and 2 are frequently accomplished by compiling the correct values.

3. Load DR0 with the descriptor that the hardware may use to build the Parameter Stack.

4. Execute the PMME form of the CLIMB instruction. Follow with an instruction to transfer control to an error routine if the CLIMB had the ALTRET bit set in the address field.

ASL-User Interface

The form for calls to the Alternate Shared Library is identical to that for calling the monitor with two exceptions:

1. There is no provision for alternate return.

2. The S,D field of the CLIMB should contain the value of ASLENTSID. This value will be supplied by the Linker if a program contains a SEGREF ASLENTSID.

Sample Programs

Two sample programs are shown in Figures 15-1 and 15-2 which display code typically generated for standard calling and receiving sequences.

```
                                    1.000  *M* Sample BMAP subroutines showing
                                    1.100  *   use of standard receiving sequences
                                    2.000          ENTDEF    TOV
                                    3.000          ENTDEF    FAD
                                    3.100  *         Add two floating point
                                    3.200  *         numbers, giving a third
                                    3.300  *
                                    4.000  *         First argument is SUM, 2nd
                                    4.100  *         and 3rd are addends
                                    5.000  *
                                    6.000          ENTREF    X66_AUTO_3
                                    6.100  *         Set up args and allocate AUTO
                                    7.000          ENTREF    X66_ARET
                                    7.100  *         Take normal return, free AUTO
                                    8.000          ENTREF    X66_AALT
                                    8.100  *         Take altret, free AUTO
                                    9.000  *
                                   10.000  *         Define locations in AUTO frame
          000000000003              11.000  SUM       EQU       3
          000000000004              12.000  ADDEND1   EQU       4
          000000000005              13.000  ADDEND2   EQU       5
                                   14.000  *
                                   15.000  *
0 000000  X 000000 7000 00         16.000  FAD       TSX0      X66_AUTO_3
                                   16.100  *         Allocate AUTO, set up args
  000001    000006 000003          17.000          ZERO      6,3
                                   17.100  *         Allocate 6 words, set up 3
                                   17.200  *         args.  6 words are for
                                   17.300  *         3 header, 3 args, 0 local
  000002    2 00004 4705 00        19.000          LDP0      ADDEND1,,2
  000003    2 00005 4715 00        20.000          LDP1      ADDEND2,,2
  000004    2 00003 4735 00        21.000          LDP3      SUM,,2
  000005    000001 6140 04         22.000          TEO       1,IC    reset overflow
  000006    000001 6150 04         23.000          TEU       1,IC    and underflow
  000007    0 00000 4311 00        24.000          FLD       0,,0
  000010    1 00000 4751 00        25.000          FAD       0,,1
  000011    3 00000 4551 00        26.000          FST       0,,3
  000012  X 000000 6140 00         27.000          TEO       X66_AALT
                                   27.100  *         Altret if overflow
  000013  X 000000 6150 00         28.000          TEU       X66_AALT
                                   28.100  *         ...or underflow
  000014  X 000000 7100 00         29.000          TRA       X66_ARET
                                   30.000  *
                                   31.000  *
                                   32.000  *
        (0)000000000015            33.000  TOV       EQU       *
                                   33.100  *         This routine requires no auto,
                                   34.000  *         and has no arguments, so it is
                                   34.100  *         simply:
  000015    000000 6170 11         35.000          TOV       0,1       ALTRETURN
  000016    000001 7100 11         36.000          TRA       1,1       RETURN
                                   37.000          END

 CONTROL SECTION TABLE

  0 CODE   EVEN  000017
  1 RODATA EVEN  000000 LITERALS

      8 SYMBOLS

      0 MACROS
```

Figure 15-1. BMAP Program - Standard Receiving Sequences (cont. next page)

```
    SYMBOL SPACE USED     61 WORDS
     MACRO SPACE USED      0 WORDS
   INPUT RECORDS READ    104
STATEMENTS PROCESSED      30
         ELAPSED TIME   0:02.49
             CPU TIME      0.83 SEC.
        ASSEMBLY RATE   2169 STATEMENTS/CPU MIN.

      NO ERRORS
```

Figure 15-1. BMAP Program - Standard Receiving Sequences

```
!PL6 PL6_EG (L0)
PL6 B02  here at 09:52 SEP 27 '83


--  Include file information  --

   CP_6.:LIBRARY  cannot be made into a system file and is referenced.
   B$JIT_C.:B030U  was found in the system file and is never referenced.
   CP_6_C.:B03T0U  was found in the system file and is referenced.

      No diagnostics issued in procedure SAMPLE

   Procedure SAMPLE requires 48 words for executable code.
   Procedure SAMPLE requires 14 words of local(AUT0) storage.


 Object Unit name= SAMPLE                                  File name= *G.
 UTS= SEP 27 '83 09:52:46.89 TUE        Compiler= PL-6/B02 Severity=00
 SharedLib= :SHARED_SYSTEM                                 Alt SharedLib=


    ****  Control sections  ****

 Sect    Type Bound  Init  Size OctSiz  Section name(segment info)
    0    DCB   even  UTS      0      0  M$UC
    1    Data  even  UTS     30     36  SAMPLE
    2    Proc  even  none    48     60  SAMPLE
    3  RoData  even  none     6      6  SAMPLE

    ****  Entry defs  ****

                              Check    Calling
                             calling  sequence
  Sect OctLoc Primary Altret sequence   type   Parms  Name
     2      0    yes    yes      yes      Std        2  SAMPLE

  ****  Entry refs  ****

         Check         Calling
        calling        sequence
Altret sequence SRef    type    Args  Name
 yes      yes           Std        3  SUBR
          yes           Std        3  SUBR1
                        nStd       0  X66_AUT0_2
                        nStd       0  X66_AALT
                        nStd       0  X66_ARET

  ****  Data refs  ****

  Flags:  r = read only, s = secondary
Flgs Name                              Flgs Name                Flgs Name
     M$UC

  ****  Segment refs  ****

  Flags:  r = read only, s = secondary
Flgs Name                              Flgs Name                Flgs Name
     ISSID                                  NULLSID


    1.000         1         /*M*  Sample to show calling sequence features*/
    2.000         2         SAMPLE:  PROC (PAR1,PAR2) ALTRET;

      2  2 000000   000000 700200 xent  SAMPLE        TSX0   ! X66_AUT0_2
      2  2 000001   000016 000002                     ZER0     14,2
```

Figure 15-2. PL-6 Program - Receiving/Calling Sequences (cont. next page)

```
 3.000        3            /*Procedure accepts 2 parameters & may altret*/
 4.000        4     1    DCL PAR1 UBIN;
 5.000        5     1    DCL PAR2 CHAR(8);
 6.000        6                /*Parameters:  all references are through
 7.000        7                  PTRs prepared by setup routines.*/
 8.000        8     1    DCL LCL1 UBIN;
 9.000        9     1    DCL LCL2 CHAR(12);
10.000       10     1    DCL LCL3 UBIN;
11.000       11                /*Local variables referenced directly in
12.000       12                  Instruction Segment or through PR2
13.000       13                  depending on whether procedure is
14.000       14                  compiled NOAUTO or not*/
15.000       15     1    DCL SUBR ENTRY(3) CONV(O) ALTRET;
16.000       16     1    DCL SUBR1 ENTRY(3);
17.000       17                /*Two subroutines may be called, the
18.000       18                  first allowing ALTRETURN and
19.000       19                  requiring all sizes to be specified
20.000       20                  and the second not permitting
21.000       21                  ALTRETURN and not requiring sizes.*/
22.000       22          %INCLUDE CP_6;
23.000       99          %FPT_CLOSE;  /*Declare FPT for monitor call*/
24.000      127
25.000      128
26.000      129     1         LCL1 = PAR1;

 129   2 000002   200003 470500                    LDP0     #PAR1,,AUTO
 129   2 000003   000000 235100                    LDA      0,,PR0
 129   2 000004   200005 755100                    STA      LCL1,,AUTO

27.000      130     1         LCL2 = PAR2;

 130   2 000005   200004 471500                    LDP1     #PAR2,,AUTO
 130   2 000006   040100 100500                    MLR      fill='040'0
 130   2 000007   100000 000010                    ADSC9    0,,PR1 cn=0,n=8
 130   2 000010   200006 000014                    ADSC9    LCL2,,AUTO cn=0,n=12

28.000      131     1         LCL3 = PAR1;

 131   2 000011   000000 235100                    LDA      0,,PR0
 131   2 000012   200011 755100                    STA      LCL3,,AUTO

29.000      132                   /*note re-use of same PTR*/
30.000      133     1         CALL SUBR (LCL1,PAR2,SUBSTR(LCL2,0,LCL1)) ALTRET
            133                   (LBL);

 133   2 000013   200005 236100                       LDQ      LCL1,,AUTO
 133   2 000014   200015 756100                       STQ      LCL3+4,,AUTO
 133   2 000015   200006 633500                       EPPR3    LCL2,,AUTO
 133   2 000016   200014 453500                       STP3     LCL3+3,,AUTO
 133   2 000017   200004 236100                       LDQ      #PAR2,,AUTO
 133   2 000020   200013 756100                       STQ      LCL3+2,,AUTO
 133   2 000021   200005 634500                       EPPR4    LCL1,,AUTO
 133   2 000022   200012 454500                       STP4     LCL3+1,,AUTO
 133   2 000023   200012 630500                       EPPR0    LCL3+1,,AUTO
 133   2 000024   000000 631400 3                     EPPR1    0
 133   2 000025   000000 701000 xent                  TSX1     SUBR
 133   2 000026   000045 702000 2                     TSX2     LBL

31.000      134                  /*Subroutine called, Altreturn accepted
32.000      135                    Argument list includes:
33.000      136                       argument passed in, size implied by
33.100      137                            data type
34.000      138                       argument is local variable, size is
35.000      139                            constant(12)
36.000      140                       argument is local, size computed at
36.100      141                            exeuction*/
37.000      142     1         LCL1 = PAR1;
```

Figure 15-2. PL-6 Program - Receiving/Calling Sequences (cont. next page)

```
    142  2 000027   200003 470500                      LDP0    #PAR1,,AUTO
    142  2 000030   000000 235100                      LDA     0,,PR0
    142  2 000031   200005 755100                      STA     LCL1,,AUTO

   38.000      143                 /*Note PTR requires reloading after CALL*/
   38.100      144    1       CALL M$CLOSE (FPT_CLOSE) ALTRET (LBL);

    144  2 000032   000000 630400 1                    EPPR0   FPT_CLOSE
    144  2 000033   450001 713400                      CLIMB   alt,close
    144  2 000034   406000 401760                      pmme    nvectors=13
    144  2 000035   000045 702000 2                    TSX2    LBL

   38.200      145                 /*Monitor call with altreturn specified*/
   38.300      146    1       LCL1 = PAR1;

    146  2 000036   200003 470500                      LDP0    #PAR1,,AUTO
    146  2 000037   000000 235100                      LDA     0,,PR0
    146  2 000040   200005 755100                      STA     LCL1,,AUTO

   39.000      147    1       CALL M$CLOSE (FPT_CLOSE);

    147  2 000041   000000 630400 1                    EPPR0   FPT_CLOSE
    147  2 000042   050001 713400                      CLIMB   close
    147  2 000043   406000 401760                      pmme    nvectors=13

   40.000      148                 /*Monitor call with no altret specified*/
   41.000      149    1       ALTRETURN;

    149  2 000044   000000 702200 xent                 TSX2  ! X66_AALT

   42.000      150                 /*Take alternate return from SAMPLE*/
   43.000      151    1   LBL: CALL SUBR1 (LCL1,PAR2,SUBSTR(LCL2,0,LCL1));

    151  2 000045   200006 630500       LBL           EPPR0   LCL2,,AUTO
    151  2 000046   200014 450500                      STP0    LCL3+3,,AUTO
    151  2 000047   200004 236100                      LDQ     #PAR2,,AUTO
    151  2 000050   200013 756100                      STQ     LCL3+2,,AUTO
    151  2 000051   200005 631500                      EPPR1   LCL1,,AUTO
    151  2 000052   200012 451500                      STP1    LCL3+1,,AUTO
    151  2 000053   200012 630500                      EPPR0   LCL3+1,,AUTO
    151  2 000054   000005 631400 3                    EPPR1   5
    151  2 000055   000000 701000 xent                 TSX1    SUBR1
    151  2 000056   000000 011000                      NOP     0

   44.000      152              /*Same as call to SUBR above except no altret
   45.000      153                   and no size information required*/
   46.000      154    1       RETURN;

    154  2 000057   000000 702200 xent                 TSX2  ! X66_ARET

FPT_CLOSE
 Sect OctLoc
 1 000000 000003 777640 000032 000000 000000 177640 000000 000000 .....
 1 000004 000000 177640 000000 000000 000000 177640 000000 000000 .....
 1 000010 000000 177640 000000 000000 000000 177640 000000 000000 .....
 1 000014 000000 177640 000000 000000 000000 177640 000000 000000 .....
 1 000020 000000 177640 000000 000000 000000 177640 000000 000000 .....
 1 000024 000000 177640 000000 000000 000000 177640 000000 000000 .....
 1 000030 000000 177640 000000 000000 000000 000000 000000 000040 .....
 1 000034 000220 000000 000000 000000 ........
```

