HYDRA USER DOCUMENTATION

Edited by David Alex Lamb

February 16, 1976

DEPARTMENT
of
COMPUTER SCIENCE

Carnegie-Mellon University

HYDRA USER DOCUMENTATION

Edited by David Alex Lamb

February 16, 1976

This work was supported by the Defence Advance Research Projects Agency under contract number F44620-73-C-0074 and is monitored by the Air Force Office of Scientific Research.

Part VI: General Information

1. Introduction

This document is an addendum to the HYDRA Songbook. As a result, its page numbering may be somewhat unexpected. Chapter 7 replaces the corresponding chapter of the Songbook. The remainder is all new material. It is numbered so that it can be taken apart and filed with the Songbook.

This document follows a certain conventions. If a file name is mentioned without a PPN, it is to be found on N810HY97. All numbers are decimal unless preceded by a hash mark (#), in which case they are octal.

If you have any routines you think are of general interest, write a brief (or wordy, if you care to) description, preferably using the macros discussed in chapter 16. Then send mail to David Lamb and it will be included in an future version of this document.

There is a committee whose responsibilities include supervision of user software on HYDRA. It consists of Sam Harbison, David Lamb, Joe Newcomer, and George Robertson. These are the people you should see about design issues if you are writing software to be used by many people, or if you are using such software and have suggestions.

Portions of this document were largely copied from the Songbook. Guy Almes wrote the file system chapter. Rick Gumpertz documented the routines for which he is the maintainer. Sam Harbison, Phil Karlton, and Joe Newcomer made many helpful comments.

7. Getting started with BLISS-11

7.1. An Overview

BLISS-11 is a programming language for the PDP-11. It is specifically intended to be used for implementing "System Software". As such, it differs from other languages in several significant ways.

> Higher-level languages derive their suitability for a particular problem area: FORTRAN or ALGOL for mathematics, SNOBOL for strings, GPSS for simulations, etc., because they provide to the user a vocabulary and set of conventions suitable to that problem area. The BLISS-11 authors view "Implementation Languages" in a similar way: as application languages where the application is a particular species of hardware. As such, an implementation language must reflect the capabilities and architecture of its machine, and must not block the programmer's use of these capabilities.

> I/O is not a part of BLISS-11. I/O can be done directly in the language much as an assembly program might, or through subroutine calls.

> Every attempt has been made to give the user explicit control over the code his program generates, while providing maximum convenience otherwise.

> There are no explicit or implicit data modes other than the 16-bit binary word. Data modes are essentially user-defined via the STRUCTURE mechanism which allows the user to set or compute an algorithm for data access.

7.2. Steps to a running BLISS program

There is a fairly complicated and hazard-filled path from a BLISS-11 source program to its execution on C.mmp. Remember that BLISS-11 is a batch system, but that you are running it in an environment which provides poor support of batch-mode operation. In brief, the major steps are these:

> Build a BLISS-11 source program with your favorite editor. It should have an extension of .B11.

> Run the source program through the BLISS-11 compiler, producing an assembly language program as output. This output file will have an extension of .P11.

> Run the assembly-language program through the "MACN11" or "MACY11" assemblers, producing a relocatable file as output. This file will have an extension of .OBJ

> Alternately, you may compile your program by saying "LOAD/NOLINK filename"; this combines the BLISS11 and MACN11 steps.

> Using the Hydra linker, produce a link-edited collection of C.mmp page images. This page image file will be stored in your area on the PDP10 disk; it will have an extension of .PAG. The Linker will also produce a storage map in a file with an extension of .MAP.

> Using the special utility program "DT11" copy your page image file (the .PAG file) to a DECtape.

> Carry that DECtape to C.mmp and mount it on a DECtape drive on a processor that is up and configured into the system.

> At the Hydra command interpreter, invoke the command object which loads and initializes BLISS programs. This command object is documented in a later section of this chapter.

> Run widdershins around the switch three times.

7.3. A convenience package for BLISS-11 users under Hydra

A collection of compiler macros, command objects, and runtime routines has recently been developed to ease the pain of writing and debugging BLISS programs under Hydra. This package, HYDUSR, consists of three pieces: a file to require into your source program, object files to link in with your object program, and Hydra command objects to get things running on C.mmp.

7.3.1 The BLISS source program

The HYDUSR runtime routine is the 'main program'; i.e. it contains the starting address for the program. For this reason, you should never include a MAIN or START declaration in your module head if you are going to use the HYDUSR package. Your routine will be called as a procedure from HYDUSR initialization, and HYDUSR will take care of the wrapup after your routine has returned.

HYDUSR will pass control to you at a GLOBAL ROUTINE named HENTRY. You must have a GLOBAL ROUTINE by this name. HYDUSR will work equally well with or without SIX12, but it is suggested that you include SIX12 by compiling with the /D switch or by specifying DEBUG in the module header.

Your program need observe no other conventions, or call any other routines or invoke any other macros. It would in fact take the following general form:

```
MODULE myprog =
    BEGIN
```

```
        REQUIRE hydusr.req[n810hy97];
        ...
        GLOBAL ROUTINE hentry = . . .
        ...
    END
ELUDOM
```

HYDUSR.REQ requires the kernel call and base call definition files KERKAL.REQ[N811HY97] and BASCAL.REQ[N811HY97]. If you do not wish to use kernel calls or base calls in your program, you may use HYDLIB.REQ instead of HYDUSR.REQ. This file declares all of the library routines and defines structures and macros you would use in your programs.

7.3.2 The Runtime Environment

The BLISS programmer must be familiar with the general Hydra environment, and should understand the significance of LNS slots, procedures and processes, Kernel calls, etc. The HYDUSR compile-time require file invokes all of the standard Hydra macro definitions and parameter definitions, so that a HYDUSR program may use any of the kernel calls or base calls at will. The I/O routines in HYDIO and the rest of the HYDLIB library are also included.

7.3.2.1 Programming considerations

When HYDUSR transfers conrol to the user program at the entry point HENTRY, it will have set up various facilities and LNS slots for the user's convenience. When HENTRY is called, the following conditions are met:

> SIX12 has been initialized if it is present; if not, it is ignored. The user may explicitly enter SIX12 by calling SIXCMD, or he may ignore its presence entirely.

> Buffer control blocks have been allocated for teletype input and output (see 19.1 and 19.2).

> SIGPC and ERRPC in the user's LNS have been set up to trap kernel signals into HYDUSR. HYDUSR will, when it receives a signal, print a message and then return the value returned by the kernel call which failed. Hardware errors will also trap to HYDUSR. When one of these errors or signals ocurrs, HYDUSR will call SIXCMD if SIX12 is present.

It is suggested that the user return from HENTRY rather than executing a KRETURN himself, in order that any wrapup which might be necessary can be performed.

7.3.2.2 LNS slot allocations

HYDUSR allocates the first several LNS slots and puts useful things into them. The compile-time require file includes BINDs to the following names, which match those slots:

> SYSDIRECTORY : a capability for the system directory.
> USERDIRECTORY : a capability for the user directory.
> IOPORT : a capability for a port connecting to the TTY.
> HDUSELF : a capability for the procedure object from which the process was created.
> HDUPARMS : the first of a range of slots intended for parameter templates
> HDURETCAP : a slot for a capability to be returned by the procedure
> HDUPAGES : the first of a range of slots containing capabilities for the code pages of the procedure.

7.3.2.3 Addressing and RELOC register considerations

The canonical linker command file causes HYDUSR to use relocation registers 1, 2, and 3, so they should not be changed by a user program. Several CPS slots are also used by HYDUSR. The variable FREECPS is set to the first free CPS slot that the user may safely allocate. The Address Space Management (ASM) routines will be included in HYDUSR when they exist.

7.3.3 Linking your program with the Hydra Linker

The Hydra linker is a big awful hairy complicated program, and everybody is constantly making mistakes in their command files for it. So the HYDUSR package includes, for your convenience and good fortune, a 'Linker Command File Template', which you can hack over with your favorite text editor to make your very own linker command file. The file XXXXXX.LMD[N810HY10] is a ready-to-eat Linker command file: all you have to do is go through with a text editor and change all occurrences of the string XXXXXX into the name of your program. As long as your program will fit onto a single Hydra page (4096 words) you need use no other command file. If you need more than one page, you will have to modify this Linker command file slightly, putting part of your code in other RPS slots besides slot 2.

To run the linker, give it the input @XXXXXX$, where XXXXXX has been changed, of course.

The canonical Linker command file looks like this:

```
&
  Print tracing information while linking
&
% /S
&
```

```
  Send the load map to disk
&
%XXXXXX.MAP
XXXXXX
&
  Put the process-local OWN and GLOBAL sections on the first
  page in the CPS, page 2 of the initial RPS.  The startup
  command object will make a copy of this page so that
  different processes invoked from the same procedure will
  have their own copy of these variables.  This line MUST
  come first (or else the C:1 switch must be specified for
  this page)
&
/R:#40000/Q←XXXXXX[N810HY97](OG),@HYDOWN[N810HY97]
&
  Output the code and PLITS.
&
/R:#20000/Q←XXXXXX[N810HY97](CP),@BASIC[N810HY97]
/R:#60000/Q←@STDRTN[N810HY97],@OPENIT[N810HY97],
        @OUTPUT[N810HY97],@REST[N810HY97]
&
  Output the information used by SIX12
&
/R:#20000/S←XXXXXX[N810HY97]/S,@SIX12S[N810HY97]
/R:#40000/S←@TESTS[N810HY97]
/R:0/N←XXXXXX[N810HY97]/N,@TESTN[N810HY97],@SIX12N[N810HY97]
```

The "$" symbol represents an altmode. Note to SOS users: yes, you <u>can</u> put an altmode in your files; use the escape "?=$"! The files TESTS.LMD and TESTN.LMD provide SIX12 symbol and name tables for the library modules. You may omit these if you don't want to be able to see the library routines from SIX12. If any of the library modules you load have debug linkages, you will get warnings from the linker about CSECTs whose names are of the form XXXX.S; you can safely ignore these messages.