Figure 15-2. PL-6 Program - Receiving/Calling Sequences (cont. next page)

```
(unnamed)
 Sect OctLoc
 3 000000 000003 000004 000030 000044 000025 000010 000032 200000 .......$...
 3 000004 600000 000015 000003 000000 ........
   47.000      155                    /*Take normal return from SAMPLE*/
   48.000      156    1          END;
   49.000      157               %EOD;


--  Include file information  --

   CP_6.:LIBRARY  cannot be made into a system file and is referenced.
   B$JIT_C.:B030U  was found in the system file and is never referenced.
   CP_6_C.:B03T0U  was found in the system file and is referenced.

      No diagnostics issued in procedure SAMPLE

   Procedure SAMPLE requires 48 words for executable code.
   Procedure SAMPLE requires 14 words of local(AUTO) storage.


--  Include file information  --

   CP_6.:LIBRARY  cannot be made into a system file and is referenced.
   B$JIT_C.:B030U  was found in the system file and is never referenced.
   CP_6_C.:B03T0U  was found in the system file and is referenced.

      No diagnostics issued in procedure SAMPLE

   Procedure SAMPLE requires 48 words for executable code.
   Procedure SAMPLE is declared NOAUTO and requires 42 words of local
     (STATIC) storage.


    No errors detected in file PL6_EG.DHEXMPL    .


 Object Unit name= SAMPLE                                    File name= *G.
 UTS= SEP 27 '83 09:53:26.18 TUE              Compiler= PL-6/B02 Severity=00
 SharedLib= :SHARED_SYSTEM                                   Alt SharedLib=


    ****  Control sections  ****

 Sect    Type Bound  Init   Size OctSiz  Section name(segment info)
    0    Data  even  UTS      42     52  SAMPLE
    1    DCB   even  UTS       0      0  M$UC
    2    Proc  even  none     48     60  SAMPLE
    3  RoData  even  none      8     10  SAMPLE

    ****  Entry defs  ****

                              Check   Calling
                             calling  sequence
  Sect OctLoc Primary Altret sequence   type   Parms  Name
     2      0   yes     yes      yes      Std        2  SAMPLE

  ****  Entry refs  ****

         Check         Calling
        calling        sequence
Altret sequence SRef    type    Args  Name
 yes     yes            Std        3  SUBR
         yes            Std        3  SUBR1
                        nStd       0  X66_STATIC_2

  ****  Data refs  ****
```

Figure 15-2. PL-6 Program - Receiving/Calling Sequences (cont. next page)

```
  Flags:  r = read only, s = secondary
Flgs Name                          Flgs Name                Flgs Name
     M$UC

  ****  Segment refs  ****

  Flags:  r = read only, s = secondary
Flgs Name                          Flgs Name                Flgs Name
     ISSID                              NULLSID


   50.000        1        SAMPLE:  PROC (PAR1,PAR2) ALTRET NOAUTO;

      1  2 000000   000000 700200 xent  SAMPLE       TSX0  ! X66_STATIC_2
      1  2 000001   000000 000002 0                  ZERO    0,2

   51.000        2            /*Procedure accepts 2 parameters & may altret*/
   52.000        3    1   DCL PAR1 UBIN;
   53.000        4    1   DCL PAR2 CHAR(8);
   54.000        5                /*Parameters:  all references are through
   55.000        6                  PTRs prepared by setup routines.*/
   56.000        7    1   DCL LCL1 UBIN;
   57.000        8    1   DCL LCL2 CHAR(12);
   58.000        9    1   DCL LCL3 UBIN;
   59.000       10                /*Local variables referenced directly in
   60.000       11                  Instruction Segment or through PR2
   61.000       12                  depending on whether procedure is
   62.000       13                  compiled NOAUTO or not*/
   63.000       14    1   DCL SUBR ENTRY(3) CONV(O) ALTRET;
   64.000       15    1   DCL SUBR1 ENTRY(3);
   65.000       16                /*Two subroutines may be called, the
   66.000       17                  first allowing ALTRETURN and
   67.000       18                  requiring all sizes to be specified
   68.000       19                  and the second not permitting
   69.000       20                  ALTRETURN and not requiring sizes.*/
   70.000       21        %INCLUDE CP_6;
   71.000       98        %FPT_CLOSE;  /*Declare FPT for monitor call*/
   72.000      126
   73.000      127
   74.000      128    1       LCL1 = PAR1;

    128  2 000002   000001 470400 0              LDP0    #PAR1
    128  2 000003   000000 235100               LDA     0,,PR0
    128  2 000004   000003 755000 0              STA     LCL1

   75.000      129    1       LCL2 = PAR2;

    129  2 000005   000002 471400 0              LDP1    #PAR2
    129  2 000006   040000 100500               MLR     fill='040'O
    129  2 000007   100000 000010               ADSC9   0,,PR1      cn=0,n=8
    129  2 000010   000004 000014 0             ADSC9   LCL2       cn=0,n=12

   76.000      130    1       LCL3 = PAR1;

    130  2 000011   000000 235100               LDA     0,,PR0
    130  2 000012   000007 755000 0              STA     LCL3

   77.000      131                /*note re-use of same PTR*/
   78.000      132    1       CALL SUBR (LCL1,PAR2,SUBSTR(LCL2,0,LCL1)) ALTRET
               132               (LBL);
```

Figure 15-2. PL-6 Program - Receiving/Calling Sequences (cont. next page)

```
 132  2 000013   000003 236000 0                    LDQ     LCL1
 132  2 000014   000051 756000 0                    STQ     FPT_CLOSE+33
 132  2 000015   000005 236000 3                    LDQ     5
 132  2 000016   000050 756000 0                    STQ     FPT_CLOSE+32
 132  2 000017   000002 236000 0                    LDQ     #PAR2
 132  2 000020   000006 235000 3                    LDA     6
 132  2 000021   000046 757000 0                    STAQ    FPT_CLOSE+30
 132  2 000022   000046 630400 0                    EPPRO   FPT_CLOSE+30
 132  2 000023   000000 631400 3                    EPPR1   0
 132  2 000024   000000 701000 xent                 TSX1    SUBR
 132  2 000025   000045 702000 2                    TSX2    LBL

79.000      133                 /*Subroutine called, Altreturn accepted
80.000      134                   Argument list includes:
81.000      135                      argument passed in, size implied by
82.000      136                           data type
83.000      137                      argument is local variable, size is
84.000      138                           constant(12)
84.100      139                      argument is local, size computed at
84.200      140                           exeuction*/
85.000      141   1       LCL1 = PAR1;

 141  2 000026   000001 470400 0                    LDPO    #PAR1
 141  2 000027   000000 235100                      LDA     0,,PRO
 141  2 000030   000003 755000 0                    STA     LCL1

86.000      142                 /*Note PTR requires reloading after CALL*/
87.000      143   1       CALL M$CLOSE (FPT_CLOSE) ALTRET(LBL);

 143  2 000031   000010 630400 0                    EPPRO   FPT_CLOSE
 143  2 000032   450001 713400                      CLIMB   alt,close
 143  2 000033   406000 401760                      pmme    nvectors=13
 143  2 000034   000045 702000 2                    TSX2    LBL

88.000      144                 /*Monitor call with altreturn specified*/
88.100      145   1       LCL1 = PAR1;

 145  2 000035   000001 470400 0                    LDPO    #PAR1
 145  2 000036   000000 235100                      LDA     0,,PRO
 145  2 000037   000003 755000 0                    STA     LCL1

88.200      146   1       CALL M$CLOSE (FPT_CLOSE);

 146  2 000040   000010 630400 0                    EPPRO   FPT_CLOSE
 146  2 000041   050001 713400                      CLIMB   close
 146  2 000042   406000 401760                      pmme    nvectors=13

88.300      147                 /*Monitor call with no altret specified*/
89.000      148   1       ALTRETURN;

 148  2 000043   000000 221200 0                    LDX1  ! 0
 148  2 000044   000000 702211                      TSX2  ! 0,X1

90.000      149                 /*Take alternate return from SAMPLE*/
91.000      150   1   LBL: CALL SUBR1 (LCL1,PAR2,SUBSTR(LCL2,0,LCL1));

 150  2 000045   000005 236000 3     LBL            LDQ     5
 150  2 000046   000050 756000 0                    STQ     FPT_CLOSE+32
 150  2 000047   000002 236000 0                    LDQ     #PAR2
 150  2 000050   000006 235000 3                    LDA     6
 150  2 000051   000046 757000 0                    STAQ    FPT_CLOSE+30
 150  2 000052   000046 630400 0                    EPPRO   FPT_CLOSE+30
 150  2 000053   000007 631400 3                    EPPR1   7
 150  2 000054   000000 701000 xent                 TSX1    SUBR1
 150  2 000055   000000 011000                      NOP     0
```

Figure 15-2. PL-6 Program - Receiving/Calling Sequences (cont. next page)

```
   92.000       151                /*Same as call to SUBR above except no altret
   93.000       152                     and no size information required*/
   94.000       153     1        RETURN;

    153   2 000056    000000 221200 0                    LDX1   ! 0
    153   2 000057    000001 702211                      TSX2   ! 1,X1

FPT_CLOSE
 Sect OctLoc
 0 000010 000003 777640 000042 000000 000000 177640 000000 000000 .....".....
 0 000014 000000 177640 000000 000000 000000 177640 000000 000000 ............
 0 000020 000000 177640 000000 000000 000000 177640 000000 000000 ............
 0 000024 000000 177640 000000 000000 000000 177640 000000 000000 ............
 0 000030 000000 177640 000000 000000 000000 177640 000000 000000 ............
 0 000034 000000 177640 000000 000000 000000 177640 000000 000000 ............
 0 000040 000000 177640 000000 000000 000000 000000 000000 000040 ............
 0 000044 000220 000000 000000 000000 ........

(unnamed)
 Sect OctLoc
 3 000000 000003 000004 000030 000044 000025 000010 000032 200000 .......$...
 3 000004 400000 000051 000004 000000 000003 000000 000003 000000 ...).......
   95.000       154                     /*Take normal return from SAMPLE*/
   96.000       155     1        END;


--  Include file information  --

   CP_6.:LIBRARY  cannot be made into a system file and is referenced.
   B$JIT_C.:B030U  was found in the system file and is never referenced.
   CP_6_C.:B03T0U  was found in the system file and is referenced.

      No diagnostics issued in procedure SAMPLE

   Procedure SAMPLE requires 48 words for executable code.
   Procedure SAMPLE is declared NOAUTO and requires 42 words of local
     (STATIC) storage.


    No errors detected in file PL6_EG.DHEXMPL     .
```

Figure 15-2. PL-6 Program - Receiving/Calling Sequences

Appendix A

Job Information Table

A discussion of accessing the Job Information Table (JIT) is included in Section 8 of this manual. This appendix includes descriptions of JIT fields and a depiction of the JIT structure.

JIT Fields

The following fields are contained in the JIT:

ACCESS.

ACCESS. The ACCESS field contains counts of various kinds of physical accesses to devices.

ACCESS.FORMS

ACCESS.FORMS - SBIN. The FORMS field counts the number of physical accesses to resource devices other than disk or tape (e.g. resource Line Printer).

ACCESS.PACKS

ACCESS.PACKS - SBIN. The PACKS field counts the number of physical accesses to files during this job or session. Also included are requested physical accesses which were satisfied by the I/O cache.

ACCESS.TAPES

ACCESS.TAPES - SBIN. The TAPES field counts the number of physical accesses to tapes during this job or session.

ACCN

ACCN - CHAR(8). The ACCouNt field contains the user's log on account.

ARECX

ARECX - UBIN(16). The Accounting RECord indeX field contains the key to be used in the next resource accounting record to be written to *S. This field is incremented each time a resource accounting record is written.

BILL

BILL - CHAR(6). The BILL field is used to locate the user's charge rate record.

BLINDACCTNG

BLINDACCTNG - BIT(1). The BLIND ACCounTiNG field is a flag which, if set, specifies that this user is only to see resources used, and not charge rates or actual monetary units.

BUDLIM

BUDLIM - SBIN. The BUDget LIMit field contains the amount of charges which this job or session is allowed to incur. This field is recorded in hundredths of pennies.

CALCNT

CALCNT - SBIN. The CALCNT field contains the total number of monitor services (PMMEs) executed during this job or session.

CCARS

CCARS - SBIN HALF. The Control Command Actual Record Size field contains the size of the last run unit invocation command.

CCBUF

CCBUF - CHAR(256). The Control Command BUFfer contains the text of the first record of the last run unit invocation.

CCDISP

CCDISP - SBIN HALF. The Control Command DISPlacement field contains the position within the last run unit invocation at which the beginning of options may be found.

CPFLAGS1

CPFLAGS1 - BIT(36). CPFLAGS1 is used by the monitor to communicate with the command processor and is also available for use by the command processor as a word where he may remember user directives.

Bits 0 -> 8 are used by the monitor to communicate job step information to the command processor. These bits are not to be set or reset by the command processor. The meanings of these bit settings are:

CP_LOGOFF# '400000000000'O	When set indicates that the system has detected a line hang-up of a time-sharing terminal or an operator abort of a user and indicates to the command processor that no more job steps are allowed. This bit may be set in conjunction with any of the other CPFLAGS1 bits that are owned by the monitor.
CP_JSTEP# '200000000000'O	Indicates that the user is at Job Step.
CP_RUND# '100000000000'O	Indicates that all levels of exit control processing are completed and the user is about to be rundown.
CP_YC# '040000000000'O	Indicates that the command processor is being entered because the time-sharing user has typed the Control-Y sequence.

CP_YCPMME# '020000000000'0	Indicates that the command processor is being entered because of an M$YC monitor service request.
CP_STTART# '010000000000'0	Used only by the monitor. Indicates that a Start Step Accounting record has been written. This bit is reset when the Stop Step Accounting record is written.

The following bits are also used for monitor / command processor or LOGON / command processor communication:

CP_STARSACC# '000020000000'0	Set by LOGON if the '*S_ACCOUNTING' option of SUPER was specified for this user. When this bit is set, :ACCTLG records will be written to the user's *S file as well.
CP_LASTCPEXISTS# '000010000000'0	Set by LOGON if the 'LAST CPROC' option of SUPER has been specified for this user. Indicates that the default Command Processor is to make its final M$CPEXIT to this user's logoff command processor and not use the delete user form of M$CPEXIT.
CP_LASTCP# '000004000000'0	To be set by the command processor when issuing an M$CPEXIT to the logoff command processor. This allows one final job step after the monitor has set CP_LOGOFF#.
CP_FIRSTCP# '000002000000'0	LOGON sets this bit when exiting to the user's command processor.
CP_STARPROC# '000001000000'0	Set by the monitor whenever a proprietary accounting record is written to the *S file. Refer to M$ACCT in the Monitor Services Reference Manual.
CP_DRIBBLE# '000000100000'0	To be set and reset by the command processor to indicate to the monitor if interactive terminal transactions are to be recorded on the file or device assigned to the M$DRIBBLE DCB.
CP_EXIT# '000000020000'0	Set by the monitor to indicate that the error in JIT.USRERR is not to be reported to the user. The command processor should, however, use the value of JIT.USRERR.SEV as the severity level of the error message last reported to the user.
CP_KEEPDS# '000000010000'0	This bit may be set by the command processor to indicate that the monitor is not to release the command processor dynamic data segments.
CP_PROCACCT# '000000001000'0	Set by the monitor whenever a run unit from the :SYS account that has been LINKed with the PROCACC option is put into execution. When this bit is set, the monitor will cause proprietary start and stop records to be written to the *S file.
CP_STEPACCT# '000000000400'0	Set by LOGON if the 'STEPACCNT' option of SUPER was specified for this user. When set, the monitor will cause job step start and stop records to be written to the *S file.