As an exercise, go over the Linker documentation and figure out what this file is all about.

7.4. Executing the BLISS program on C.mmp (at last)

Sit down at a terminal connected to C.mmp and type

```
↑K
LOG()
```

to the command interpreter. If it just sits there and stares at you, the chances are

that Hydra is down, and you are so busy reading this manual that you forgot to notice. Normally, however, it will either prompt you for a name, or say

```
Err 151: Directory lookup failed
LOG
```

In the latter case, you are already logged in; type KJOB to kill that job and repeat the operation. If you have never logged in before, hit carriage return in response to the "Name:" question and it will as you if you want to create a user entry. Reply "y" and it will prompt again for a name. You may chose any ten characters you like, but by convention you should use your last name. When it's finished creating a user entry for you, you have to go back and type LOG again (sigh!).

Now presumeably you are logged in. Type

```
&SYSDIRECTORY.PUBLIC.KARLTON.PROFILE()
```

This will run a command object that defines several macros that make life easier. In particular, you can now type

```
READ(processor, unit, 'filnam')
```

where PROCESSOR and UNIT tell where your DECtape is and FILNAM is the name of your file, without the ".PAG" extension; it can be either upper or lower case. This macro runs a command object that reads in your file from tape, creates a Procedure object, and enters the procedure in your user directory under PROCEDURES.filnam, where "filnam" is the name of the DECtape file. You don't need to create a directory called PROCEDURES; if it doesn't exist it will get created when the READ command object tries to put something into it.

After this point, whenever you wish to run your program you may type

```
RUN(filnam)
```

and several things will happen.

> A copy of your procedure is made, so the command object can do things to the copy without disturbing your original procedure. This copies only the procedure object, and not the pages of your program, which are pointed to by the procedure object. Thus every process you create from your procedure will share the same pages.

> A copy of the own/global page is made and placed in the copied procedure, replacing the original page (so that the original page remains untouched). Thus each time you run your procedure you will get a fresh copy of this data page.

> A port is created and connected to the teletype, and a capability for it is placed

in the C-list of the procedure. A capability for the connection object is placed in your user directory under the name TTY.

> A process is created from the procedure and a capability for it is placed in your user directory under the name CURRENTPRO. You may use this capability to KILL, BLAST, or CONTROL your process (if you care to do any of these strange things).

> A TALK command is given to force the terminal to be talking to the newly-created process. Anything you type will now be transmitted to the program.

> If SIX12 is loaded, SIXCMD will be called. It will print "Pause 0 at:" followed by an address, then wait for commands from your terminal. When you have set whatever breakpoints you want, type "GO".

> Your HENTRY routine is called.

> At any point in the middle of your program you may type a ↑K to break the connection to the process and return to the command interpreter. You may resume talking to your process by typing TALK(TTY).

> At the end of execution, upon return to HYDUSR, the terribly corny message 'End of LNS execution' is printed. You should then type a ↑K to break the terminal back to the command interpreter.

Part VI: General Information

16. Documenting User Routines

This chapter describes how to use a set of PUB macros in documenting user routines for HYDRA. The macros are based on those used in the HYDRA Songbook (and in fact are a superset of them). The Songbook macros are defined in SBOOK.DFS[N810HD99]. The extensions are defined in ROUTIN.DFS[N810DL10]. This document was produced with these macros.

Typically, the first few lines of a file using these macros will be

```
.require "sbook.dfs[n810hd99]" source;
.sourcefile(|routin.dfs[n810dl10]|);
```

16.1. Conventions

The HYDRA library documentation generally follows a set of conventions of usage. BLISS keywords are in lower case letters of the standard character set and are underlined. Names of files, modules, and variables are usually capitalized, using the standard character set. HYDRA kernel calls use the capital letters of NGB25, a boldface character set. The MODULE and ROUTINE macros described below impose a convention on the style of a section, though this is not followed throughout.

More important are conventions about document maintenance. The MODULE macro has a field naming the person responsible for the code; this is the person to whom complaints and suggestions should be sent. For news about changes, experience indicates that maintaining a news file in a standard place is the thing to do.

The MODULE macro forces each module definition to begin on an odd page, so that small changes can be made without having to recreate the whole document.

16.2. Text Responses

The source files define text responses used to perform special purpose functions. A text line of the form

```
text1<und-TEXT>text2
```

will produce an underlined version of the enclosed text, namely

text1<u>TEXT</u>text2

The response

```
<ital-TEXT>
```

switches to the B character set to print the enclosed text. Because of timing problems in the XGP, you must have loaded the B character set at least one full output line before you use the new character set.

16.3. Sectioning

These macros create section, subsection, (etc.) titles and make entries in the table of contents. The easiest way to see their effect is to read the following section with a copy of the Songbook.

All of these macros take a character string argument which provides the title for the chapter, section, or whatever. For each of these macros there is a corresponding one formed by prepending "eval!" to the front of the macro name; this form of the macro takes an expression as an argument, rather than a string. Thus if the PUB variable GORP contains the string "XXXX" then CHAP(GORP) produces a chapter titled GORP, while EVAL!CHAP(GORP) produces a chapter titled XXXX.

Many of these macros have a tag argument which can be used to associate a label with the division in question. The label can be used with the YON macro to provide cross-references (see 16.4.3). If a label is provided it must end with a colon; e.g.

```
chap(|The Hippopotamus|,hippo:)
```

produces a chapter titled "The Hippopotamus", which can be referenced elsewhere as {yon hippo}.

16.3.1 MAJORPART(TITLE)

Begins a new major part, that is, a division labelled "PART" in the Songbook. Each major part begins on a new page. The part number and title are printed on the teletype when the part is reached in pass one of PUB.

16.3.2 CHAP(TITLE,TAG)

Begins a chapter. Chapters are numbered sequentially from 1, and are not renumbered within each new major part. The first chapter of a major part begins on the same page as the part title; all other chapters begin on a new page. The chapter number and title are printed on the teletype when the chapter is reached in pass one of PUB.

16.3.3 SEC(TITLE,TAG)

Begins a section. Sections are numbered sequentially within chapters.

16.3.4 SUBSEC(TITLE,TAG)

Begins a subsection. Subsections are numbered sequentially within sections.

16.3.5 PARA(TITLE,TAG)

Begins a paragraph. Paragraphs are numbered sequentially within subsections.

16.3.6 AIPPENDIX(TITLE,TAG)

Begins an appendix. Appendices behave like chapters, except that their indices print as upper case alphabetics rather than arabic numbers.

16.4. Miscellaneous Macros

16.4.1 SOURCEFILE(FILENAME)

This macro is used to insert other sourcefiles into the text. It is essentially a require FILENAME source, bracketed with a labelled begin - end pair.

16.4.2 CHGFONT(FONT,CHARSET)

This macro is used to change the A or the B character set; it should be used in place of the standard GETFONT macro. CHARSET is the single letter A or B, either upper or lower case. FONT is either a standard font name in quotes, such as "NGR25", or a special Songbook character set name without quotes. The Songbook character sets are summarized in the following table, giving the Songbook name, the corresponding actual character set, and a brief description of its purpose.

TXFONT	BKR25	standard text font
EXFONT	LPT128	Font for examples
H1FONT	NGB25	Subsection titles in the text; section title numbers in the table of contents
H2FONT	NGR30	Section titles in the text; chapter titles in table of contents
H3FONT	NGR40	Chapter titles in the text
H4FONT	BDR40	Part titles in text and table of contents.
HD1FONT	NGB25	Light header font
HD2FONT	NGB30	Intermediate header font
HD3FONT	NGR40B	bold header font
HD4FONT	BDR40	Unit header font

Because of timeing problems in the XGP, a character set must be loaded at least one full output text line before it is used. Thus you must place your CHGFONT requests a reasonable distance before you use the character set, and you cannot use more than two character sets on a line. Also, changing the A character set can cause problems.

16.4.3 Cross References

As metioned in 16.3, most of the sectioning macros allow a tag to be attatched to the division started by the macro. Elsewhere, the division number (e.g. 1 for a chapter, 1.2 for a section, and so on) can be inserted at appropriate places in the text by using the YON macro. Since the macro must be evaluated in command mode, its use must be preceded by an open curly brace and terminated by a closing curly brace. Thus for example in this document the section on the SUBSEC macro is labelled XSUBSEC; the text "(see {yon xsubsec})" produces "(see 16.3.4)".

16.4.4 Examples

Two macros, EXAMPLE and ENDEXAMPLE, can be used to bracket sections of text which give programming examples. EXAMPLE goes into NOFILL mode, sets the B character set to EXFONT and selects the B character set, sets the paragraph separation to 0, and indents the text seven spaces. ENDEXAMPLE restores things to their previous state.

16.4.5 Tables

The macros STARTTABLE and ENDTABLE can be used to bracket tables. STARTTABLE takes three numbers as arguments, giving the starting positions for each of three columns in the table. It is assumed that the first three columns will be short enough to fit in the available space; the final column may extend across several lines, and subsequent lines will be properly aligned. Each new entry in the table should be separated from the previous one by a blank line. The STARTTABLE macro also goes into GROUP mode, so that the table will appear all on a page if possible.

16.4.6 Indexing and the Table of Contents

The macro BACK, placed at the end of your document, causes the table of contents to be prepared and prints the index. Entries in the table of contents are made by the sectioning macros (see 16.3). The table of contents will be placed wherever you have said "insert CONTENTS".

The macros IX and EVAL!IX are used to make entries in the index. IX takes an unevaluated string argument; EVAL!IX evaluates its argument. The macro PRINTINDEX causes the index to be placed at the point of invocation; a PRINTINDEX is done by the standard BACK macro.

16.5. Module Documenting

This section describes the macros in ROUTIN.DFS[N810DL10]. Their effect is best understood by reading this section along with a sample section from the description of BLISS routines under HYDRA (see part VII).