The remainder of the bits in CPFLAGS1 may be used as seen fit by the command processor. IBEX has predefined the bits for a specific use as follows:

```
   CP_SOMENOTIFY# There is something to NOTIFY user.
'000040000000'0

   CP_SKIPABORT#  Don't abort user at this time
'000000040000'0

   CP_TRMNATE#    Logoff this user after rundown
'000000004000'0

   CP_NOTIFY#     NOTIFY user of changes in BATCH jobs
'000000002000'0

   CP_STEPLMT#    Step limits in effect
'000000000200'0

   CP_PROTECT#    Don't prompt !quit
'000000000100'0

   CP_BUFFULL#    Command in B$JIT.CCBUF
'000000000040'0

   CP_CFREAD#     Read from XEQ file
'000000000020'0

   CP_ECHO#       Echo commands from XEQ
'000000000010'0

   CP_BRK#        Break received
'000000000004'0

   CP_SCREECH#    Prevent multiple snaps when IBEX aborts
'000000000002'0

   CP_DELTA#      !U command found
'000000000001'0
```

CTIME

CTIME - SBIN. The Compensatory TIME field contains the number of microseconds by which this quantum will be shortened due to I/O operations. This value is the number of physical I/O operations done this quantum times the I/O Time Allowance (IOTA) for the user's mode.

CURPNUM

CURPNUM - SBIN HALF. The CURPNUM field is incremented every time a run unit that is the target of an M$LINK or M$LDTRC request is put into execution. This field is decremented when that run unit exits. CURPNUM and HIGHPNUM are used when processor accounting is in effect.

CURRCORE

CURRCORE - UBIN HALF. The CURRent CORE field contains the current number of memory pages chargeable to this user. This value is the result of the following calculation of values from the JIT. CURRCORE = PCD + PCDD + PCDS + PCDDS + PCADS + PCL + PCV + PCROS - 1. PCP is also added unless MMFLGS.FREE_PPGS is set.

CURSUDO

CURSUDO - ARRAY(0:7) UBIN BYTE. The CURrent pSeUDO field contains the current number of each pseudo resource defined which is currently allocated to this job or session. The order is as defined by TIGR.

CURTMPDP

CURTMPDP - SBIN. The CURrent TeMP Disk Pack field contains the current number of granules of temporary disk space allocated.

DCB$

DCB$ - PTR. Contains a pointer to the DCB that was specified on the current monitor service request. If the service request doesn't have a DCB associated with it, this value will be nil. As with JIT.DCBNO, JIT.DCB$ should not be referenced by the user program.

DCBNO

DCBNO - UBIN(9). Contains the number of the DCB that was specified on the current monitor service request. If the service request doesn't have a DCB associated with it, this value will be zero. If an ALTRETURN is made to the user request, this field will be placed in the ALTRET frame along with the value from JIT.ERR. As with JIT.ERR, JIT.DCBNO should not be referenced by the user program.

DDLL

DDLL - UBIN HALF. The Dynamic Data Lower Limit field contains the virtual page number of the first page in the Instruction Segment that may be used for dynamic data.

DDUL

DDUL - UBIN HALF. The Dynamic Data Upper Limit field contains the virtual page number of the last page in the Instruction Segment that may be used for dynamic data.

DEFEXP

DEFEXP - SBIN HALF. The DEFault EXPire field contains the default value for the duration that a file created by this user will be unexpired. This value is used if no expiration date is specified when a file is created.

DEFPRI

DEFPRI - UBIN BYTE. The DEFault PRIority field contains the default batch priority to assign to jobs batched by this job or session.

DLL

DLL - UBIN HALF. This field contains the Data Lower Limit, which is the virtual page number of the first data page of the run unit or standard shared processor currently executing.

DO$

DO$ - PTR. The DO$ field contains a pointer to an active do-list entry used during several file management operations.

DUL

DUL - UBIN HALF. This field contains the Data Upper Limit, which is the virtual page number of the last data page of the run unit or standard shared processor currently executing. If the user is at job step, JIT.DUL will be set to JIT.DLL -1.

ENQS

ENQS - UBIN(18). The ENQS field contains the current number of ENQue resources owned by this job or session.

ERR.

ERR. The ERR field in the JIT always contains the "current" error code reported on this user. This field will be moved to the ALTRET or Stack Frame on the Task Control Block of the domain (user, alternate shared library, debugger or command processor) in control at the time of the error. If this is the error to be reported to the user following all levels of exit control processing, this field will be moved to JIT.USRERR. In any event JIT.ERR is subject to change on any change of domain and should therefore never be referenced by the user program. JIT.ERR is in VLP_ERRCODE format and contains the following subfields:

ERR.CODE

ERR.CODE = DEC(0-16383). This field contains the number that identifies a particular error condition. The file B_ERRORS_C contains a list of the error codes reported by the monitor.

ERR.FCG

ERR.FCG = BIT(12). This field contains the two special six bit characters that identify the functional code group that is reporting the error. Each character is composed of the low-order 6 bits of the ASCII code.

ERR.MID

ERR.MID = BIT(6). This field contains the special six bit character that identifies which module in the functional code group is reporting the error. This character is composed of the low-order 6 bits of the ASCII code.

ERR.MON

ERR.MON = BIT(1). This bit is set if this error is reported by the monitor; reset if reported by a processor.

ERR.SEV

ERR.SEV = DEC(0-7). This field serves a double purpose. Within the monitor it is used to indicate the seriousness of an error. When passed by the user to M$ERRMSG it indicates the level of detail requested in the error message.

EUP

EUP - UBIN HALF. The End User Page field contains the highest virtual page number in the instruction segment which is available for user allocation.

EXTUS

EXTUS - UBIN. The EXecution Time microseconds field contains the number of microseconds (0-9999) of execution time which were not able to be reflected in TPEXT or TUEXT at the end of the previous quantum.

FACCN

FACCN - CHAR(8). The File ACCouNt field contains the users default file account. This field is used as the account for any file reference made when an account is not explicitly specified. This field may be freely be changed by the user, and is not used to determine accessability of files. See M$SETFMA, !DIRECTORY.

FACNACS

FACNACS - BIT(18). The File ACcouNt ACceSs field contains the account permissions for this user vis-a-vis his current file account (FACCN).

FACNCM

FACNCM - UBIN(9). The File ACcouNt Character Match field contains the number of characters of FACCN which matched on a wild compare.

FBUC

FBUC - UBIN HALF. The File Buffer Use Count field contains the total number of file buffers (FPOOLs) currently in use by this user.

FBUL

FBUL - UBIN HALF. The File Buffer Upper Limit field contains the maximum number of file buffers which file management will use on this user's behalf. See !LIMIT FPOOL=n.

FEXT

FEXT - ARRAY(0:35) BIT(1). The File EXTension field contains bits which specify whether ('1'B) or not automatic file extension is currently active on the four command line DCBs as well as a number of DCBs of the form M$xx, where xx is any of a set of CP-6 special names.

FPSN

FPSN - CHAR(6). The File PackSet Name field contains the packset name which is to be used in conjunction with FACCN for default file references.

FRS

FRS - BIT(9). The Final Run Status field contains the accumulated abort flags. This is a logical OR of all the flags that may appear in bits 0 through 4 of RNST as the job step is rundown through all the various levels of exit control. Refer to RNST for the EQUated values and the meanings of the bit settings in this field.

For example, if a user is aborted by the operator, RS_XKEY will appear in RNST until it is put in either the user's exit control frame or USRRNST. At that point the RS_XKEY flag is moved to FRS and RNST is set to zero. Should the user exceed MRT in the exit control routine, RS_LIMX would temporarily appear in RNST, and both RS_XKEY and RS_LIMX would be set in FRS.

GAC

GAC - ARRAY(0:2) UBIN. The Granule ACcounting field contains accounting information for files deleted during this job or session which had been created by this user. Each entry contains a floating point number which is the integral of granules times time. The three entries are for the three charging classes of file granules:

AZ$GACBACKUP	0	Granules of files eligible for backup.
AZ$GACNOBACKUP	1	Granules of files not eligible for backup.
AZ$GACSTOWACT	2	Granules of stowed active files.

HIGHPNUM

HIGHPNUM - SBIN HALF. The HIGHPNUM field is incremented every time a run unit that is the target of an M$LINK or M$LDTRC is put into execution and is reset only on job step termination; thus, this is a count of the number of "fetches" per job step.

HPSN

HPSN - CHAR(6). The Home Pack Set Name field contains the name of the home pack set for this user. The principal use of this field is to determine the pack set on which to create the user's account if it does not already exist.

IDELTAT

IDELTAT - SBIN. The IDELTAT field contains the total quantum time allocated at the last quantum end. This will normally be the quantum specified by the system manager for this mode, partition, user, etc.

INSTWORD

INSTWORD - ARRAY(0:3) UBIN(18). The INSTallation WORD field is a set of four values available for installation use. See M$USRFIELD.

INTER

INTER - SBIN HALF. The INTERactions field contains the number of terminal interactions which have occurred during this time sharing session.

INTTIME

INTTIME - SBIN. The INTeraction COMPute time field contains the time in microseconds expended until the time of the last terminal read. This is used in the calculation of compute time used per interaction. This field is used for time sharing only. See also STATS HISTOGRAM.

JOBNAME

JOBNAME - CHAR(31). The JOB NAME field contains the job name as specified by !JOB NAME=x. Regardless of mode, this job name will be carried with any output symbiont files generated. It may be subsequently used by a user in interrogating the status of output. It will also be displayed on operator displays.

JOBUNIT

JOBUNIT - ARRAY(0:3) UBIN(18). The JOB UNIT field is a set of four counters which are maintained through an entire job and may be used for 'transaction charging'. See M$CHGUNIT and RATES processor.

JPEAK

JPEAK - UBIN HALF. The Job PEAK field contains the maximum value of CURRCORE attained during the job or session.

JRESPEAK

JRESPEAK - UBIN HALF. The Job RESource PEAK field contains the value which must be specified on a !RESOURCE or !ORES command to assure the job of successful execution. This value may differ from JPEAK due to overlay path considerations.

JSLEV

JSLEV - UBIN(3). The Job Statistics LEVel field specifies the type of accounting summary which is to be displayed at the end of job or session. See !OFF. The field can contain the following values:

AZ_ALL#	1	Full Accounting Display
AZ_SUMMARY#	2	One line Accounting Summary
AZ_NONE#	3	No Accounting Display

JTMPDPPK

JTMPDPPK - SBIN. The Job TeMP Disk Pack PeaK field contains the maximum amount of temporary disk storage allocated during this job or session.

JUNK

JUNK - BIT(18). Bits in JIT.JUNK are used by job step processing and help us keep track of what we are doing:

JJ_MLINKIP# '000001'0	Set while processing an M$LINK or M$LDTRC. Reset when run-unit has been fetched or the LINK/LDTRC process is aborted.
JJ_LNKRETIP# '000002'0	Set while the M$LINKing program is being restored.

JJ_RTNXIT# '000100'0 — Set while an M$LINKed to run-unit is in execution.

JJ_SAVING# '010000'0 — Set while a program is being saved for M$SAVE or the SAVE command via IBEX.

JJ_GETTING# '020000'0 — Set while a SAVEd image is being restored.

JJ_NOSAVE# '040000'0 — Set when an M$SCON service request has been issued with SAVEFLG = NO specified. When set, all requests to SAVE the program are ignored. Once set, this bit may only be reset when the job step terminates.

JJ_SCON# '100000'0 — Set when an M$SCON service request has been issued with XCON = YES specified. When set, the user's exit control routine (if any) is entered prior to writing the save image. Reset only on job step termination.

JJ_SCCSET# '000040'0 — Set when an M$EXIT, M$ERR or M$XXX service request with the STEPCC option is specified (from any domain). Reset when the trickle down for exit control reaches the user level, thus allowing the user to reset his STEPCC settings via his final exit from exit control processing.

JJ_LOGOFF# '000200'0 — Set on M$CPEXIT when CP_LASTCP# is set in CPFLAGS1 and the user is at job step. Used to allow entry to a Logoff Command Processor, and to prevent multiple entries to same.

JJ_BYPASSD# '000400'0 — Set on M$DRTN when the Debugger indicates that the user exit control routine is to be entered. Reset when entering the user exit control routine.

JJ_EVENT# '400000'0 — Set on M$DRTN with EVENT = YES specified. Causes the Scheduler to defer events for this user until after the user program has been re-entered. Reset by the scheduler.

JJ_UDELTA# '200000'0 — Set when a run-unit is started under control of a debugger and not set when a debugger is associated via M$ALIB. Control goes to the debugger on all user exceptional conditions (break, traps, etc.) when this bit is set.

JJ_BAKIC# '001000'0 — Set by various monitor service processing routines to indicate that control is to be returned to the user with the IC is Safe-Store reset to the address of the outstanding service request. Used to back out of reads when a break is received, for example.

JJ_RUNXCON# '002000'0 — Set by the exit control logic when the user exit control processing is complete and giving exit control logic for the special shared processors is to be entered. Causes the subcode in the exit control frame for the special shared processor to be set to one.

JJ_DLIB# '004000'0 — Set when a debugger or ASL is disassociated because of an M$DLIB request. Causes the exit control logic to be entered (run-up) for the processor being disassociated and indicates that the user program is to be resumed following exit from the processor's exit control routine.

JJ_EXONLY# '000010'0 — Set when an execute-only run-unit is put into execution. Causes association of a debugger to be disallowed.

JUNK2

JUNK2 - BIT(18). JUNK2 is the overflow of JIT.JUNK. Bits in this word are used as follows:

JJ2_PACCESS# — Set prior to transfer of control to the debugger if the user page table has been modified to allow the debugger to modify user procedure. Will not be set if ALIB. Reset on M$DRTN when write access to procedure is reset.

JJ2_DBRK# — Set on M$DRTN when the DBRK = YES option is specified. Write access to data pages with the SCDRRK bit set in the user's page table will be disabled. This bit will be reset and the write access allowed on those pages when the debugger is next entered.

JJ2_DFRBRK# — Set by the scheduler to defer break control to the user when the break is received during M$YC PMME processing. This bit is reset when the command processor issues the M$CPEXIT, at which time control will be given to the user's break control routine.

LANG

LANG - CHAR(1). The LANGuage field is a single character which specifies the native language of the user. This field is used to select the correct error message and help files for this user.

LBJID

LBJID - UBIN HALF. The Last Batch Job ID field contains the sysid of the last batch job submitted by this job or session.

LLL

LLL - UBIN HALF. The Library Lower Limit field contains the virtual page number of the first page of procedure of an associated run-time library.

LNKCNT

LNKCNT - UBIN(9). Contains the current number of nested M$LINK service requests. This field is incremented on every M$LINK request and decremented each time the linking program is restored.

LOCK

LOCK - ARRAY(0:71) BIT(1). The LOCK is really a "KEY". This double word contains bit settings that allow users to access restricted processors. The LOCK is initialized from the :USERS record when the user enters the system. Refer to the description of the KEY option of SUPER in the System Support Reference Manual and to the description of the SLOCK and WLOCK options of LINK in the Programmer Reference Manual.

LOGONTIME

LOGONTIME - UBIN. The LOGON TIME field contains the time, in UTS units, when this job or session was initiated.

LUL

LUL - UBIN HALF. The Library Upper Limit field contains the virtual page number of the last page of procedure of an associated run-time library.

MAXCORE

MAXCORE - UBIN HALF. The MAXimum CORE field contains the maximum value which CURRCORE will be allowed to reach. The word CORE is retained for nostalgia. See !RESOURCE, !ORES, !LIMIT.

MAXEXP

MAXEXP - SBIN HALF. The MAXimum EXPiration field contains the maximum expiration time that this user may specify.

MAXPRI

MAXPRI - UBIN BYTE. The MAXimum PRIority field contains the maximum batch priority which this job or session may assign to a submitted batch job.

MAXTMPDP

MAXTMPDP - SBIN. The MAXimum TeMP Disk Pack field contains the maximum number of granules of temporary disk space which this job or session is allowed to use.

MMFLGS.

MMFLGS. The Memory Management FLaGS field contains a set of flags which describe the current state of this user from a memory management standpoint.

MMFLGS.FREE_PPGS

MMFLGS.FREE_PPGS - BIT(1). The FREE Procedure PaGeS field indicates whether or not the procedure pages in the currently executing run unit are to be charged to this user. A value of '1'B specifies that the pages are not to be charged.

MODE

MODE - UBIN(4). Specifies the type of user. One of the following EQUed values will be contained in this field:

M_BATCH#	1	Batch User
M_GHOST#	2	Ghost User
M_INT#	3	Interactive User
M_TP#	4	Transaction Processing User

MOUNTS

MOUNTS - ARRAY(0:2) SBIN HALF. The MOUNTS field contains the number of operator mounts required for various resource devices. The entries are used as follows: disk=0, tape=1, other=2.

MRT

MRT - SBIN. The Maximum Run Time field contains the maximum allowed execution plus service time at the beginning of a job and when a !LIMIT command is processed. At any other time it contains the amount of time remaining within the limit. This field is maintained in UTS units.

MSGID.

MSGID. The MeSsaGe IDentification field contains the message id of the last comgroup read done by this job or session. This is used primarily for Transaction Processing.

MSGID.PRIMARY

MSGID.PRIMARY - UBIN. The PRIMARY subfield contains the primary identification of the transaction. This value is the same for all spawned transactions and identifies the parent transaction.

MSGID.XT

MSGID.XT - UBIN. The eXTension field provides a unique identifier for spawned transactions.

NEXTCC

NEXTCC - UBIN(9). The NEXT Control Command field specifies where the next command will be obtained. The possible values are:

CC_FROMNO#	0	There are no more (end of batch job).
CC_FROMJOB#	1	Batch job not in XEQ file.
CC_FROMXEQ#	2	Execute file.
CC_FROMUC#	3	Time sharing terminal.

OLTA

OLTA - ARRAY(0:1) BIT(1). The On Line TApe field contains permission bits which allow time sharing users to use 0-2 tape drives without reserving them via !ORES.

OUTPRIO

OUTPRIO - UBIN(9). The OUTput PRIOrity field contains the priority value to be assigned to output symbiont files generated by this job or session.

PCADS

PCADS - UBIN HALF. The Page Count ASL Data Segments field contains the total number of pages which an ASL has allocated on behalf of this user.

PCC

PCC - UBIN HALF. The Page Count of Context field contains the number of pages the monitor has allocated for this user's context. User context includes HJIT, Page Table, Tstack, JIT, and the first page of the Read Only Segment(ROS).

PCD

PCD - UBIN HALF. The Page Count of Data field contains the number of pages that have been allocated for program data in the instruction segment.

PCDD

PCDD - UBIN HALF. The Page Count Dynamic Data field contains the number of pages which have been allocated dynamically in the Instruction Segment. See MGDP, MGVP.

PCDDS

PCDDS - UBIN HALF. The Page Count Debugger Data Segments field contains the total number of pages which a debugger has allocated on behalf of this user.

PCDS

PCDS - UBIN HALF. The Page Count Dynamic Segments field contains the number of pages that the user has allocated in dynamic data segments. See M$GDS.

PCL

PCL - UBIN HALF. The Page Count of Library field contains zero if no run-time library is associated, or if a shared run-time library is associated. If the run-library becomes unshared, because of an M$DLIB or UNSHARELIB command to DELTA, this field will contain the number of pages that have been obtained for the run-time library procedure.

PCP

PCP - UBIN HALF. The Page Count of Procedure field contains the number of pages that have been allocated for procedure. If a shared processor is in execution, this number will not be included in the PPC field.

PCROS

PCROS - UBIN HALF. The Page Count Read Only Segment is the total number of pages in the Read Only Segment. This segment contains the TCB and DCBs. The first page of this segment is also counted in PCC.

PCV

PCV - UBIN HALF. The Page Count Virtual field contains the total number of real pages allocated in the three virtual segments, as well as the page tables and context necessary to support them.

PLL

PLL - UBIN HALF. This field contains the Procedure Lower Limit, which is the virtual page number of the first procedure page of the run unit or standard shared processor currently executing.

PMEMTIM

PMEMTIM - SBIN. The Processor MEMory TIMe field contains the integral of (TPEXT + TPSVT) * CURRCORE over all quanta of the entire job or session. This is maintained in UTS * page units.

PMME_COUNT

PMME_COUNT - SBIN. The PMME_COUNT and PMME_DATA field are used in conjunction to gather data on PMMEs executed when the MOUSE (Monitor Usage Evaluator) feature of STATS is in use. The PMME_COUNT field contains the current nesting level of a PMME, e.g. an M$ERRMSG PMME must invoke an M$OPEN PMME to open the appropriate error message file.

PMME_DATA

PMME_DATA - ARRAY(0:2). The PMME_DATA field contains various information about starting a PMME for this user. When the PMME completes, the resultant information is recorded in the system MOUSE tables. This field is indexed by PMME_COUNT.

PMME_DATA.CPU

PMME_DATA.CPU - UBIN. The CPU field contains the service time used by this job or session up to the start of this PMME.

PMME_DATA.I_O

PMME_DATA.I_O - UBIN. The I_O field contains the sum of the three ACCESS fields at the start of this PMME.

PMME_DATA.MISC1

PMME_DATA.MISC1 - SBIN. The MISCellaneous1 and MISC2 fields contain various information about the PMME in progress, e.g. for file management operations, the ASN field of the DCB.

PMME_DATA.MISC2

PMME_DATA.MISC2 - SBIN. See MISC1.

PNR

PNR - UBIN(9). The Partition NumbeR field contains, if batch, the batch partition in which this user is running.

PPC

PPC - UBIN HALF. The Physical Page Count field contains the number of real memory pages which are only recorded in the page table of this user. The value in this field may or may not be equal to CURRCORE depending on a variety of factors, such as shared procedure, shared data segments, free procedure, etc.

PPRIV

PPRIV - BIT(36). Contains bit settings indicating which privileged processors may be put into execution by this user. Refer to the description of the PPRIVILEGE option of SUPER in the System Support Reference Manual for a description of the various processor privileges that may be set. Each of the processor privileges have an EQU defined here in the JIT that may be used to test the bit setting in JIT.PPRIV. These are of the form:

```
%EQU PPR_pprivname# = value;
```

where pprivname is the same as that of the option.

PRDPRM

PRDPRM - SBIN. The PeRmanent Disk Pack ReMaining field contains the number of granules of permanent space which this job or session is still allowed to allocate. Note that this is not the same as file account limits.

PRFLAGS.

PRFLAGS. The PRocessor FLAGS field contains a set of flags which are set by the command processor based on the run unit invocation line and other commands.

PRFLAGS.COMMENT

PRFLAGS.COMMENT - BIT(1). The COMMENT field is a flag which is set unless a !DONT COMMENT command was entered immediately prior to this job step.

PRFLAGS.CONTINUED

PRFLAGS.CONTINUED - BIT(1). The CONTINUED field is a flag which is set if a run unit invocation command is continued. The continuation records may be found in the file *CONTINUATION_COMMANDS.

PRFLAGS.LIST

PRFLAGS.LIST - BIT(1). The LIST field is a flag which is set unless a !DONT LIST command was entered immediately prior to this job step.

PRFLAGS.LS

PRFLAGS.LS - BIT(1). The List Source field is a flag which is set if a fid was specified in the fid4 position of the run unit invocation.

PRFLAGS.NSSYNTAX

PRFLAGS.NSSYNTAX - BIT(1). The Non-Standard SYNTAX field is a flag which is set if a run unit invocation command did not conform to the standard syntax. This command will be executed only if the run unit specifies that non-standard syntax is allowed.

PRFLAGS.OU

PRFLAGS.OU - BIT(1). The Object Unit field is a flag which is set if a fid was specified in the fid3 position of the run unit invocation.

PRFLAGS.OUTPUT

PRFLAGS.OUTPUT - BIT(1). Field not currently used.

PRFLAGS.SI

PRFLAGS.SI - BIT(1). The Source Input field is a flag which is set if a fid was specified in the fid1 position of the run unit invocation.

PRFLAGS.UI

PRFLAGS.UI - BIT(1). The Update Input field is a flag which is set if a fid was specified in the fid2 position of the run unit invocation.

PRIINC

PRIINC - REDEF PNR UBIN(9). The PRIority INCrement field contains the execution priority increment to be given to this system ghost over that established for ghosts as a default.

PRIV.

PRIV. There are five words that are defined in the JIT that are used to verify a user's privilege prior to performing certain system functions for this user. A description of the contents of each of these words follows. Within each of the words, the bit settings will correspond to a value for which an EQU statement is included in B$JIT_C. These are of the form:

```
%EQU PR_privname# = value;
```

where privname is the same as that of the sub-option available on the PRIVILEGE option of SUPER. Please refer to the System Support Reference Manual for the names and meaning of these sub-options.

PRIV.ACTIVE

PRIV.ACTIVE - BIT(36). Contains the privileges that are currently in effect. These active privileges are the combination of PRIV.JOB and PRIV.PRC. These privilege bits may also be set and reset by the M$SPRIV and M$RPRIV monitor service request. Refer to the Monitor Services Reference Manual for a description of these requests.

PRIV.AUTH

PRIV.AUTH - BIT(36). Contains the user's privilege indicators as defined, via SUPER, in the :USERS file.

PRIV.JOB

PRIV.JOB - BIT(36). Contains the privileges that have been requested via the !PRIV command of IBEX. The privilege must appear in JIT.AUTH before IBEX will set it in JIT.JOB.

PRIV.PRC

PRIV.PRC - BIT(36). Contains the processor privilege bits, as defined by LINK options, from the :SYS processor's head record. If the currently executing run unit is not from :SYS, PRIV.PRC is set to zero.

PRIV.SAVED

PRIV.SAVED - BIT(36). Contains the value from PRIV.ACTIVE while the associated Command Processor is in control. This value is then restored to PRIV.ACTIVE when the command processor returns control to the user.

PROG_ENTRY

PROG_ENTRY - BIT(9). Set to indicate how the currently executing run-unit was put into execution as follows:

```
PE_CP#      '000'O    Started via M$CPEXIT;
PE_LINK#    '020'O    Started via M$LINK.
PE_LDTRC#   '010'O    Started via M$LDTRC.
```

PSEUDOPGS

PSEUDOPGS - UBIN HALF. The PSEUDO PaGeS field contains the total number of pages which the user is charged for but which are not reflected in PPC.

PSLEV

PSLEV - UBIN(3). The Processor Statistics LEVel field specifies the type of accounting summary which is to be displayed when proprietary processor charging is in effect. See JSLEV for possible values. Also see RATES, CONTROL.

PUL

PUL - UBIN HALF. This field contains the Procedure Upper Limit, which is the virtual page number of the last procedure page of the run unit or standard shared processor currently executing. If the user is at job step, JIT.PUL will be set to JIT.PLL -1.

REMCPO

REMCPO - SBIN. This field contains the number of punched cards allowed remaining for this job.

REMDO

REMDO - SBIN. This field contains the number of pages of DO output remaining for this job.

REMLO

REMLO - SBIN. This field contains the number of pages of LO output remaining for this job.

RERUN

RERUN - BIT(1). The RERUN field specifies, if set, that this batch job is being rerun as a result of some precipitate termination on a previous run.