16.5.1 MODULE(TITLE,FILNAM,PGMER,CODE,DEBUG,DATA)

This macro begins the documentation for a module. It is on the same level as the SEC macro (see 16.3.3), except that each module begins on a new odd page; this is to ease maintenance of the documentation. FILNAM is the name of the module (source, P11, object file). It is also used as a tag for cross-referencing. PGMER identifies the last known maintainer of the code; if left blank, it prints as "No known maintainer". CODE, DEBUG, and DATA are sizes in words of the code without debug linkages, the code with debug linkages, and the data of the module.

16.5.2 ROUTINE(NAME,CALLSEQ,CODE,DEBUG,DATA,INNAME)

Begins the documentation for a routine; corresponds to the SUBSEC macro (see 16.3.4). NAME is the name of the routine, the identifier used to invoke the routine. CALLSEQ is a list of identifiers naming arguments to the routine. For example,

```
routine(ATAN2,|X,Y|)
```

would start the documentation for the FORTRAN two-argument routine ATAN2. If CALLSEQ is omitted, it is presumed that the routine takes no arguments. CODE, DEBUG, and DATA are as for the MODULE macro. INNAME is the internal name of the routine; it is intended for cases where users invoke the routine via a long external name which is a macro for the short internal name. This macro also makes an entry for the routine in the calling sequence summary and the index.

16.5.3 FILE(NAME,PPN)

This macro makes entries in the index for the given file. A corresponding macro, EVAL!FILE, takes expressions as arguments.

16.5.4 CROSSREF(REFFED,REFFER)

REFFED and REFFER are both string expressions. Makes an entries in the cross-reference summaries, indicating that REFFED is called by REFFER.

16.5.5 REF(REFFED)

Calls CROSSREF to indicate that the current routine or module (the contents of the variable CURRENTREF, set by the MODULE and ROUTINE macros) calls module REFFED.

16.5.6 MODSIZE(MODNAME,CODE,DATA,DEBUG)

Makes an entry in the sizes summary for the module named MODNAME, with code, debug, and data sizes given by the corresponding variables. The file name for the module is taken from the variable FILENAME, set by the MODULE macro or by explicit asignment. There is a corresponding EVAL!MODSIZE macro which takes an expression as an argument.

16.5.7 RTNSIZE(RTNNAME,CODE,DEBUG,DATA)

As with MODSIZE, except the entry is made for a routine rather than a module (the format is slightly different).

16.5.8 NROUTINE(NAME,CALLSEQ,TAG)

This module is much like a stripped-down variant of ROUTINE, except that it does not make any index entries or entries to the calling sequence summary.

16.5.9 RTNCALL(NAME,CALLSEQ,MODULE)

This macro makes an entry into the calling sequence summary. NAME and CALLSEQ are as for ROUTINE. MODULE is the name of the module containing the routine.

16.5.10 PRINTCALL()

This macro prints a summary of routine calling sequences. It should be the last thing in the chapter which documents the routines.

16.5.11 PRINTFREF()

Print a summary of cross-references, ordered by calling routine.

16.5.12 PRINTBREF()

Print a summary of cross-references, ordered by called routine.

16.5.13 PRINTNAME()

Print the summary of external name/internal name correspondences.

16.5.14 PRINTSIZE()

Print the summary of module and routine sizes.

17. FilSys -- an Initial File System

The FilSys subsystem implements a simple form of disk transput geared toward the general needs of users of C.mmp. Objects of TYPE File are randomly addressable, but sequential access is conveniently and efficiently provided. Transmissions may be of variable length, but blocked transmissions do improve efficiency. They may be accessed simultaneously by several sophisticated users, but the mutual exclusion used by ordinary system utilities is also provided. While performance is rather modest, it is hoped that these files will be useful until more sophisticated file systems are designed and implemented.

17.1. The File Type

A File object has both a Data Part and a C-List. The Data Part includes the file status and usage information listed in Table 17.1. These fields can be interrogated by users through FQuery, but can be modified only by FilSys. The C-List has three entries:

1. WSema, a SEMAPHORE that provides exclusive access among the Writers of the File.

2. RPfile, an object that holds the data being accessed by current Readers and Updaters.

3. WPfile, an object that holds the data being accessed by the current Writer.

In the initial PAGE-based implementation, RPfile and WPfile are each UNIVERSAL objects holding capabilities for the PAGEs of the File. Since each can hold 120 capabilities, PAGE-based Files are limited in size to 960K bytes, subject further to the capacity of the Kernel paging system.

The data within a File is addressed with 32-bit byte addresses. In these long integers, the first word is the least significant. Seek addresses, physical sizes, and end-of-file addresses are all of this form. See 17.4 for the use of addresses in transput operations.

Associated with the File type are four auxiliary rights listed in Table 17.2. These rights control CALLs to the various PROCEDUREs of the subsystem. These procedures are described in 17.5 and 17.3.

Fields of the File Data Part

Field Name	no. wds	Meaning
FAcDat	4	64-bit clock on last open
FMoDat	4	64-bit clock on last modification
FCrDAt	4	64-bit clock on creation
FEoF	2	Byte address of first undefined byte
FPSiz	2	Number of bytes of physical storage
FRdCnt	2	Number of read operations
FWrCnt	2	Number of write operations
FOpCnt	2	Number of opens
FDevTp	1	Device type code
FPName	10	File Print Name

Table 17.1

File Type Auxiliary Rights

Aux sRight	octal	PROCEDURE controlled
FSReadRts	100000	FOpnRd
FSWriteRts	40000	FOpnWr
FSUpdateRts	20000	FOpnUp
FSQueryRts	10000	FQuery

Table 17.2

17.2. File Creation

Files are CREATed by CALLing the FilSys PROCEDURE FCreat. FCreat takes a single DATA parameter of eleven words. The first word is a Device Type code from Table 17.3 (currently, the PAGE-based 'Disk' type is the only implemented Device Type). Following this is an asciz print name of up to twenty characters. This name is stored in the Data Part of the object and can be inspected, but is otherwise ignored by FilSys. Upon a successful CALL, FCreat returns a 0 numeric value and a capability for the new File object. This capability has all auxiliary rights and the Kernel rights ENVRTS, MDFYRTS, DLTRTS, and UCNFRTS. Should the Device Type code be illegal or unimplemented, the signal ErDev will result.

File Device Type Codes

Device	Code	Meaning
FSDevDisk	1	PAGE-based Disk File
FSDevLinePtr	2	Spooled Line Printer
FSDevDECTape	3	DEC-Tape
FSDevMagTape	4	Nine-track Mag Tape
FSDevPipe	5	UNIX-style pipe

Table 17.3

17.3. File Opening

To open a File for transput, one of the three PROCEDURES FOpnRd, FOpnWr, or FOpnUp (denoted generically as FOpnxx in this document) is CALLed, depending on the permissions desired. Each of these PROCEDURES requires three parameters. The first is a File object with ENVRTS, MDFYRTS, UCNFRTS, and an auxiliary right peculiar to the open PROCEDURE CALLed. The second is a PORT object with UCNFRTS and PCONNRTS; this PORT will be CONNECTed to the FilSys PORT for all File transput. The third is a DATA object containing two words. The first word, WaitV, is currently ignored; eventually it will specify how long we may block on WSema on an FOpnWr CALL. The second word, OChan, is the output channel of the PORT to be CONNECTed to the FilSys PORT; since the CONNECT KALL will attempt to find a free output channel if a -1 is passed, this value is allowed here and passed directly to the CONNECT. Upon a successful CALL, FOpnxx returns only the numeric value of the output channel used. Transput messages can then be sent.

FilSys Signals and Error Minor Statuses

Symbol	octal	Meaning	Source
FSErrFull	150101	File System Full	FOpnxx
FSErrBusy	150102	File is Busy	FOpnWr
FSErrDev	150103	Invalid Device Type	FCreat
FSErrEoF	150104	End of File	FilMon
FSErrPerm	150105	Invalid Permissions	FilMon
FSErrOpCd	150106	Invalid MOpCod Field	FilMon
FSErrMess	150107	Bad Message Buffer	FilMon
FSErrNWrt	150110	File Not Yet Written	FOpnRd
FSErrPort	150111	Bad Port or Chan Arg	FOpnxx
FSErrRTim	150112	Insufficient RunTime	FOpnxx

Table 17.4

To become a Reader of a File, FOpnRd is CALLed with a File parameter containing FRdRts. The signal ErNWrt will result unless the File has either been

written and closed by a Writer or been opened by an Updater. The current RPfile is then opened for read operations.

To become the Writer of a File, FOpnWr is CALLed with a File parameter containing FWrRts. A $PCONDITIONAL will be performed on WSema; should this fail, the signal ErBusy will result (the parameter WaitV will eventually be used here). A new WPfile is CREATed and opened for read and write operations. Upon an FClose, this WPfile will supercede the current RPfile for subsequent FOpnRd's and FOpnUp's.

To become an Updater of a File, FOpnUp is CALLed with a File parameter containing FUpRts. The current RPfile is then opened for in-place read and write operations. Updating is intended for sophisticated applications and no mutual exclusion is provided; caution in its use is therefore suggested.

Several signals may result from any FOpnxx CALL: ErFull indicates that FilSys capacity for open Files is temporarily exhausted; ErPort indicates that the CONNECT operation failed due to a bad PORT or OChan specification; ErRTim indicates that a RUNTIME, necessary for protection, failed.

FilSys Message Header Format

Field	Word Offset	Meaning
FSMessOpCode	0	Message OpCode
FSMessByteCount	1	Requested Byte Count
FSMessAddress	2	32-bit Byte Address
FSMessActualCount	6	Actual Byte Count
FSMessStatus	7	Minor Error Status

Table 17.5

17.4. Transput Operations

Once a File is opened, transput can be performed by RSVPing requests to the FilSys PORT. These messages are RECEIVEd by FilMon, the File Monitor PROCESS, which checks the message for validity, performs the necessary transput, and REPLYs the message. When the user RECEIVEs the message again, the transput will have been performed. The messages in this communication have two parts: a fixed length header of eight words and a variable length buffer. The format of the header is given in Table 17.5; the MOpCod values are given in Table 17.6. Upon a RECEIVE, the message type holds a major status, given in Table 17.7. For EoF or Fatal major statuses, the MAcCnt field indicates the actual number of bytes transferred and the MStats field holds a minor status; these minor statuses are listed with the signal values in Table 17.4.