RESCORE

RESCORE - UBIN WORD. The RESource CORE field contains the current amount of resource memory allocated (not necessarily physically allocated). The MAXCORE field is always less than or equal to RESCORE. See !RESOURCE, !ORES.

RESPEAK

RESPEAK - REDEF JRESPEAK UBIN HALF. See JRESPEAK.

RNST

RNST - BIT(9). The RNST field in the JIT always contains the "current" run status reported on this user. This field will be moved to the exit control frame on the Task Control Block of the domain (user, alternate shared library, debugger or command processor) in control at the time of the exit condition. If this is the status to be reported to the user following all levels of exit control processing, this field will be moved to JIT.USRRNST. JIT.RNST is subject to change on any change of domain and should therefore never be referenced by the user program.

This field contains one of the following EQUed values:

RS_EXIT#	'000'0	M$EXIT was issued.
RS_ERR#	'001'0	M$ERR was issued.
RS_XXX#	'002'0	M$XXX was issued.
RS_SSP#	'004'0	Aborted by Special Shared Processor.
RS_ABRT#	'010'0	Job step aborted by the monitor. This may be because of program trap and no trap control or an errored monitor service request and no ALTRET.
RS_EKEY#	'020'0	Aborted because of operator !E keyin or because on user Control-Y QUIT.
RS_CAN#	'021'0	Batch job has been canceled.
RS_OFF#	'040'0	Logoff by the monitor.
RS_LIMX#	'100'0	Abort because some limit has been exceeded. See description of B$JIT.XLIMFLG.
RS_DROP#	'200'0	Interactive user's line has disconnected.
RS_XKEY#	'400'0	Abort because of !X keyin by the operator.

Note that this field may have more than one bit set on to indicate multiple exit conditions.

RUNFLAGS

RUNFLAGS - BIT(9). Indicates the currently executing process as follows:

RUN_MON#	'001'0	Monitor or Command Processor
RUN_PROC#	'002'0	Processor in :SYS linked with PROCACC
RUN_USER#	'004'0	User program or processor not linked with the PROCACC option

SCHTIME

SCHTIME - SBIN. The SCHedule TIME field contains the sum of XTIME, STIME, and CTIME as they existed the last time this user was scheduled for execution.

SPEAK

SPEAK - UBIN HALF. The Step PEAK field contains the maximum value of CURRCORE attained during the current job step.

SRESPEAK

SRESPEAK - UBIN HALF. The Step RESource PEAK field is the same as JRESPEAK for a job step.

SSLEV

SSLEV - UBIN(3). The Step Statistics LEVel field specifies the type of accounting summary which is to be displayed at each job step. See JSLEV for possible values. Also see !REPORT.

STAR

STAR - ARRAY(0:7). The STAR field contains information about several commonly used star files. The entries are in the order: *T,*G,*L,*A,*S,*N,*X,*I.

STAR.DA

STAR.DA - UBIN. The Disk Address field contains the disk address of the file information field for each of the abovementioned star files.

STDLOPGS

STDLOPGS - SBIN WORD. The STanDard LO PaGeS field contains the number of printed pages of output which have been generated using the form STDLP.

STEPCC

STEPCC - UBIN(9). Contains the Step Condition Code. This field is set from the SEV field of the error code at the time of the exit condition from the user program:

CC_EXIT#	0	M$EXIT
CC_ERR#	4	Job step has been errored.
CC_XXX#	6	Job step has been aborted.

STEPS

STEPS - SBIN HALF. Contains the number of job steps that have been executed since this user logged on.

STEPUNIT

STEPUNIT - ARRAY(0:3) UBIN(18). The STEP UNIT field is a set of four counters which are maintained through a job step and reset to zero at the beginning of each job step. These may be used for 'transaction charging'. See M$CHGUNIT and RATES processor.

STIME

STIME - SBIN. The Service TIME field contains the number of microseconds of service time used this quantum.

STMPDPPK

STMPDPPK - SBIN. The Step TeMP Disk Pack PeaK field contains the maximum amount of temporary disk storage allocated during this job step.

SVLDTF

SVLDTF - BIT(9). The SVLDT field contains bit settings to keep track of various functions (exit control processing and association of DELTA) in the course of M$LINK, M$LDTRC, SAVE and GET processing:

SVL_DIC#	'200'O	DELTA was in control at the time of the SAVE.
SVL_EXIT#	'100'O	M$EXIT from exit control for SAVE - Saved program is now to be run down.
SVL_TRTN#	'040'O	M$TRTN from exit control for SAVE - Saved program is to continue execution.
SVL_LINK#	'020'O	M$LINK in progress.
SVL_LDTRC#	'010'O	M$LDTRC in progress.
SVL_MSAVE#	'004'O	M$SAVE in progress.
SVL_YCSAVE#	'002'O	Control-Y SAVE in progress.
SVL_GET#	'001'O	GET in progress.

SVTUS

SVTUS - UBIN. The SerVice Time microseconds field contains the number of microseconds (0-9999) of service time which were not able to be reflected in TPSVT or TUSVT at the end of the previous quantum.

SWITCH

SWITCH - ARRAY(0:35) BIT(1). The SWITCH field contains a set of 36 pseudo sense switches which may be set/reset by !SWITCH and set/reset by M$SSWITCH/M$RSWITCH.

SYSID

SYSID - UBIN HALF. Specifies the unique System Identification number that has been assigned to this user by the system. This number is reset only on a cold boot or wrap around. All operator communication and external or printed form of user identification will use SYSID.

TDP

TDP - UBIN HALF. The Top Dynamic Page field contains the virtual page numnber of the highest dynamic page currently allocated.

TMPGAC.

TMPGAC. The TeMPorary Granule ACcounting field contains accounting information for temporary files used in this job or session.

TMPGAC.N

TMPGAC.N - UBIN. The TMPGAC.N field contains the integral of temporary granules times time up to the time recorded in TMPGAC.TIME in floating point. This is updated each time a temporary granule is allocated or deallocated.

TMPGAC.TIME

TMPGAC.TIME - UBIN. The TMPGAC.TIME field contains the time in UTS units of the last update of TMPGAC.N.

TPEXT

TPEXT - SBIN. The Total Processor EXecution Time field contains the processor execution time used in this job or session prior to the current quantum. This is maintained in UTS units.

TPSVT

TPSVT - SBIN. The Total Processor SerVice Time field contains the processor service time used in this job or session prior to the current quantum. This is maintained in UTS units.

TSLINE.

TSLINE. The Time Sharing LINE field contains several attributes of a time sharing users terminal connection.

TSLINE.FEX

TSLINE.FEX - UBIN(9). The Front End indeX field contains the FEP number of the FEP to which this user's terminal is connected.

TSLINE.PORT

TSLINE.PORT - UBIN(18). The PORT field identifies the port or MLCP address on the FEP to which this user's terminal is connected.

TSLINE.SPEED

TSLINE.SPEED - UBIN(9). The SPEED field contains a number signifying the line speed of this user's terminal. The current values are:

SPEED	baud	SPEED	baud	SPEED	baud
0	50	1	75	2	110
3	134	4	150	5	200
6	300	7	600	8	1050
9	1200	10	1800	11	2000
12	2400	13	4800	14	9600
15	19200				

Values 0, 1, 3, 8, 10, 11 are not currently supported.

TUEXT

TUEXT - SBIN. The Total User EXecution Time field contains the user execution time used in this job or session prior to the current quantum. This is maintained in UTS units.

TUSVT

TUSVT - SBIN. The Total User SerVice Time field contains the user service time used in this job or session prior to the current quantum. This is maintained in UTS units.

UMEMTIM

UMEMTIM - SBIN. The User MEMory TIMe field contains the integral of (TUEXT + TUSVT) * CURRCORE OVER all quanta of the entire job or session. This is maintained in UTS * page units.

UNAME

UNAME - CHAR(12). The User NAME field contains the user's log on name.

USER

USER - UBIN(9). Specifies the User Number that has been assigned to this user by the system. This number is used internally as an index into the system user tables (B$USER) and is not used for any external user identification.

USERWORD

USERWORD - ARRAY(0:3) UBIN(18). The USER WORD field is a set of four values available for user use. See M$USRFIELD.

USRDCB

USRDCB - UBIN(9). Contains the value from DCBNO at the time of the exit condition from the user program. Thus, if there is a DCB# associated with the error code in JIT.USRERR it will be here.

USRERR.

USRERR. Contains the error code that reflects the exit condition of the user program. JIT.USRERR is in VLP_ERRCODE format; refer to ERR for an explanation of the FCG, MID, MON, CODE and SEV subfields. USRERR only reflects the exit condition of the user domain.

USRIC

USRIC - UBIN(18). Contains the Instruction Counter at the time of the exit condition from the user program.

USRRNST

USRRNST - BIT(9). Contains the run status to reflect the exit condition of the user program. Refer to RNST for an explanation of the bit settings in this field. Note that USRRNST only applies to the exit condition of the user domain. For example, if the job step is terminated because a special shared processor aborts, RS_ABRT# would have been set JIT.RNST for that special shared processor's error processing, but JIT.USRRNST would have RS_ABORT# reset and RS_SSP# would be set instead.

UTIMER

UTIMER - UBIN. The User TIMER field contains the time, in microseconds, remaining before expiration of the timer which the user established with M$STIMER. A value of zero means there is no timer currently established.

VIRTUAL.

VIRTUAL. The VIRTUAL field is a structure defining currently open virtual data segment files.

VIRTUAL.DCB#

VIRTUAL.DCB# - ARRAY(0:2) UBIN(9). The DCB# field contains the DCB number of the DCB open to the file which defines each of the virtual data segments.

WOO

WOO - CHAR(8). The Workstation Of Origin field contains the name of the workstation from which this batch job originated. If the WSN option is specified on a !JOB record, that becomes the WOO for the job. Timesharing, ghost, and Transaction Processing jobs get their default WOO from the authorization file. Regardless of the source of WOO, it is used as the default workstation for all unit record output if WSN is not specified. It can also be used for banners.

XCONF

XCONF - BIT(9). Bits 6 through 8 of the XCONF field are used by the monitor to keep track of the exit control activity as control passes from one domain to another:

XC_URND	'004'0	Set when user level exit control processing is completed, or if the user has no request for exit control.
XC_ASL#	'002'0	Set when ASL exit control processing is complete.
XC_QUIT	'001'0	Set on a QUIT command to DELTA or IBEX.

Bits 0 through 7 are used to record the prior RNST for multiple entries to exit control within any domain. Refer to RNST for the EQUated values and meanings of these bits.

XLIMFLG

XLIMFLG - BIT(9). When RS_LIMX# is set in RNST, one of the following bits will be set in XLIMFLG to indicate what limit has been exceeded:

XL_PO#	'400'0	Cards Punched
XL_MEM#	'200'0	Memory -only causes abort if restoring a an M$LINK or SAVE program image.
XL_LO#	'100'0	M$LO pages
XL_DO#	'040'0	M$DO pages
XL_STACK#	'002'0	Safe-store Stack or Argument Stack
XL_TIME#	'001'0	Maximum Run Time exceeded

XTIME

XTIME - SBIN. The eXecution TIME field contains the number of microseconds of execution time used this quantum.

YCOSZ

YCOSZ - UBIN(18). Contains the bound of the M$YC monitor service request CMD parameter. This field has meaning only when the CP_YCPMME# bit is set in CPFLAGS1.

Structure Format

Each of the structure drawings in this appendix represents one %MACRO entry which defines a structure. Comments and most preprocessor constructs have been removed from the descriptions to make them easier to read. There are one, two, or three sections in each drawing, depending on the data contained in the text of the structure and whether the structure is a %MACRO or DCL.

The first section of each drawing contains an alphabetized list of all the parameters specified in the %MACRO header and their default values. If substitution keywords are also present, they are listed alphabetically under the parameter to which they apply, along with their values. Most parameters are used in the INIT clause of the DCL; in this case the (first) location in the structure in which the parameter value will be found is printed opposite the parameter name. The format used is ".octal word-byte-bit". If the parameter is not referenced in an INIT clause, the location is replaced by ".....". Only one value per INIT clause may be cross-referenced in this manner; if multiple values are present in the INIT clause, the second and succeeding ones will be flagged as "not found" (".....") in the parameter listing.

The second section contains a list of %EQUs or %SUBs found embedded in the DCL source. The fully-qualified name to which they apply appears as a header line prior to the list of source text.

The third section is the actual text of the DCL and a drawing of the structure it generates. The right half of the page contains the structure as it appears in the source file, except that comments and preprocessor expressions have been removed. The left half of each line shows the memory layout generated by the code to its right on the same line. The drawing is one word (36 bits) wide and continues for as many words as necessary to describe the entire structure. The words are divided into four nine-bit bytes to make them easier to read. Lower case letters are used to represent the bits occupied by the item being described:

    bbbbb - indicates a BIT item.
    ccccc - indicates a CHAR item.
    eeeee - indicates an EPTR item.
    ppppp - indicates a PTR item.
    sssss - indicates an SBIN item.
    uuuuu - indicates a UBIN item.

If the data item either begins or ends on a non-byte boundary, its representation in the drawing is changed to a string of "421" sequences, indicating the value of the bit within the octal nibble of the byte. This assists in decoding fields which are non-byte aligned when only an octal representation is available.

Filler or supplementary storage which will be present in the structure is indicated by a sequence of one or more periods (".").

If a data item generates six or more words of storage, the "middle" words will not be printed to conserve space. This omission is indicated by a line of "Z" characters replacing the vertical bars ("|") normally used to separate bytes.

Only one occurrence of an array element is shown, to illustrate its shape. Blank spaces and/or "Z" lines represent the remaining space occupied by the array.

Items which are variable-length are shown as one bit (or byte) wide; the user must use the actual values used in the structure to determine the true length of the item at execution time. Similarly, arrays with variable upper bounds are shown with only one element.

Each word in the drawing is preceded by its octal offset from the beginning of the structure. The total length of the structure is indicated below the last line of the DCL in the form ".octal words-bytes-bits". If the structure contains variable-length strings or variable-dimension arrays, it is flagged as "variable".