Valid MOpCod Values

FSMessOpCode	octal	Meaning
FSOpNoOp	0	No-Operation
FSOpSeek	1	Set Current Address
FSOpSeqRead	2	Read Sequential
FSOpRanRead	3	Set Curr Addr and Read
FSOpSeqWrite	4	Write Sequential
FSOpRanWrite	5	Set Curr Addr and Write
FSOpClose	6	Close File

Table 17.6

FilSys Major Statuses

Status	octal	Meaning
FSReplOKay	1	Normal Completion
FSReplEoF	2	ErEoF status
FSReplFatal	3	Clear Error Condition

Table 17.7

If an undefined MOpCod is used, an ErOpCd error results. If the MOpCod is SWrite or RWrite and the File was opened with FOpnRd, an ErPerm error results.

For each connection from user to File, there is kept a current File address, initially zero. Upon a request with an MOpCod of Seek, RRead, or RWrite, this current address is set to the MAddr field of the message.

If the MOpCod is SRead, RRead, SWrite, or RWrite, a transmission is attempted between the message buffer part and the File, beginning at the current address. If MByCnt plus 16 exceeds either the Length of the message on an SWrite or RWrite or the BuffLength of the message on an SRead or RRead, an ErMess error is given and no bytes are transmitted; this error is considered Fatal. All SRead or RRead operations are limited by the current end of file; all SWrite or RWrite operations are limited by the physical end of file (octal 03600000 for PAGE-based Files). If the current address is not within this limit, an ErEoF error occurs with no bytes transferred. If the current address plus the MByCnt field exceeds this limit, transmission occurs up to the limit and an ErEoF error is given. Otherwise, the full transmission of MByCnt bytes takes place. If any bytes are transmitted, the current address of the File is set just past the last byte transmitted.

If the MOpCod is an FClose or a DISCONNECT is RECEIVEd, the File is closed. At some future date a DISCONNECT will be detected by FilMon and signal the close; until then, an FClose message is needed. Usage and status information in the Data Part of the File is refreshed. If the File was opened with FOpnWr, the WPfile just written supercedes the RPfile of the File and the WSema is $Ved.

17.5. FQuery

In order to inspect the usage and status stored in the Data Part of a File object, a CALL can be made on the PROCEDURE FQuery. This PROCEDURE takes two parameters. The first is a PORT prepared for output to a teletype-like device on channel 1 (&TTYPORT is an obvious candidate). The second is a File with FQuRts. The only action of the PROCEDURE is to print the fields of the Data Part in human-readable format.

17.6. FSList

One important facility of the File system is provided by the PROCEDURE FSList and the PROCESS FSpool. It happens that FSpool is started upon each boot and stands ready to spool any text File onto the line printer. The PROCEDURE FSList is provided to place an entry onto FSpool's queue of Files to be printed. FSList is called with a single File parameter which must have FRdRts. It attempts to place the current RPfile on the FSpool queue. As with FOpnRd, the ErNWrtn signal will occur if the File has no current RPFile. Also, since the FSpool queue is of finite length, the ErFull signal may occur if there is no room in the FSpool queue. Otherwise, the current RPFile will be listed eventually by FSpool. Some care has been taken, in fact, to delete the RPFile from the FSpool queue only when it has been completely listed. Thus, a system crash during the listing of the File will result in a fresh attempt to list it.

The form of the listing itself is conservative and simple. The listing is preceded by a banner indicating the first 16 printable characters of the File's FPName and the times of creation, last modification, and listing. Within the body of the File, the following conventions are observed:

1. All normal printable ASCII characters are printed in the standard way, except that the caret (octal 136) is printed as an up-arrow and the underscore (octal 137) is printed as a left-arrow.

2. All characters with bit eight set are regarded as non-text characters. They are denoted by a square box followed by whatever would be printed for the low seven characters.

3. The characters HT, LF, VT, FF, and CR (octal 11 - 15) are given their standard control functions. When nine CR's are given, however, without a LF, a LF is printed automatically to avoid tearing of the paper.

4. The characters DLE, DC1, DC2, DC3, and DC4 (octal 20 - 24) are printer control characters. Thus they are passed uninterpreted to the printer, but only when immediately following a CR.

5. The characters down-arrow through lambda (octal 01 - 10) and infinity through logical-or (octal 16 - 37; except when paragraph four applies) of the Stanford extended ASCII set are printed just as by the SOS text editor. That is, down-arrow through lambda are printed "?!" through "?(" and infinity through logical-or are printed "?)" through "?8".

6. Nulls are ignored.

7. The delete character (octal 177) is printed as "?".

18. User Software Design Considerations

This section contains a discussion of issues which have arisen in designing user software for HYDRA. An attempt is made to discuss each problem, describe the alternatives, and tell what decision was finally made. Some design issues are still open; we welcome any comments or suggestions on any of these topics.

The purposes of this chapter are twofold: to make programmers aware of the sort of problems that arise in writing user software for HYDRA, and to invite comments on the issues that have already arisen. The HYDRA user standards committee, consisting of Sam Harbison, David Lamb, Joe Newcomer, and George Robertson, is responsible for overseeing user software design for HYDRA.

18.1. Runtime LNS Allocation

Arose: November 1975
Resolved: 1 December 1975

There often arise sitiations in which a running program needs an LNS slot for some purpose, and doesn't care which slot it is. There is thus an obvious need for some sort of routine that would manage LNS slots, providing free ones and returning used ones to the free list. Moreover, there arise situations in which different parts of a program need to use the same slot (again, not caring which one it is).

One proposal was to have named LNS slots. A user would call an allocator passing it a string containing a name. If a slot by that name were already allocated, the allocator would increment a reference count and return the number of the slot. If there were no such slot, the allocator would fetch a free slot and remember the name associated with it. Slots would be freed by name; when the last reference to a slot went away, the slot would be returned to the free list.

This scheme was rejected as requiring too much overhead.

Another problem involves the addressibility of the tables used by the allocator. On school of thought said that the allocator could never presume its page of data were in core, or that the table should be kept out of core to be protected. A special page could be loaded as the allocator needed it and removed when not needed. This leads to CPS and RPS management troubles. Another possibility was to keep the information in a data object in a particular slot, without $DELETERTS rights. This still leads to unworkable overheads and complexity of programming. A lot of space is needed for storing 120 16-byte names (the proposed length), and most space management routines require rapid random-access memory for any kind of decent performance.

The current situation is that there is an allocator, the module LNSLOT[N810HY97], that provides routines to allocate and free LNS slots without names.

18.2. Internal Error Reporting

Arose: 30 November 1975
Updated: 20 January 1976

Situations arise in which library routines detect some situation they cannot correct, and so the routine detecting the error must give some sort of diagnostic message.

18.2.1 The Reporting Routine

Unfortunately, there is no guarantee of a teletype on which to report errors. If there is no device on which to report errors, it seems that the only other way of indicating an error is to do a $SUSPEND, giving some sort of negative value as an error indication. This only returns a value to a calling procedure, if any existed; processes do not return values.

Next, an error reporting routine called USERERROR was written. It took a 16-bit value as an argument. It presumed a routine called TTYPRESENT which returned a boolean which was TRUE (1) if there was a teletype on which errors could be printed. If a teletype existed it printed a diagnostic message. If there was no teletype, it checked to see if the user had a signal handling routine by checking the SIGPC address from the LCB. If such a routine existed, it faked a HYDRA signal by calling this routine. If neither of these could be tried, it did a BLISS signal in hopes that some higher level routine could do something. It also took care to avoid recursive calls.

Finally it was decided that the only appropriate thing to do was to have the user's signal handling routine take care of library errors. USERERROR was simplified to just invoke the user's signal handler.

18.2.2 The Information Reported

Currently all the information about what error has occurred is contained in a sixteen-bit integer, as with HYDRA signals. A proposed allocation of signal values was

7xxxx		For the Policy Module, Command Interpreter, and Directory Subsystem, and command objects, etc.
	71xxx	Policy module
	72xxx	Command Interpreter
	73xxx	Directory subsystem
6xxxx		User Procedures
5xxxx		User Procedures
	501xx	File System
4xxxx		Object modules
	401xx	HYDIO

Unfortunately the policy module has to be able to return any possible kernel

signal that could occur during LNS setup; the suggested range for policy module errors is far too small (the policy module ORs #160000 into any kernel signal it receives). The file system, HYDIO, and other modules continue to use the "object module signals" and "user procedures" portion of this suggestion.

A more recent proposal is to reserve a $TYPECALL index for error reporting. Given a signal number, the error reporting procedure would return a data object containing an ASCII string describing the error.

18.3. File Structure

Arose: Many Moons Ago
Resolved: 9 December 1975

The file structure debate goes back a long way; this entry just describes the most recent happenings. Currently there are two file formats on HYDRA.

FILE objects have a three-element C-list: an exclusion semaphore, a reader part, and an optional writer part. The writer part is present only when someone is writing the file. When the file is closed, the capability for the writer part is copied into the reader part. Anyone who was reading the file keeps reading the old reader part, but any new readers will get the new reader part. A reader part is a universal object whose C-list is filled with pages which contain the data. Files are read and written by sending messages to a File Monitor process.

An SOS file is a universal object with pages in its C-list. An SOS page corresponds exactly to a HYDRA page; this leads to some waste space, but very fast processing. The SOS editor knows this structure and manipulates the pages of an SOS file directly. Each line in SOS is preceded by a one-byte record count and a three-byte binary line number.

The proposal made on December 9 is that the universal object underneath the 3-part file object should be typed. There would be four subtypes initially: SOS files, TECO files, GENERAL files, and MAIL files. The $TYPECALL mechanism would be used to deal with these subfile types. All subtypes would have associated with them an EDIT procedure, an "Open for sequantial ASCII read" procedure, and a "make my kind of file from sequential ASCII text" procedure. If someone wishes to edit a file, by default he gets the editor associated with that subtype. If he wishes to use a particular editor, the command object for that editor opens the file for sequential ASCII read and calls the procedure to convert sequential ASCII text to the appropriate subtype.

The "open for READ" procedure for subtype X would take a port as a parameter and create a connection between that port and a File Monitor for type X files. The program wishing to read a type X file makes requests for input via messages on the Port System. In addition to the request for the ASCII text of a line, there would also be a request for the "header information" associated with the previous line of text. The header would be ASCII text; in the case of an SOS file the header would be the line number in decimal as an ASCII string.

All of this is inadequate for Mary Shaw's program support project. For example, in a highly structured file there may be more than one way of looking at it as a stream of ASCII text. Joe Newcomer described the "read ASCII text" sequence as the way a compiler would read a file; the program support project people cringe in horror at this notion (a compiler might well know a lot more about the structure of the file in a particular environment).

18.4. User I/O Package

Arose: Many Moons Ago
Resolved: 12 February 1976

The first set of I/O routines for HYDRA was Bill Corwin's PORTIO package. It was never intended for users; it did just what Bill needed. There was a corresponding package for the line printer.

Phil Karlton wrote a package that provided virtual channels. One would call an OPENCHAN routine, passing it a string containing the name of the device to be opened, the address of a buffer into which text would be placed, the size of this buffer, and the number of Port System messages to be created for use on that channel. OPENCHAN would allocate a free channel and return its index. There were character input and output routines, string output, numeric output, and routines to force buffers to be written out. The only devices supported at first were "TTYIN" for teletype input and "TTYOUT" for teletype output.

David Lamb took over the package to add file I/O and complete the handling of the Line Printer. There were a separate set of file handling routines: file open, file close, sequential block read, sequential block write. There were not incorporated into HYDIO for several reasons. For one, HYDIO presumes to use only one port; the file routines allow the use of several ports. Also, HYDIO is inherently sequential; it was intended that the file package eventually provide random read and write and seek operations.

On December 8, Paul Knueven, representing the ALGOL68 group, looked into the possibility of using HYDIO or some variant within the ALGOL68 compiler. HYDIO turned out to be far from desireable; built into it were far more routines than the compiler needed. Also, Paul wanted to have the compiler's I/O connections set up by a command object before the compiler ever began executing; he didn't have any use for the OPEN routines or for all of the parameter checking that the routines performed. Both Paul and George Robertson (who was interested in the file system interface for L*) were concerned with the size of the package. George said that 300 words would be too much; the file interface was over 1K words, and HYDIO was over 800 more.

The next version of the I/O package was split into several modules, in the hopes that users like ALGOL68 could use the lower levels and omit the undesireable ones. There was a module which did block transfers to and from port messages, one to do buffering, and one to open and close channels.

The current package deals with buffer control blocks. Like the previous version, it is split into several modules in hope that special applications might be able to avoid some of the routines.

18.5. Channels in User IO Routines

Arose: 16 December 1975
Resolved: 19 January 1976

The December I/O library had each routine take a channel as a parameter. This provided flexibility (and happens to be the way BLILIB and SAIL on the PDP10 does it), but involves quite a bit of extra cost at runtime in pushing the same channel number onto the stack many times. If a user is switching back and forth rapidly between channels, he will prefer this setup. If he is largely referring to only one channel, he would prefer a "set channel" routine to be invoked only as needed.

The current package uses buffer control blocks; all of the buffer manipulation routines use a blissreg1 linkage convention. Thus the compiler can often realize that register 1 already contains the desired control block address, and save code space.

18.6. I/O Routine Control Information

Arose: 16 December 1975
Updated: 12 February 1976

The December I/O library maintained an own table to control its operations. This table had to be local to the process using the routines, unless external synchronization was imposed on the routines using the table. The full-blown version required 12 bytes per channel for 16 channels, a total of 192 bytes.

An alternative to this is to reserve a few words at the head of the user-supplied buffer for this information, keeping only a pointer to the start of each buffer in the table. The table then becomes small enough (32 bytes, 2 per channel) to keep in the user communications area of the stack page, locations #000 to #177. Also, the header needed by the I/O devices could be kept in the buffer, thus requiring only a single $WRITEMSG to fill the buffer. It also means the user has to be responsible for reserving enough space for both his data and the header information, and has to be sure not to clobber the header.

In the present scheme, the I/O control information is kept in I/O buffer control blocks (see 19.2). The I/O header is kept in the few words preceding the buffer in core. A set of macros is provided to hide details from the user.

18.7. Conversion and Formatting Routines

Arose: 7 January 1976
Resolved: 14 January 1976

From time to time people write I/O formatting and conversion routines. Typically these routines are intended for output, and call the character output routine for each character they generate. For example, NUMOUT (see 22.7) and HYDTIM (see) behave this way. At some later time someone decides he needs to do the same conversion, but needs the output to go to an in-core string rather than an I/O channel.

One approach to solving this problem is to provide a pseudo-device for core-to-core "I/O"; this is the approach used by WATFIV. This usually involves kludging the I/O routines somehow. Another approach, that used by FormString (see 22.6) is for the formatting routines to take as a parameter the address of a routine to call for each character output. This has the disadvantage of clumsiness and excessive overhead.

Guy Almes suggested another approach. A formatting routine would have one parameter giving the address of a block of core where it should place its characters. It would return the address of the byte following the last character it output. Output to a channel, if desired, could be done by one call to a string output routine.

This has several advantages. It is clean and potentially quite efficient if the string output routine can be optimised; the conversion routines can use autoincrement mode on some register for outputting characters and thus could be quite efficient. Calls to formatting routines could be nested, the previous formatter providing and address for its successor. The disadvantages are that there is no way short of a BLISS signal to report errors, and no provision is made for buffer overflow.

The approach which has been adopted is to direct all formatting via buffer control blocks. The scheme is described elsewhere; see 19.1 and 19.2.

Part VII: The BLISS User Library for HYDRA

This section describes the HYDRA user library modules that are currently available. Each module describtion includes the name of the object file and the name of the programmer responsible for maintaining the module. The modules can be assumed to be independent of each other unless dependencies are explicitly mentioned.

The functions described in this section may be implemented as routines, macros, or a combination of the two. Those functions described in the module size and cross-reference summaries in chapter 23 are routines; anything not mentioned there is likely to be a macro.

The require file HYDUSR.REQ provides most of the symbols a user would need in a BLISS program. It requires the kernel call and base call macros and includes externals for most of the routines described in this document. The file HYDLIB.REQ is a subset of HYDUSR.REQ, omitting the kernel and base call macros.

The file XXXXXX.LMD is a sample linker command file which loads all of these modules. To use it, you need only change all occurences of XXXXXX to the name of your object file. HYDLIB.LMD on the same account points to the various object files. Certain of the routines have own variables which should be local to the LNS using the routines; HYDOWN points to these object modules. The standard example loads SIX12 with your program. If you omit SIX12 the linker will give warning messages about unresolved external references to the SIX12 modules; these may be safely ignored.

19. Introductory Material

19.1. Buffer Manipulation and Formatting

All formatting routines take as a parameter the address of a buffer control block which directs where they put or get characters. A buffer control block is an eight-word block of storage containing the fields

BCBminpointer	The address of the first character position in the buffer
BCBmaxpointer	An address which is one greater than the address of the last character position in the buffer
BCBputpointer	The address where the next character will be output; this ranges between the MIN pointer and the MAX pointer.
BCBgetpointer	The address from which the next character will be fetched; this ranges between the MIN pointer and the PUT pointer.
BCBroutine	The address of a routine to call when end-of-buffer is reached. This field may be zero if it is expected that the buffer will never overflow. If the end of the buffer is reached and this field is zero, the error reporting routine USERERROR (see 21.3) is called and a BLISS signal is generated.
BCBptrASM	A word of information about the buffer for the address space management routines.
BCBrtnASM	A word of information about the end-of-buffer routine for the address space management routines.
BCBextra	An eighth word reserved for future expansion (and to round things out to 8 words).

The address fields (BCBROUTINE and the fields whose names end in POINTER) are all 16-bit absolute addresses. The ASM fields are currently irrelevant; they will become meaningful when the Address Space Management routines are implemented. The phrase "end-of-buffer" is intended to include both the "no more room to output characters" and "no more characters left to read" situations.

The MIN and MAX pointers describe the physical limits of the buffer. The PUT pointer points past the last meaningful character in the buffer. The GET pointer indicates where characters should be fetched. Output formatting routines place characters beginning at the PUT pointer until they place a character just before the spot pointed to by the MAX pointer. Input conversion routines fetch characters from the GET pointer until they attempt to read the character pointed to by the PUT pointer. At these points they call the end-of-buffer routine, passing it the address of the buffer control block. By convention, the end-of-buffer routine will either reset the pointers so that more information can be output or input, or give a BLISS signal to blast out of the formatting routine. The latter is done so that formatting routines need not check for errors detected by the end-of-buffer routine. As noted above, if the end-of-buffer routine address is zero, a call to USERERROR and a BLISS signal are performed. At end-of-input, an input buffer flush routine will typically fill or partially fill the buffer with characters, set the GET pointer to the beginning of the buffer, and set the PUT pointer just pass the last character it placed in the buffer (this is why we need four different pointers, since on input the text may not fill the buffer). At end-of-output, an output buffer flush routine will typically reset both the PUT and the GET pointers to the beginning of the buffer.