The following sample structure illustrates many of these points. Numbers in parentheses refer to the items indicated by the <bracketed> numbers on the example.

(1) The %MACRO or DCL name is printed at the left margin at the beginning of the structure drawing and on each continuation page.

(2) This is a listing of the parameters in alphabetic order in a two-column format.

(3) Indicates that the storage for "TYPE" begins at word 3 (octal), byte 0, bit 0.

(4) This is the parameter ("TYPE") and its default value ("0").

(5) These are the keywords which may be specified to change the value of the parameter (e.g., "TYPE=TERMINAL" places "3" in the TYPE field of the structure) and the values they represent.

(6) The periods (".") indicate that this parameter was not found in an INIT clause or is part of a list of initial values. In this case, the latter is true.

(7) This is the %EQU and %SUB section.

(8) This is the fully-qualified name to which these EQUs apply. Note that "FPTN" is a parameter which will be changed when this macro is invoked.

(9) These are the names which may be used when comparing FPTN.V.TYPE# to a value.

(10) The remainder of the entry is the drawing section.

(11) The right half of this section contains the DCL text.

(12) The left half of this section contains the drawing of the storage generated by the DCL text.

(13) The location for this line is indicated in octal. Since there is nothing else in the drawing, this line of the DCL generates no storage.

(14) This line of the DCL has generated a 72-bit (two word) bit item, represented by the b's. OTher items generate e's, u's, and so forth, depending on their type.

(15) This is an array element. Note that the two bytes following it in word .4 are blank, indicating that the array occupies this space.

(16) Although DIRECTION# is a UBIN value, its storage is represented by the two bits "42" since it does not end on a byte boundary. The "42" indicates that DIRECTION# occupies the leftmost two bits of the first octal nibble of this byte.

(17) Since HEADER# generates 20 words of storage, it is shortened to its first two words and last two words. The "Z" line indicates that one or more words have been omitted to conserve space.

(18) Since C# is a variable-length string, only one byte is indicated as its length in the drawing. The user would consult the L# byte to determine the true length of C# at run time.

(19) Periods indicate supplementary or filler storage. Three bytes of supplementary storage are added to L# since it is (by default) ALIGNED. Up to seven bytes of filler storage are added to the structure after the last character of C# since it is DALIGNED.

(20) The total length of the structure is octal .34 words, 0 bytes, and 0 bits; however, since this structure contains a variable-length item, the user must adjust the total length appropriately at run time.

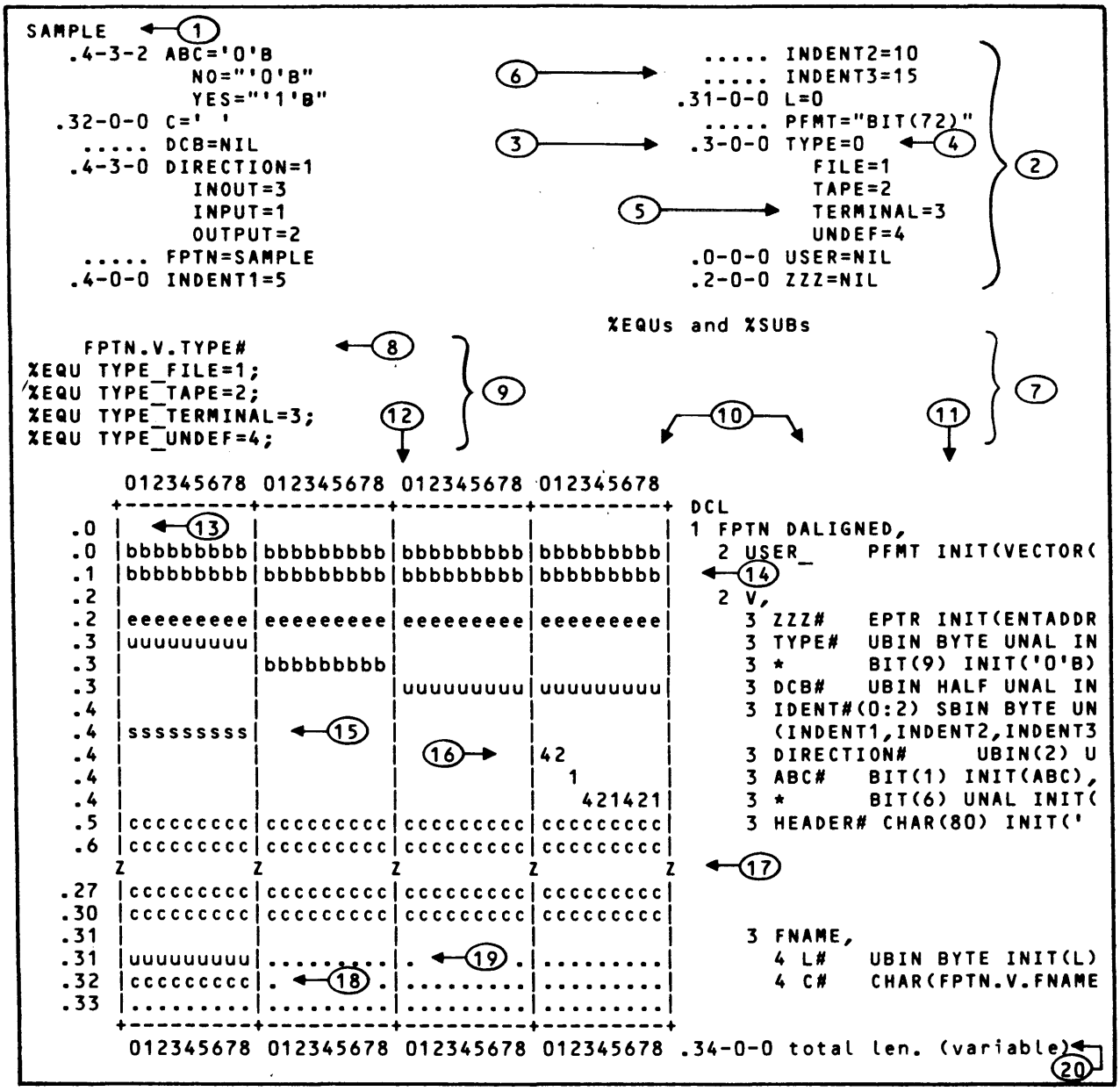


Figure A-1. Sample Structure

JIT Structure

The Job Information Table is illustrated on the following pages. The JIT structure is available in B$JIT_C.:LIBRARY.

```
..... B$JIT=B$JIT                                    ..... STCLASS="BASED(B$JIT$)"

                                                     %EQUs and %SUBs

--> B$JIT.MODE <--
%EQU M_INT#=3;
%EQU M_BATCH#=1;
%EQU M_GHOST#=2;
%EQU M_TP#=4;

--> B$JIT.PRIV.ACTIVE <--
%EQU PR_SPCLMM#='000000000001'0;
%EQU PR_EXMM#='000000000002'0;
%EQU PR_MAXMEM#='000000000004'0;
%EQU PR_MSYS#='000000000010'0;
%EQU PR_JIT#='000000000040'0;
%EQU PR_TND#='000000000100'0;
%EQU PR_PM#='000000000200'0;
%EQU PR_EXPM#='000000000400'0;
%EQU PR_IOQ#='000000001000'0;
%EQU PR_IOQW#='000000002000'0;
%EQU PR_CFEP#='000000020000'0;
%EQU PR_MFEP#='000000040000'0;
%EQU PR_SYSLOG#='000000100000'0;
%EQU PR_GPP#='000000400000'0;
%EQU PR_ASAVE#='000001000000'0;
%EQU PR_SYSCON#='000002000000'0;
%EQU PR_DISPJOB#='000010000000'0;
%EQU PR_FMEFT#='400000000000'0;
%EQU PR_FMBLK#='200000000000'0;
%EQU PR_FMSEC#='100000000000'0;
%EQU PR_FMDIAG#='040000000000'0;
%EQU PR_FMREAD#='020000000000'0;

--> B$JIT.PPRIV <--
%EQU PPR_CNTRLD#='400000000000'0;
%EQU PPR_CNTRLC#='200000000000'0;
%EQU PPR_EFT#='100000000000'0;
%EQU PPR_EL#='040000000000'0;
%EQU PPR_LABEL#='020000000000'0;
%EQU PPR_PIGD#='010000000000'0;
%EQU PPR_PIGC#='004000000000'0;
%EQU PPR_SPIDERD#='002000000000'0;
%EQU PPR_SPIDERC#='001000000000'0;
%EQU PPR_SUPER#='000400000000'0;
%EQU PPR_FEPANLZ#='000200000000'0;
%EQU PPR_SUPERAUTH#='000100000000'0;
%EQU PPR_SUPERWSN#='000040000000'0;
%EQU PPR_SUPERFORM#='000020000000'0;
%EQU PPR_PADMIN#='000010000000'0;
%EQU PPR_SUPERD#='000004000000'0;
%EQU PPR_VOLINIT#='000002000000'0;
```

Figure A-2. B$JIT0 Structure (cont. next page)

```
%EQU PPR_REPLAY#='000001000000'0;
%EQU PPR_RATES#='000000400000'0;

--> B$JIT.CPFLAGS1 <--
%EQU CP_DELTA#='000000000001'0;
%EQU CP_SLEAZE#='000000000001'0;
%EQU CP_SCREECH#='000000000002'0;
%EQU CP_BRK#='000000000004'0;
%EQU CP_ECHO#='000000000010'0;
%EQU CP_CFREAD#='000000000020'0;
%EQU CP_BUFFULL#='000000000040'0;
%EQU CP_PROTECT#='000000000100'0;
%EQU CP_STEPLMT#='000000000200'0;
%EQU CP_STEPACCT#='000000000400'0;
%EQU CP_PROCACCT#='000000001000'0;
%EQU CP_NOTIFY#='000000002000'0;
%EQU CP_TRMNATE#='000000004000'0;
%EQU CP_KEEPDS#='000000010000'0;
%EQU CP_EXIT#='000000020000'0;
%EQU CP_SKIPABORT#='000000040000'0;
%EQU CP_DRIBBLE#='000000100000'0;
%EQU CP_INITIALIZE#='000000200000'0;
%EQU CP_TESTMODE#='000000400000'0;
%EQU CP_STARPROC#='000001000000'0;
%EQU CP_FIRSTCP#='000002000000'0;
%EQU CP_LASTCP#='000004000000'0;
%EQU CP_LASTCPEXISTS#='000010000000'0;
%EQU CP_STARSACC#='000020000000'0;
%EQU CP_SOMENOTIFY#='000040000000'0;
%EQU CP_SSTART#='010000000000'0;
%EQU CP_YCPMME#='020000000000'0;
%EQU CP_YC#='040000000000'0;
%EQU CP_RUND#='100000000000'0;
%EQU CP_JSTEP#='200000000000'0;
%EQU CP_LOGOFF#='400000000000'0;
%EQU CP_RESET#='417777777777'0;
%EQU CP_RSSTART#='407777777777'0;

--> B$JIT.SSLEV <--
%EQU AZ_ALL#=1;
%EQU AZ_SUMMARY#=2;
%EQU AZ_NONE#=3;

--> B$JIT.NEXTCC <--
%EQU CC_FROMNO#=0;
%EQU CC_FROMJOB#=1;
%EQU CC_FROMXEQ#=2;
%EQU CC_FROMUC#=3;
%EQU CC_FROMXEQEND#=4;

--> B$JIT.PROG_ENTRY <--
%EQU PE_CP#='000'0;
```

Figure A-2. B$JIT0 Structure (cont. next page)

```
%EQU PE_LINK#='020'0;
%EQU PE_LDTRC#='010'0;

--> B$JIT.RNST <--
%EQU RS_EXIT#='000'0;
%EQU RS_ERR#='001'0;
%EQU RS_XXX#='002'0;
%EQU RS_SSP#='004'0;
%EQU RS_ABRT#='010'0;
%EQU RS_EKEY#='020'0;
%EQU RS_OFF#='040'0;
%EQU RS_LIMX#='100'0;
%EQU RS_DROP#='200'0;
%EQU RS_XKEY#='400'0;
%EQU RS_CAN#='021'0;
%EQU RS_PMME#='774'0;
%EQU RS_CL23#='740'0;
%EQU RS_CL3#='640'0;
%EQU RS_XCON#='760'0;

--> B$JIT.RUNFLAGS <--
%EQU RUN_MON#='001'0;
%EQU RUN_PROC#='002'0;
%EQU RUN_USER#='004'0;

--> B$JIT.JUNK <--
%EQU JJ_MLINKIP#='000001'0;
%EQU JJ_LNKRETIP#='000002'0;
%EQU JJ_AMERGE#='000004'0;
%EQU JJ_EXONLY#='000010'0;
%EQU JJ_ENQBIT#='000020'0;
%EQU JJ_SCCSET#='000040'0;
%EQU JJ_RTNXIT#='000100'0;
%EQU JJ_LOGOFF#='000200'0;
%EQU JJ_BYPASSD#='000400'0;
%EQU JJ_BAKIC#='001000'0;
%EQU JJ_RUNXCON#='002000'0;
%EQU JJ_DLIB#='004000'0;
%EQU JJ_SAVEING#='010000'0;
%EQU JJ_GETTING#='020000'0;
%EQU JJ_NOSAVE#='040000'0;
%EQU JJ_SCON#='100000'0;
%EQU JJ_UDELTA#='200000'0;
%EQU JJ_EVENT#='400000'0;

--> B$JIT.XCONF <--
%EQU XC_QUIT#='001'0;
%EQU XC_ASL#='002'0;
%EQU XC_URND#='004'0;
%EQU XC_DOMAIN#='007'0;
%EQU XC_PRNST#='770'0;
```