To simplify the coding of routines which manipulate buffers and buffer control blocks, all routines manipulating BCBs assume that they need only make one call to the ASM make-addrressable function in order to be able to access the entire buffer throughout the rest of the routine. Thus one of two things must be true of your buffer when you call a BCB-manipulating routine. If the page containing the buffer is

not in the RPS when such a routine is entered, then the buffer must not cross a page boundary. If the pages containing the buffer are in the RPS, then the buffer may cross page boundaries. Because of this restriction, only one ASM field is needed to describe the buffer, and the end-of-buffer check can therefore be made by a comparison of two addresses. Both signed and unsigned comparisons will work equally well if the buffers do not cross page boundaries and do not abut the end of page 3 (for signed comparison) or page 7 (for unsigned comparisons). The library routines all use unsigned comparisons; thus the only difficulty arises if the last byte of a buffer is the last byte of page 7.

A user can declare a BCB by saying

```
LOCAL bcb X:Y:Z;
OWN bcb W;
```

which will declare BCBs X, Y, and Z on the stack and W as an own BCB. Alternately, you can allocate a BCB from any eight word block of storage you have available. Note that BCBs are word-oriented; it makes no sense to declare a byte local BCB, although nothing will go wrong if you do so.

Each of the above fields may be accessed by name via a structure access. Thus

```
.X[BCBputpointer]
```

is the contents of the PUT pointer field of a BCB whose name is X (such as the X declared in the above local declaration). In addition there are macros to initialize BCBs and compute related information such as the length of the buffer. If you have some arbitrary expression which evaluates to the address of a BCB, you may access its fields by saying

```
.BCB[expr; BCBfieldname]
```

where EXPR is the expression and BCBFIELDNAME is one of the names given in the above table.

If you wish to build a more complicated structure on top of this mechanism, you may do so by appending extra control information to the end of the BCB. For example, the I/O system has its own port and channel information following an associated BCB. The end-of-buffer routine for an I/O BCB is either the input-from-channel or output-to-channel routine. OPENCHAN takes care of setting up this information (see 20.2). If you wish to declare a BCB with extra words for your own purposes, you may do so by saying

```
LOCAL bcb x[N+1];
```

where N is the number of extra words you wish to allocate. The "+1" is needed for reasons to do with BLISS default values for omitted incarnation actuals (take my word for it, you don't want to know). You may access fields in your extension by saying

```
.X[BCBextra(i,j,k)]
```

where I is the index of the extra field (your first extra word is 1, your second is 2, and so on), and J and K are BLISS position-and-size field specifiers.

The basic formatting routines (see 20.3), numeric output (see 22.7), and date/time conversion (see 20.5) are described in later sections of this chapter. All BCB-related routines are called with the blissreg1 calling convention: the first parameter, the BCB, is passed in register 1. You need not normally be overly upset by this, as the require file HYDLIB.REQ has all the necessary declarations. HYDLIB.REQ is required by the standard file HYDUSR.REQ.

19.2. The I/O Library

The modules IOTRAN (22.4), MAKMSG (22.3), OPENCH (20.2), and OPENIT (20.1) form a preliminary input/output library, referred to collectively as HYDIO. The lowest level, IOTRAN, does not care how I/O connections were established so that in fact the connections could have been set up before the procedure began execution. (No other support for this possibility is implemented yet, however; it is still necessary to use OPENCHAN.)

Bufferring of I/O is done using the standard BCB manipulation package (see 19.1). I/O is controlled by I/O buffer control blocks, the first eight words of which are a buffer control block as described in 19.1. You may declare an I/O control block by saying

```
LOCAL IOBCB X:Y;  OWN IOBCB Z:W;
```

or

```
LOCAL X[IOBCBSIZE],Y[IOBCBSIZE];
OWN Z[IOBCBSIZE],W[IOBCBSIZE];
```

both of which are equivalent. You must then use BCBINIT to point your IOBCB at a buffer. The I/O routines assume they can use a few words immediately preceding the buffer for their own purposes. The number of words needed varies with the device to which the buffer is being sent, but is never more than the number of words specified by the symbol MAXBUFFER. Thus you can set up a buffer for input by saying

```
LOCAL iobcb x;  LOCAL buf[MAXBUFFER+mybuffersize];
bcbinit(x,buf[MAXBUFFER],mybuffersize,BCBinput,0);
...
```

Alternately, if you wish to skip the messy details and just allocate a buffer for I/O, you may say

```
LOCAL txbuffer mybuffer[parm1,parm2];
 ...
```

```
txbinit(mybuffer);
 ...
```

PARM1 is the number of words in the buffer; PARM2 is the number of words immediately preceding the buffer needed by routines manipulating the buffer. This parameter varies depending on the use to which you will put the buffer. Symbolic names for various sizes, and their intended purpose, are described below.

textbuffer	0	The buffer is intended purely for manipulating text and will never be used for I/O.
shortbuffer	2	The buffer is intended purely for the teletype, the physical line printer, or other devices requiring a two-word header. In particular, it will not be used for files.
filebuffer	8	The buffer is intended for I/O to line printer, teletype, spooler, or files.
maxbuffer	8	This is the largest header needed.

Thus one can declare a buffer intended for files via

```
LOCAL txbuffer fbuf[StdBufSize,FileBuffer];
txbinit(fbuf)
```

Note that TXBINIT is a macro that depends upon being able to get at the incarnation actuals of the TXBUFFER (i.e., STDBUFSIZE and FILEBUFFER in the above example). Its argument must therefore be the name you used for your variable in the TXBUFFER declaration, and CANNOT be an expression. It is provided to hide the details of buffer and BCB setup. You must call OPENCHAN to associate an I/O channel with the IOBCB or TXBUFFER. At this point you may use all of the bufferring and formatting routines, passing them the address of the IOBCB or TXBUFFER as the first parameter.

Many of the routines indicate errors by returning negative numbers as status indicators. All the error codes have #140100 as a base; the bottom two octal digits indicate the error. The following table describes the errors briefly. All numbers are octal.

BadChannel	#20	The channel was outside the range 0-15 (or sometimes -1 to 15; the range should be clear from context)
NotOpen	#21	The port/channel combination is not connected to any device.
ChanStop	#22	An error or end-of-file has occurred on the port/channel; no further activity may take place. CLOSE is the only valid operation.
BadDirection	#23	An input operation has been attempted on an output device, or vice versa.
BadMode	#24	This message comes from OPENFILE. The MODE parameter to this routine was not 0, 1, or 2 (read, write, update).
NoSlot	#25	A routine was unable to allocate an LNS slot for internal use.
TooManyMsg	#26	An attempt was made to create too many (or too large) port messages; the port had insufficient resources.
BadPort	#27	The port parameter was outside the range of valid LNS slots.
NoSuchDev	#30	The DEV parameter to OPENCHAN was illegal.
BadDev	#31	Invalid operation for this device.
ChanEOF	#32	End-of-file has been reached.
NoSuchFile	#33	OPENIT was unable to find an input file or create an output file.
ProcLookUp	#34	A routine was unable to get a Procedure object it needed from SYSDIRECTORY.
LookUp	#35	A routine was unable to get some object (other than a procedure) from SYSDIRECTORY.
CantList	#36	CLOSECHAN was unable to list a spooled file.
NoFlushRtn	#37	An attempt was made to call a nonexistent buffer flush routine.

Error numbers less than #20 are reply types issued by the I/O devices to indicate error conditions. These error codes are defined in UIO.REQ[N810HY97] and described in the HYDRA reference manual.

19.2.1 IOSTATUS(IOBCB)

Returns the status of the I/O connection associated with the IOBCB. The return codes are

IOnormstat	0	Normal status
IOBCBbad	1	Not a valid IOBCB
IOeof	2	End-of-file has been reached
IOerr	3	A nonstandard error has occurred

19.3. Generalized Formatting -- the FormString Convention

Defined below is a convention for producing character strings. Routines which implement this convention have been implemented for many PDP-11 environments and may be easily adapted to almost any PDP-11 application. A source require file (FORMST.R11[N810RG02]/A) is available for helping in other implementations. In addition, a module (FrmStr) which may be used in almost any environment with the addition of a small interface routine is documented below (cf. 22.6).

19.3.1 Introduction

Many programs require a facility for generating character strings from other character strings, integers, and other sorts of data which can be converted to a character string representation. A good example of this is output, where results must be represented in a form easily understood by a human. Although this is not the only useful application of a string production facility, it is so common that speaking of string production as "printing" can make documentation much easier to follow. For this reason, the remainder of this document will refer to "printing" a string, when in fact the facility has been found useful in applications which in no way resemble output.

Several different modules, documented elsewhere, use a specific convention for producing character strings. We shall attempt here to define that convention, known as the "FormString" convention. Reasons for using this convention include easily read source code, compact object code, and consistency with a single standard thereby easing learning time.

19.3.2 Rough description of the routines

Routines which meet the FormString convention may take any arguments. It is essential, however, that the following be included:

1) a formatting control string (which we shall call simply a "format"), which is a string of characters which act as a skeleton or format for the string produced. (Although all existing routines define the format to be a BLISS-11 asciz string, the method of delimiting the string is not so constrained by this standard.) The format contains literal text and "control codes" which are described below (cf. 19.3.3).

2) optional extra arguments (referred to as ARGs) of arbitrary type. The number and type of thses arguments will depend on the control codes present in the format. This standard makes no attempt to define what happens if the wrong type or number of ARGs is provided. So far, the BLISS-11 implementations have obviously been blind to type errors (other than causing NXM errors or such). They ignore extra ARGs and simply pick up whatever is next in memory if too few are provided.

Since we do not here specify the exact calling sequence of routines, we shall denote calling sequences as:

"Format String", Arg1,Arg2,...,Argn

Actual calling sequences will generally include some other arguments, such as some sort of destination for the characters produced.

Since the effect of such an argument list is usually clear from the effect of its components, it will usually be sufficient to document each component separately. For this reason,

"...foo...", Arg1,Arg2,...,Argi,...,Argn

(where ARGi is used by a control code in "foo") will be abbreviated to just

"foo", Argi

Any routine which implements the FormString convention must be recursively reentrant (e.g. callable from an &*@ routine as defined in section 19.3.7) and must return as its value (in the BLISS-11 sense) the number of characters printed.

19.3.3 Formatting control strings

As mentioned above, a format is a character string comprised of literal text and control codes. All of the characters are assumed to be literal text unless they match the syntax of a legal control code. Literal text is printed as it is encountered.

The left end of a control code is always delimited by the "escape" character (currently "&"); the right end is determined by the syntax of control codes, which is:

&<char> or &<#><char>

where <char> is a single character and <#> is a string of digits forming an unsigned decimal number (or the letter "n"). The character <char> determines the action taken when the control code is encountered in the format string. If alphabetic, <char> may be either upper or lower case with no change in meaning.

An example of a simple format is "foo = &D, bar = *&O", which contains two control codes. The former prints its ARG in decimal, the latter in octal. Thus,

"foo␣=␣&D,␣bar␣=␣*&O",47,33 prints "foo␣=␣47,␣bar␣=␣*41"

Control codes of the form &<#><char> allow one to pass some extra information to the routine which performs the function indicated by that control code. For example, "&3D" will cause a number to be printed using at least 3 digits. If fewer are required, leading zeroes will be added. If more than 3 are required to express the number, however, none of those digits will be truncated -- they will all be printed.

Control codes are divided into two groups: those which expect a <#> and those which do not. If a <#> is included with a code which does not expect one, the entire control code will go unrecognized. If, on the other hand, a control code expects a <#> and one is not provided, a default value of 1 will be assumed. This means that our previous use of &D was really equivalent to "&1D".

Sometimes we do not know at the time we create the format control string (e.g. compile time if it is a plit) what value we would like to give to <#>. In this case we use the letter "n" for <#>. This will cause the extra information to be fetched from the argument list. Thus,

"&n<char>", 3+6, 22 is equivalent to **"&9<char>", 22**

Obviously the 3+6 in this example may be replaced with any arbitrary expression whose value is desired for <#>.

19.3.4 Simple Control Codes

In the following list of the available control codes we will differentiate those codes which expect a <#> by listing them as &#<char>. Remember that the default value, if omitted, is 1.

&& causes a single "&" to be printed.

&#D prints ARG as a decimal integer using at least <#> digits, padding with zeroes if needed. (For blank padding see 19.3.6.) Note that a "-", if produced, is considered a digit for purposes of counting the digits.

&#UD is like &#D except that the number printed is considered to be unsigned, i.e. in the range 0-65535. Note that in this case <char> is actually two characters ("UD") rather than one as specified in the syntax.

&#O is like &#D but octal.

&#UO is like &#UD but octal. Note that &6UO is a commonly used control code.

&#B is like &#D but binary.

&#UB is like &#UD but binary.

&#H is like &#D but hexadecimal.

&#UH is like &#UD but hexadecimal.

&L is currently equivalent to &6UO in all implementations, but is intended to be used to print ARG as a symbolic location (using some sort of symbol table or such). For instance, in a BLISS-11 error trap routine one might use **"Error at PC = &L", .OldPC** to type out where the trap occurred.

&C prints the character with the code (ARG and #377). E.g. **"&C", #445** prints "%".

&P is like &C, but if the character is not a printing character a "." is substituted. (The ASCII values #40 through #176 are considered printing characters.) In addition, the mask used is #177 rather than #377. This control code is principally useful in debugging output, such as dumps, where it is not known whether the character being printed is a printing character.

&#A prints the <#> characters pointed to by ARG. Note that &1A (or &A, since <#> defaults to 1) is similar to &C, except that ARG is a pointer to the character rather than the character itself.

&Z prints an asciz string pointed to by ARG. It is thus like &#A except that instead of taking a character count it prints until it finds a null (0) character.

&#/ (slash) prints <#> CarriageReturn-LineFeed combinations. Note that &/, even without a count, can be useful in a BLISS-11 string (instead of ?M?J) since it is a little easier to read (and to type in SOS!!).

&#X prints <#> space characters.

&#N null operation -- prints nothing. Note that &nN may be used to skip over one ARG in the argument list. (This is marginally useful in conditionals and such -- see 19.3.5.) Note also that &N (without an explicit <#>) cannot be used. This is because &n<text> will be seen as a control code with <#> fetched from the argument list and the first character of <text> treated as <char>.

19.3.5 Loops and Conditionals

A primitive looping facility has been provided in FormString. There are actually two types of loops:

&#<...&> and &#(...&)

These constructs, or any mixture thereof, must be well balanced in the usual manner.

The loop constructs provide that everything inside the loop will be interpreted <#> times.

```
"&7<_&0D&>",5,4,3,2,1,0,1      prints      "_5_4_3_2_1__1"
```

&#(...&) differs from &#<....&> in that the argument list pointer will be reset at the beginning of each iteration. Thus,

```
"&7(_&0d&)",5,4,3,2,1,0,1      prints      "_5_5_5_5_5_5_5"
```

and the other six arguments will be "left over" for any future control codes which may follow. Furthermore, if <#> is zero, &0<...&> will cause the intervening control codes and text to be completely ignored while &0(...&) will cause the arguments to be skipped which would have been used had <#> been non-zero. In other words, &#<...&> will use up <#>*n arguments and &#(...&) will use up 1*n arguments (where n is the number of arguments which are used by the loop body).

It should be noted that if one only passes 0 and 1 as values for the loop count, &n(...&) and &n<...&> may be used effectively as conditional constructs.

To make this even more convenient, &| (vertical bar) may be used as an "else". That is, "&|" is equivalent to "&>&#<" or "&)&#(" (depending on which type of conditional it is in) where <#> is given the value:

```
(if <#> of current loop eql 0 then 1 else 0)
```

If more than &| appears in the text, each one will effectively cause complementation of the condition. That is,

```
"&n<x&|y&|z&>",1      prints      "xz"
"&n<x&|y&|z&>",0      prints      "y"
```

19.3.6 Right Justification

An obvious question on encountering the definition of &*D above is, "What if I want my number padded with blanks, not zeroes?" This has been answered by a general right justification mechanism which may be used on any combination of literal text and control codes, rather than just numbers. The construct "&*W" indicates that the following text should be right justified in a field <#> characters wide. The text which follows is delimited by the next control code after the &*W (inclusively). Note, however, that for this purpose "&&" is not considered a control code. In addition, the sequences "&#(...&)" and "&#(...&)" are considered to be a single delimiter. That is, the right justified text includes any text produced inside the loop. Thus "&(...&)" and "&<...&>" (with <#> defaulting to 1) may be used simply to bracket text to be right justified which includes several control codes. If the text to be right justified is wider than the field, it will be printed with no extra leading space characters. Otherwise enough leading space characters will be printed ahead of the text to cause it to be appropriately right justified. Some examples are:

```
"&5W&D",-3      prints      "___-3"
```

```
"&5W&3D",-3              prints      "␣␣-03"
"&5W&(&D,&D&)",1,2       prints      "␣␣1,2"
```

19.3.7 Extensions to the control codes

The control code "&*@" is used to provide extensions to the control codes. ARG is interpreted as the address of a routine to be called. The exact calling sequence is not defined by the FormString convention. As a minimum, however, two other arguments will be required: a count of how many characters have been printed so far, and the value <*> (which still defaults to 1 if missing).

The calling sequence must provide some way for the called routine to fetch successive ARGs from the list of extra arguments. A typical way to do this is to pass an argument which is the address of a pointer to the argument list. This means an extra dot is needed when accessing elememts from the argument list. If the routine needs any ARGs from the argument list, it should do the following:

```
value ← ...ArgListPtr;
.ArgListPtr ← ..ArgListPtr + 2;
```

where ARGLISTPTR is the name of the argument containing the pointer to the argument list pointer.

It is important that the routine not randomly clobber the argument list pointer since it will be used to fetch any remaining arguments needed from the argument list. The only side effects of the routine should be the bumping of the location pointed to by ARGLISTPTR or the printing of characters; its value should be the number of characters that it prints. Any other side effects may cause improper operation of zero iteration loops and right justification.

19.3.8 Buffer flushing

When used for producing output to a device such as a terminal, it may be desirable to cause any buffered characters to be forced out to the terminal. The control code "&$" is reserved for this purpose. It does not affect the number of characters printed so far and is not considered a delimiter by &*W. If encountered in a zero iteration loop, it, like all other text and control codes in the loop, will have no effect.

In situations not oriented to output, this control code probably has no meaning and so should be ignored.

20. Input, Output, and Formatting

20.1. Initiating and Terminating I/O

Module OPENIT Maintainer: David Lamb

The module OPENIT provides routines somewhat higher-level than those of OPENCH. It is part of the HYDIO package. LNS slot 3, given the symbolic name USERDIRECTORY, must contain a capability for the directory to be searched for files.

The routines in this module use the blissregl linkage convention, as do all routines taking a buffer control block as a parameter (see 19.1). You need not normally be aware of this convention, since the require file HYDLIB.REQ, which is part of HYDUSR.REQ, contains all of the necessary declarations.

20.1.1 OPENIT(IOBCB,PORT,CHAN,DEV,INFLAG,NUM)

Internal name CHA003

IOBCB is the address of an input/output buffer control block (see 19.2). DEV is the address of an asciz string containing an I/O device specification. INFLAG is odd to open for input and even for output. NUM is a small integer, the number of port messages to be created for I/O to or from this device. The parameters PORT, CHAN, INFLAG, and NUM are passed untouched to OPENCHAN. DEV is interpreted by OPENIT to provide the DEV and FSLOT parameters of OPENCHAN.

OPENIT calls the string library routines EQU, STRINDEX , and GETARG, the I/O initialization routine OPENCHAN, the directory routines DIRNAMES, GETDIRECTORY, and PUTDIRECTORY, the LNS allocation routines ALLOSLOT and FREESLOT, and the routines USERERROR, SETSIGNAL, and MAKEFILE.

The principal use of OPENIT is to open files. If DEV points to a string which does not contain the character ":", OPENIT will interpret the string as a directory name and will look up the file in USERDIRECTORY and call OPENCHAN to open the file. If the file does not exist, and the file is to be opened for writing, OPENIT will call MAKEFILE to create a file and will place the result in the named directory position.