Figure A-2. B$JITO Structure (cont. next page)

```
--> B$JIT.STEPCC <--
%EQU CC_EXIT#=0;
%EQU CC_ERR#=4;
%EQU CC_XXX#=6;

--> B$JIT.XLIMFLG <--
%EQU XL_TIME#='001'0;
%EQU XL_STACK#='002'0;
%EQU XL_TAPE#='004'0;
%EQU XL_TDISC#='010'0;
%EQU XL_PDISC#='020'0;
%EQU XL_DO#='040'0;
%EQU XL_LO#='100'0;
%EQU XL_MEM#='200'0;
%EQU XL_PO#='400'0;

--> B$JIT.SVLDTF <--
%EQU SVL_DIC#='200'0;
%EQU SVL_EXIT#='100'0;
%EQU SVL_TRTN#='040'0;
%EQU SVL_READY#='140'0;
%EQU SVL_LINK#='020'0;
%EQU SVL_LDTRC#='010'0;
%EQU SVL_LYNX#='030'0;
%EQU SVL_MSAVE#='004'0;
%EQU SVL_YCSAVE#='002'0;
%EQU SVL_SAVE#='006'0;
%EQU SVL_ECCB#='016'0;
%EQU SVL_GET#='001'0;

--> B$JIT.JUNK2 <--
%EQU JJ2_DBRK#='000001'0;
%EQU JJ2_PACCESS#='000002'0;

       012345678 012345678 012345678 012345678
      +---------+---------+---------+---------+ DCL
   .0 |         |         |         |         | 1 B$JIT STCLASS DALIGNED,
   .0 |4214     |         |         |         |   2 MODE        UBIN(4) UNAL,
   .0 |    21421|         |         |         |   2 *           BIT(5),
   .0 |         |uuuuuuuuu|         |         |   2 USER        UBIN(9) UNAL,
   .0 |         |         |uuuuuuuuu|uuuuuuuuu|   2 SYSID       UBIN HALF UNAL,
   .1 |ccccccccc|ccccccccc|ccccccccc|ccccccccc|   2 ACCN        CHAR(8),
   .2 |ccccccccc|ccccccccc|ccccccccc|ccccccccc|
   .3 |ccccccccc|ccccccccc|ccccccccc|ccccccccc|   2 UNAME       CHAR(12),
   .4 |ccccccccc|ccccccccc|ccccccccc|ccccccccc|
   .5 |ccccccccc|ccccccccc|ccccccccc|ccccccccc|
   .6 |ccccccccc|ccccccccc|ccccccccc|ccccccccc|   2 FACCN       CHAR(8),
   .7 |ccccccccc|ccccccccc|ccccccccc|ccccccccc|
  .10 |ccccccccc|ccccccccc|ccccccccc|ccccccccc|   2 WOO         CHAR(8),
  .11 |ccccccccc|ccccccccc|ccccccccc|ccccccccc|
  .12 |         |         |         |         |   2 ERR,
  .12 |421421421|421      |         |         |     3 FCG       BIT(12),
  .12 |         |   421421|         |         |     3 MID       BIT(6),
  .12 |         |         |4        |         |     3 MON       BIT(1),
  .12 |         |         | 21421421|421421   |     3 CODE      UBIN(14) UNAL,
  .12 |         |         |         |      421|     3 SEV       UBIN(3) UNAL,
  .13 |         |         |         |         |   2 PRIV,
  .13 |bbbbbbbbb|bbbbbbbbb|bbbbbbbbb|bbbbbbbbb|     3 ACTIVE    BIT(36),
  .14 |bbbbbbbbb|bbbbbbbbb|bbbbbbbbb|bbbbbbbbb|     3 AUTH      BIT(36),
  .15 |bbbbbbbbb|bbbbbbbbb|bbbbbbbbb|bbbbbbbbb|     3 JOB       BIT(36),
  .16 |bbbbbbbbb|bbbbbbbbb|bbbbbbbbb|bbbbbbbbb|     3 PRC       BIT(36),
  .17 |bbbbbbbbb|bbbbbbbbb|bbbbbbbbb|bbbbbbbbb|     3 SAVED     BIT(36),
  .20 |bbbbbbbbb|bbbbbbbbb|bbbbbbbbb|bbbbbbbbb|   2 PPRIV      BIT(36),
  .21 |ccccccccc|ccccccccc|ccccccccc|ccccccccc|   2 FPSN        CHAR(6),
  .22 |ccccccccc|ccccccccc|         |         |
  .22 |         |         |uuuuuuuuu|         |   2 OUTPRIO     UBIN(9) UNAL,
  .22 |         |         |         |uuuuuuuuu|   2 DCBNO       UBIN(9) UNAL,
```

Figure A-2. B$JITO Structure (cont. next page)

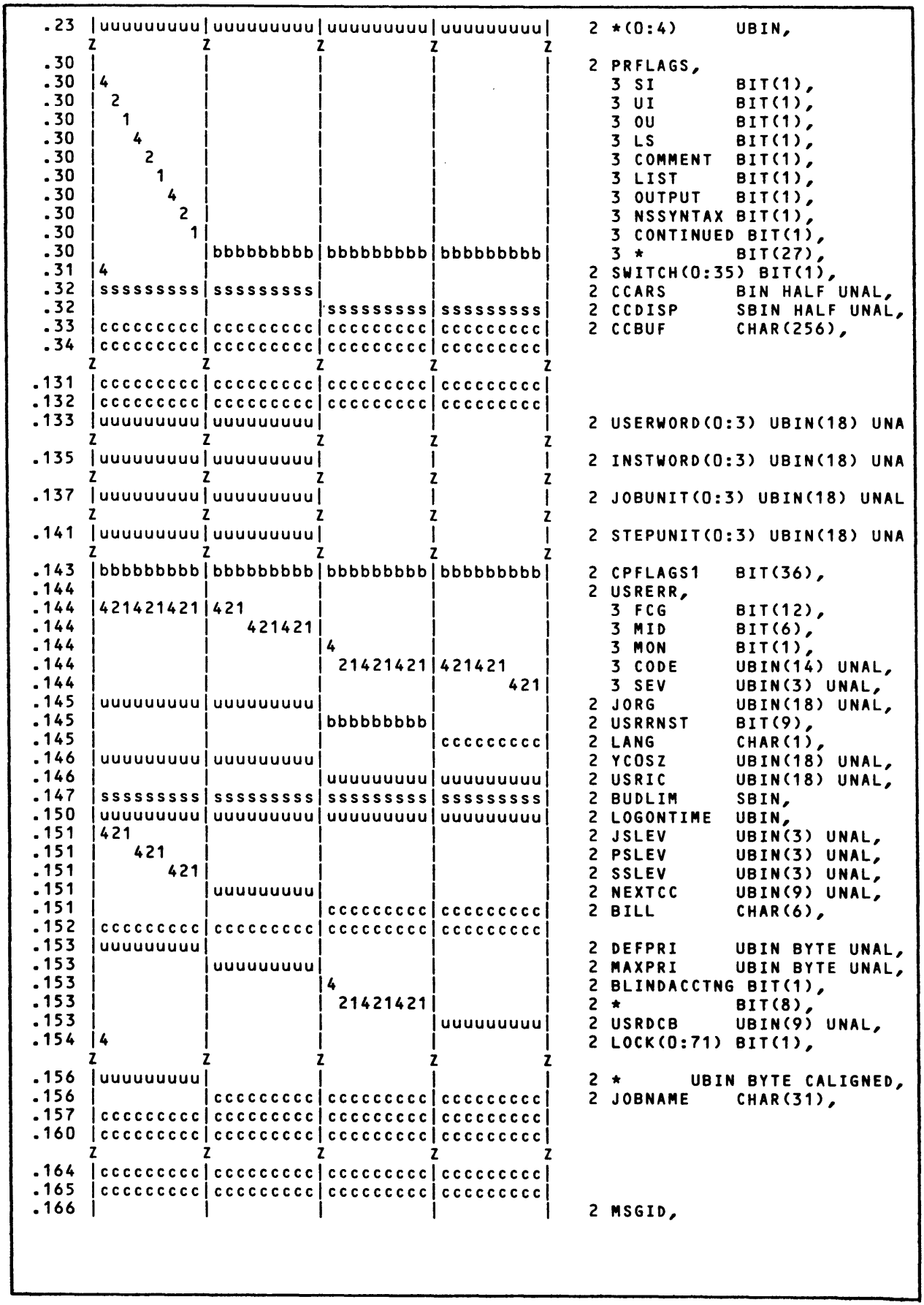


Figure A-2. B$JITO Structure (cont. next page)

```
.166 |uuuuuuuuu|uuuuuuuuu|uuuuuuuuu|uuuuuuuuu|     3 PRIMARY  UBIN,
.167 |uuuuuuuuu|uuuuuuuuu|uuuuuuuuu|uuuuuuuuu|     3 XT       UBIN,
.170 |bbbbbbbbb|         |         |         |   2 PROG_ENTRY BIT(9),
.170 |         |bbbbbbbbb|bbbbbbbbb|bbbbbbbbb|   2 *          BIT(27),
.171 |uuuuuuuuu|uuuuuuuuu|uuuuuuuuu|uuuuuuuuu|   2 *(0:4)     UBIN,
     Z         Z         Z         Z         Z
.176 |uuuuuuuuu|uuuuuuuuu|         |         |   2 PLL    UBIN HALF HALIGNED,
.176 |         |         |uuuuuuuuu|uuuuuuuuu|   2 PUL    UBIN HALF HALIGNED,
.177 |uuuuuuuuu|uuuuuuuuu|         |         |   2 DLL    UBIN HALF HALIGNED,
.177 |         |         |uuuuuuuuu|uuuuuuuuu|   2 DUL    UBIN HALF HALIGNED,
.200 |uuuuuuuuu|uuuuuuuuu|         |         |   2 DDLL   UBIN HALF HALIGNED,
.200 |         |         |uuuuuuuuu|uuuuuuuuu|   2 DDUL   UBIN HALF HALIGNED,
.201 |uuuuuuuuu|uuuuuuuuu|         |         |   2 PCP    UBIN HALF HALIGNED,
.201 |         |         |uuuuuuuuu|uuuuuuuuu|   2 PCD    UBIN HALF HALIGNED,
.202 |uuuuuuuuu|uuuuuuuuu|         |         |   2 PCDS   UBIN HALF HALIGNED,
.202 |         |         |uuuuuuuuu|uuuuuuuuu|   2 PCC    UBIN HALF HALIGNED,
.203 |uuuuuuuuu|uuuuuuuuu|         |         |   2 PCROS  UBIN HALF HALIGNED,
.203 |         |         |uuuuuuuuu|uuuuuuuuu|   2 PCDD   UBIN HALF HALIGNED,
.204 |uuuuuuuuu|uuuuuuuuu|         |         |   2 TDP    UBIN HALF HALIGNED,
.204 |         |         |uuuuuuuuu|uuuuuuuuu|   2 EUP    UBIN HALF HALIGNED,
.205 |uuuuuuuuu|uuuuuuuuu|         |         |   2 FBUC   UBIN HALF HALIGNED,
.205 |         |         |uuuuuuuuu|uuuuuuuuu|   2 FBUL   UBIN HALF HALIGNED,
.206 |         |         |         |         |   2 MMFLGS,
.206 |4        |         |         |         |     3 FREE_PPGS BIT(1),
.206 | 21421421|421421421|         |         |     3 *         BIT(17),
.206 |         |         |uuuuuuuuu|uuuuuuuuu|   2 *      UBIN HALF HALIGNED,
.207 |uuuuuuuuu|uuuuuuuuu|         |         |   2 *      UBIN HALF HALIGNED,
.207 |         |         |uuuuuuuuu|uuuuuuuuu|   2 PPC    UBIN HALF HALIGNED,
.210 |uuuuuuuuu|uuuuuuuuu|         |         |   2 MAXCORE UBIN HALF HALIGNED
.210 |         |         |uuuuuuuuu|uuuuuuuuu|   2 CURRCORE UBIN HALF HALIGNE
.211 |uuuuuuuuu|uuuuuuuuu|         |         |   2 SPEAK  UBIN HALF HALIGNED,
.211 |         |         |uuuuuuuuu|uuuuuuuuu|   2 JPEAK  UBIN HALF HALIGNED,
.212 |uuuuuuuuu|uuuuuuuuu|         |         |   2 JRESPEAK UBIN HALF HALIGNE
.212 |         |         |         |         |   2 RESPEAK REDEF JRESPEAK
.212 |uuuuuuuuu|uuuuuuuuu|         |         |            UBIN HALF HALIGNED,
.212 |         |         |uuuuuuuuu|uuuuuuuuu|   2 PSEUDOPGS UBIN HALF HALIGN
.213 |uuuuuuuuu|uuuuuuuuu|         |         |   2 SRESPEAK UBIN HALF HALIGNE
.213 |         |         |uuuuuuuuu|uuuuuuuuu|   2 PCDDS    UBIN HALF UNAL,
.214 |uuuuuuuuu|uuuuuuuuu|         |         |   2 PCADS    UBIN HALF UNAL,
.214 |         |         |uuuuuuuuu|uuuuuuuuu|   2 LLL      UBIN HALF UNAL,
.215 |uuuuuuuuu|uuuuuuuuu|         |         |   2 LUL      UBIN HALF UNAL,
.215 |         |         |uuuuuuuuu|uuuuuuuuu|   2 PCL      UBIN HALF UNAL,
.216 |uuuuuuuuu|uuuuuuuuu|         |         |   2 PCV    UBIN HALF HALIGNED,
.216 |         |         |uuuuuuuuu|uuuuuuuuu|   2 *      UBIN HALF HALIGNED,
.217 |         |         |         |         |   2 VIRTUAL,
.217 |uuuuuuuuu|         |         |         |     3 DCB#(0:2) UBIN(9) CALIGN
.217 |         |         |         |uuuuuuuuu|     3 *     UBIN(9) CALIGNED,
.220 |4        |         |         |         |   2 FEXT(0:35) BIT(1),
.221 |sssssssss|sssssssss|         |         |   2 DEFEXP  SBIN HALF UNAL,
.221 |         |         |sssssssss|sssssssss|   2 MAXEXP  SBIN HALF UNAL,
.222 |         |         |         |         |   2 STAR(0:7),
.222 |uuuuuuuuu|uuuuuuuuu|uuuuuuuuu|uuuuuuuuu|     3 DA       UBIN,
     Z         Z         Z         Z         Z
.232 |ppppppppp|ppppppppp|ppppppppp|ppppppppp|   2 DCB$       PTR,
.233 |ppppppppp|ppppppppp|ppppppppp|ppppppppp|   2 DO$        PTR,
.234 |ccccccccc|ccccccccc|ccccccccc|ccccccccc|   2 HPSN       CHAR(6),
.235 |ccccccccc|ccccccccc|         |         |
.235 |         |         |bbbbbbbbb|bbbbbbbbb|   2 FACNACS    BIT(18),
.236 |sssssssss|sssssssss|sssssssss|sssssssss|   2 STMPDPPK   SBIN,
.237 |sssssssss|sssssssss|sssssssss|sssssssss|   2 JTMPDPPK   SBIN,
.240 |sssssssss|sssssssss|sssssssss|sssssssss|   2 CURTMPDP   SBIN,
.241 |sssssssss|sssssssss|sssssssss|sssssssss|   2 MAXTMPDP   SBIN,
.242 |sssssssss|sssssssss|sssssssss|sssssssss|   2 PRDPRM     SBIN,
.243 |uuuuuuuuu|uuuuuuuuu|uuuuuuuuu|uuuuuuuuu|   2 GAC(0:2)   UBIN,
     Z         Z         Z         Z         Z
.246 |         |         |         |         |   2 TMPGAC,
.246 |uuuuuuuuu|uuuuuuuuu|uuuuuuuuu|uuuuuuuuu|     3 N        UBIN,
```

Figure A-2. B$JITO Structure (cont. next page)

```
.247 |uuuuuuuuu|uuuuuuuuu|uuuuuuuuu|uuuuuuuuu|      3 TIME      UBIN,
.250 |uuuuuuuuu|         |         |         |    2 FACNCM      UBIN(9) UNAL,
.250 |         |bbbbbbbbb|         |         |    2 *(0:2)      BIT(9),
.251 |uuuuuuuuu|uuuuuuuuu|uuuuuuuuu|uuuuuuuuu|    2 *(0:6)      UBIN,
     Z         Z         Z         Z         Z
.260 |sssssssss|sssssssss|sssssssss|sssssssss|    2 CEXT        SBIN,
.261 |sssssssss|sssssssss|sssssssss|sssssssss|    2 STIME       SBIN,
.262 |sssssssss|sssssssss|sssssssss|sssssssss|    2 XTIME       SBIN,
.263 |sssssssss|sssssssss|sssssssss|sssssssss|    2 CTIME       SBIN,
.264 |sssssssss|sssssssss|sssssssss|sssssssss|    2 INTTIME     SBIN,
.265 |sssssssss|sssssssss|sssssssss|sssssssss|    2 SCHTIME     SBIN,
.266 |sssssssss|sssssssss|sssssssss|sssssssss|    2 IDELTAT     SBIN,
.267 |sssssssss|sssssssss|sssssssss|sssssssss|    2 CALCNT      SBIN,
.270 |sssssssss|sssssssss|sssssssss|sssssssss|    2 TPEXT       SBIN,
.271 |sssssssss|sssssssss|sssssssss|sssssssss|    2 TPSVT       SBIN,
.272 |sssssssss|sssssssss|sssssssss|sssssssss|    2 PMEMTIM     SBIN,
.273 |sssssssss|sssssssss|sssssssss|sssssssss|    2 TUEXT       SBIN,
.274 |sssssssss|sssssssss|sssssssss|sssssssss|    2 TUSVT       SBIN,
.275 |sssssssss|sssssssss|sssssssss|sssssssss|    2 UMEMTIM     SBIN,
.276 |sssssssss|sssssssss|sssssssss|sssssssss|    2 MRT         SBIN,
.277 |sssssssss|sssssssss|         |         |    2 RCOMT       SBIN HALF UNAL,
.277 |         |         |sssssssss|sssssssss|    2 RCURT       SBIN HALF UNAL,
.300 |sssssssss|sssssssss|         |         |    2 RESPT       SBIN HALF UNAL,
.300 |         |         |sssssssss|sssssssss|    2 TURNT       SBIN HALF UNAL,
.301 |uuuuuuuuu|uuuuuuuuu|uuuuuuuuu|uuuuuuuuu|    2 UTIMER      UBIN UNAL,
.302 |uuuuuuuuu|         |         |         |    2 PNR         UBIN(9) UNAL,
.302 |uuuuuuuuu|         |         |         |    2 PRIINC REDEF PNR UBIN(9) U
.302 |         |4        |         |         |    2 RERUN       BIT(1),
.302 |         | 21421421|         |         |    2 *           UBIN(8) UNAL,
.302 |         |         |uuuuuuuuu|uuuuuuuuu|    2 LBJID       UBIN HALF UNAL,
.303 |uuuuuuuuu|uuuuuuuuu|uuuuuuuuu|uuuuuuuuu|    2 EXTUS       UBIN,
.304 |uuuuuuuuu|uuuuuuuuu|uuuuuuuuu|uuuuuuuuu|    2 SVTUS       UBIN,
.305 |uuuuuuuuu|uuuuuuuuu|uuuuuuuuu|uuuuuuuuu|    2 *(0:6)      UBIN,
     Z         Z         Z         Z         Z
.314 |bbbbbbbbb|         |         |         |    2 RNST        BIT(9),
.314 |         |bbbbbbbbb|         |         |    2 FRS         BIT(9),
.314 |         |         |bbbbbbbbb|         |    2 RUNFLAGS    BIT(9),
.314 |         |         |         |uuuuuuuuu|    2 LNKCNT      UBIN(9) UNAL,
.315 |bbbbbbbbb|bbbbbbbbb|         |         |    2 JUNK        BIT(18),
.315 |         |         |sssssssss|sssssssss|    2 STEPS       SBIN HALF UNAL,
.316 |bbbbbbbbb|         |         |         |    2 XCONF       BIT(9),
.316 |         |uuuuuuuuu|         |         |    2 STEPCC      UBIN(9) UNAL,
.316 |         |         |bbbbbbbbb|         |    2 XLIMFLG     BIT(9),
.316 |         |         |         |bbbbbbbbb|    2 SVLDTF      BIT(9),
.317 |bbbbbbbbb|bbbbbbbbb|bbbbbbbbb|bbbbbbbbb|    2 YCERR      BIT(36) ALIGNED,
.320 |sssssssss|sssssssss|         |         |    2 CURPNUM     SBIN HALF UNAL,
.320 |         |         |sssssssss|sssssssss|    2 HIGHPNUM    SBIN HALF UNAL,
.321 |bbbbbbbbb|bbbbbbbbb|         |         |    2 JUNK2       BIT(18),
.321 |         |         |bbbbbbbbb|bbbbbbbbb|    2 *           BIT(18),
.322 |sssssssss|sssssssss|sssssssss|sssssssss|    2 *(0:1)      SBIN,
.324 |sssssssss|sssssssss|         |         |    2 REMCP0      SBIN HALF UNAL,
.324 |         |         |sssssssss|sssssssss|    2 REML0       SBIN HALF UNAL,
.325 |sssssssss|sssssssss|         |         |    2 REMD0       SBIN HALF UNAL,
.325 |         |         |sssssssss|sssssssss|    2 INTER       SBIN HALF UNAL,
.326 |sssssssss|sssssssss|sssssssss|sssssssss|    2 STDLOPGS    SBIN WORD,
.327 |         |         |         |         |    2 ACCESS,
.327 |sssssssss|sssssssss|sssssssss|sssssssss|      3 PACKS     SBIN,
.330 |sssssssss|sssssssss|sssssssss|sssssssss|      3 TAPES     SBIN,
.331 |sssssssss|sssssssss|sssssssss|sssssssss|      3 FORMS     SBIN,
.332 |sssssssss|sssssssss|         |         |    2 MOUNTS(0:2) SBIN HALF UNAL
.333 |         |         |4        |         |    2 OLTA(0:1)   BIT(1),
.333 |         |         |  1421421|421421421|    2 ARECX       UBIN(16) UNAL,
.334 |uuuuuuuuu|         |         |         |    2 CURSUD0(0:7) UBIN BYTE UNA
     Z         Z         Z         Z         Z
.336 |uuuuuuuuu|uuuuuuuuu|uuuuuuuuu|uuuuuuuuu|    2 RESCORE     UBIN WORD,
.337 |         |         |         |         |    2 TSLINE,
.337 |uuuuuuuuu|         |         |         |      3 FEX       UBIN(9) UNAL,
```

Figure A-2. B$JIT0 Structure (cont. next page)

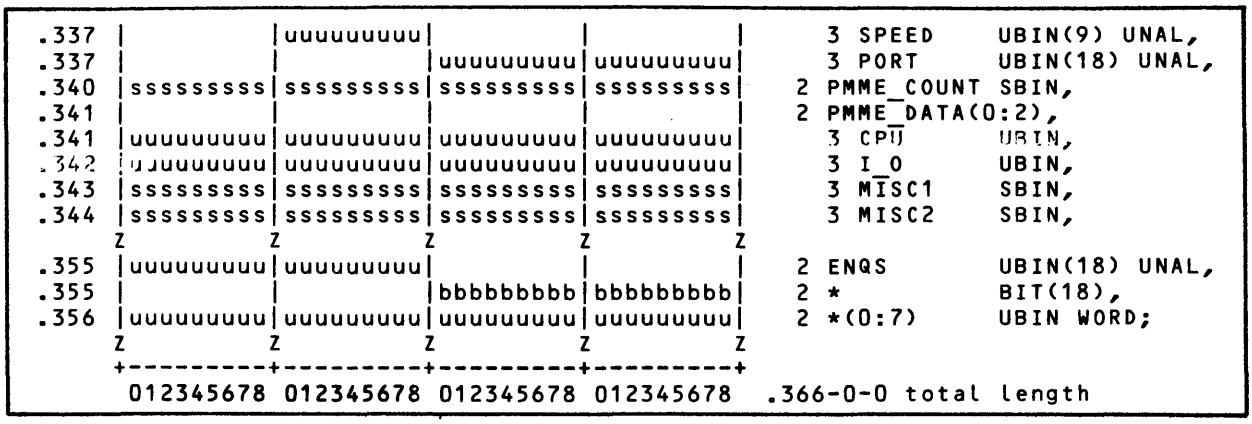


Figure A-2. B$JITO Structure

```
   ..... B$JITO=B$JITO                                    ..... STCLASS=BASED
      012345678 012345678 012345678 012345678
     +---------+---------+---------+---------+ DCL
   .0 |         |         |         |         | 1 B$JITO STCLASS DALIGNED,
   .0 |uuuuuuuuu|uuuuuuuuu|uuuuuuuuu|uuuuuuuuu|   2 *(0:23)    UBIN WORD,
      Z         Z         Z         Z         Z
  .30 |uuuuuuuuu|uuuuuuuuu|uuuuuuuuu|uuuuuuuuu|   2 *(0:101)   UBIN,
      Z         Z         Z         Z         Z
 .176 |uuuuuuuuu|uuuuuuuuu|uuuuuuuuu|uuuuuuuuu|   2 *(0:17)    UBIN,
      Z         Z         Z         Z         Z
 .220 |uuuuuuuuu|uuuuuuuuu|uuuuuuuuu|uuuuuuuuu|   2 *(0:31)    UBIN,
      Z         Z         Z         Z         Z
 .260 |uuuuuuuuu|uuuuuuuuu|uuuuuuuuu|uuuuuuuuu|   2 *(0:27)    UBIN,
      Z         Z         Z         Z         Z
 .314 |uuuuuuuuu|uuuuuuuuu|uuuuuuuuu|uuuuuuuuu|   2 *(0:7)     UBIN,
      Z         Z         Z         Z         Z
 .324 |uuuuuuuuu|uuuuuuuuu|uuuuuuuuu|uuuuuuuuu|   2 *(0:33)    UBIN;
      Z         Z         Z         Z         Z
      +---------+---------+---------+---------+
       012345678 012345678 012345678 012345678  .366-0-0 total length
```

Figure A-3. B$JITOX Structure

Index

Note: Index references indicate the page on which the paragraph containing the index term actually ends. Should the paragraph straddle two pages, the actual indexed term might be on the first page, while the index reference is to the second page.

A

B

C

D

E

F

G

H

I

J

K

L

M

N

O

P

Q

R

S

T

U

V

W

X

Y

HONEYWELL INFORMATION SYSTEMS
Technical Publications Remarks Form

TITLE: CP-6 SYSTEM PROGRAMMER GUIDE

ORDER NO. CE62-00

DATED JANUARY 198

ERRORS IN PUBLICATION

SUGGESTIONS FOR IMPROVEMENT TO PUBLICATION

Your comments will be investigated by appropriate technical personnel and action will be taken as required. Receipt of all forms will be acknowledged; however, if you require a detailed reply, check here. ☐

FROM: NAME _____ DATE _____

TITLE _____

COMPANY _____

ADDRESS _____

_____

LEASE FOLD AND TAPE–
OTE: U. S. Postal Service will not deliver stapled forms

NO POSTAGE
NECESSARY
IF MAILED
IN THE
UNITED STATES

BUSINESS REPLY MAIL
FIRST CLASS PERMIT NO. 39531 WALTHAM, MA02154

POSTAGE WILL BE PAID BY ADDRESSEE

HONEYWELL INFORMATION SYSTEMS
200 SMITH STREET
WALTHAM, MA 02154

ATTN: PUBLICATIONS, MS486

Honeywell

Together, we can find the answers.

Honeywell

Honeywell Information Systems
U.S.A.: 200 Smith St., MS 486, Waltham, MA 02154
Canada: 155 Gordon Baker Rd., Willowdale, ON M2H 3N7
U.K.: Great West Rd., Brentford, Middlesex TW8 9DH **Italy**: 32 Via Pirelli, 20124 Milano
Mexico: Avenida Nuevo Leon 250, Mexico 11, D.F. **Japan**: 2-2 Kanda Jimbo-cho Chiyoda-ku, Tokyo
Australia: 124 Walker St., North Sydney, N.S.W. 2060 **S.E. Asia**: Mandarin Plaza, Tsimshatsui East, H.K.

39519, 5C184, Printed in U.S.A.

CE62-00