Alternately, DEV may contain the string "TTY:", "LPT:", or "PRN:". These three strings represent the teletype, spooled printer output, and the physical printer.

20.1.2 CLOSEIT(IOBCB)

Internal name CHA004

Closes a device opened by OPENIT. CLOSEIT calls CLOSECHAN and FREESLOT.

This page intentionally left blank

20.2. Basic I/O Initiation and Termination

Module OPENCH Maintainer: David Lamb

The module OPENCH contains routines for opening and closing channels for input and output. It is part of HYDIO. The routines in this module are fairly low-level, suitable only for the teletype and line printer. Files generally require more complicated routines, although files can be opened with OPENCHAN. See 20.1 for a description of the higher-level open.

The routines in this module use the blissreg1 linkage convention, as do all routines taking a buffer control block as a parameter (see 19.1). You need not normally be aware of this convention, since the require file HYDLIB.REQ, which is part of HYDUSR.REQ, contains all of the necessary declarations.

20.2.1 OPENCHAN(IOBCB,PORT,CHAN,DEV,IN,NUM,SLOT)

Internal name CHA001

IOBCB is the address of an input/output buffer control block which has already been initialized with BCBINIT or TXBINIT (see 19.2). PORT is the index of an LNS slot containing a port capability. CHAN is an integer in the range 0-15, naming a free channel on the port through which I/O will take place, or is -1 in which case OPENCHAN will find a free channel. DEV and NUM are small positive integers. IN is a boolean value. SLOT is the LNS slot index of a file capability, indicating the file to be opened, if DEV indicates that a file is to be opened; otherwise it is ignored. This routine creates a connection to a device of type DEV through the indicated port and channel and creates NUM port messages for I/O for this device. If IN is odd (for true) the device is opened for input; otherwise it is opened for output. You may use the symbols OPENINPUT and OPENOUTPUT to specify the mode in which you wish to do the open. The block of storage described by the buffer control block portion of IOBCB will be used for internal buffering for the device. On normal exit, all the information needed to do I/O to the device is stored in the IOBCB. You may proceed to use any of the string routines to read or write information to the device (see 19.1). OPENCHAN normally returns the number of the channel; a negative value indicates an error.

Up to 16 channels may be opened simultaneously on a single port. The most common use for multiple open connections is probably to have several files open for various purposes.

Users of the HYDUSR package should use the symbol IOPORT to specify the PORT parameter, and -1 for the CHAN. IOPORT is defined in the require file HYDLNS.REQ, which is part of the standard file HYDUSR.REQ. Bear in mind that for HYDUSR, channels 0 and 1 are pre-allocated to the teletype and may not be otherwise used.

Symbolic names for the device indices are provided as binds in the standard require file, HYDLIB.REQ. The devices currently supported are

DevTTY	1	teletype
DevLPT	2	spooled line printer
DevFile	3	sequential file
DevPRN	4	physical line printer

OPENCHAN calls the LNS slot allocation routines ALLOSLOT and FREESLOT, the I/O initiation routines OPENFILE and OPENLPT, the low-level port routines INMESSAGE, OUTMESSAGE, and MAKEMESSAGE, and the error reporting routine USERERROR.

20.2.2 CLOSECHAN(IOBCB)
Internal name CHA002

IOBCB is the address of an input/output buffer control block (see 19.2). Terminates I/O on the specified channel, deletes the associated messages, and disconnects the channel. It returns negative number if an error occurs and zero or a positive number under normal circumstances. It calls the routines CLOSELPT, CLOSEFILE, LISTFILE, FREESLOT, ACCEPT, and KILLMESSAGE.

20.2.3 OPENTTY(IOBCB,PORT,CHAN,DEV,IN,NUM,SLOT)
Internal name CHA001

This is a version of OPENCHAN in which only the teletype may be opened. If one loads OPENTTY instead of OPENCHAN, one can acheive a reasonable saving in code space by eliminating the routines called by OPENCHAN and not by OPENTTY.

OPENTTY calls MAKEMESSAGE and USERERROR.

20.2.4 CLOSETTY(IOBCB)
Internal name CHA002

This is a version of CLOSECHAN in which only the teletype may be closed. It calls KILLMESSAGE.

20.3. Basic Buffer Functions

Module BUFFER Maintainer: David Lamb

The module BUFFER contains the basic routines for manipulating buffer control blocks. All of the routines in this module take the address of a buffer control block or I/O BCB as the first parameter, and use the blissreg1 linkage convention, as do all routines taking a buffer control block as a parameter (see 19.1). You need not normally be aware of this convention, since the require file HYDLIB.REQ, which is part of HYDUSR.REQ, contains all of the necessary declarations.

A number of functions described herein are implemented via macros rather than routines, as they involve only a few instructions. The macros are defined in BCB.REQ, which is part of HYDLIB.REQ.

20.3.1 BCBINIT(BCB,BUFFER,SIZE,FLAG,ROUTINE)

Initializes the BCB to point to a block of SIZE bytes beginning at location BUFFER. If FLAG is odd, the buffer is intended for input; otherwise the buffer is intended for output. ROUTINE will be called when the end of the buffer is reached, if ROUTINE is nonzero. Otherwise, a call to USERERROR and a BLISS signal will occur. If the buffer is intended for I/O, the ROUTINE parameter may be zero, as OPENCHAN will initialize the ROUTINE parameter (see 20.2).

20.3.2 BCBFLUSH(BCB)

Calls the end-of-buffer routine associated with the BCB. If the end-of-buffer routine is zero, calls USERERROR and generates a BLISS signal.

20.3.3 BCBRESET(BCB,FLAG)

Resets pointers in the BCB. If FLAG is odd, performs an input reset: the GET pointer is set back to the start of the buffer. If FLAG is even, performs an output reset: both the GET and the PUT pointers are set back to the start of the buffer.

20.3.4 BCBINCOUNT(BCB)

Returns the number of characters between the MIN and GET pointers, i.e. the number of characters which have already been read.

20.3.5 BCBOUTCOUNT(BCB)

Returns the number of characters between the MIN and PUT pointers, i.e. the number of characters which have already been written.

20.3.6 BCBINLEFT(BCB)

Returns the number of characters between the GET and PUT pointers, i.e. the number of characters left to be read in the buffer.

20.3.7 BCBOUTLEFT(BCB)

Returns the number of characters between the PUT and the MAX pointers, i.e. the number of characters which may be output before end-of-buffer is reached.

20.3.8 BCBINMOVE(BCB,OFFSET)

Moves the GET pointer forward or backward OFFSET positions, provided that the result is a valid input position. The direction is determined by the sign of OFFSET, forward if positive and backward if negative. Attempting to move the GET pointer below the MIN pointer sets the GET pointer to the MIN pointer; attempting to move it beyond the PUT pointer sets it to the PUT pointer.

20.3.9 BCBOUTMOVE(BCB,OFFSET)

Moves the PUT pointer forward or backward OFFSET positions, provided that the result is a valid output position. The direction is determined by the sign of OFFSET, forward if positive and backward if negative. Attempting to move the PUT pointer below the MIN pointer sets the PUT pointer to the MIN pointer; attempting to move above the MAX pointer sets it to the MAX pointer.

20.3.10 BCBINSET(BCB,COLUMN)

Set the GET pointer to the named column (the first column being numbered zero), provided it is a valid input column. Attempting to move the GET pointer below the MIN pointer sets the GET pointer to the MIN pointer; attempting to move it beyond the PUT pointer sets it to the PUT pointer.

20.3.11 BCBOUTSET(BCB,COLUMN)

Set the current output position to the named column (the first column being numbered zero), provided it is a valid output column. Attempting to move the PUT pointer below the MIN pointer sets the PUT pointer to the MIN pointer; attempting to move above the MAX pointer sets it to the MAX pointer.

20.3.12 OUTCHR(BCB,BYTE)

Appends the byte to the buffer. If the buffer is full after it appends the byte, it calls BCBFLUSH.

20.3.13 INCHR(BCB)

Reads a byte from the buffer. If there are no bytes left in the buffer when INCHR tries to fetch one, it calls BCBFLUSH.

20.3.14 OUTSTR(BCB,STR)

STR is the address of an asciz string. Copies the string to the buffer. The high order bit of each byte is copied untouched. Note that BCBFLUSH might be called at arbitrary points in the middle of the transfer.

20.3.15 OUTBLOCK(BCB,BLOCK,COUNT)

BLOCK is the address of a storage area of COUNT bytes. Copies the COUNT bytes from BLOCK to the buffer. The high order bit of each byte is copied untouched. Note that BCBFLUSH might be called at arbitrary points in the middle of the transfer.

20.3.16 GETLINE(BCB,BLOCK,COUNT)

BLOCK is the address of a storage area of at least COUNT bytes. Copies characters from the buffer to the block until a linefeed is seen or COUNT-1 bytes have been transferred (The linefeed, if present, is also copied). A trailing null is written at the end of the transferred string. The high order bit of each byte is copied untouched. Note that BCBFLUSH might be called at arbitrary points in the middle of the transfer.

20.3.17 OUTCRLF(BCB)

Write a carriage return and linefeed to the indicated buffer. Calls OUTCHR.

20.4. Numeric Output

Module NUMFMT Maintainer: David Lamb

The module NUMFMT contains functions for numeric output. The functions in this module, except NUMSTRING, use the blissreg1 linkage convention, as do all functions taking a buffer control block as a parameter (see 19.1). You need not normally be aware of this convention, since the require file HYDLIB.REQ, which is part of HYDUSR.REQ, contains all of the necessary declarations.

20.4.1 NUMSTRING(BUFFER,VAL,BASE)

Internal name NUM001

BUFFER is the address of a block of storage (NOT a BCB!). VAL is an arbitrary 16-bit integer. BASE is a number assumed to be in the range 2-36. Converts the given value, considered to be an unsigned 16-bit integer, to a character string representing the number in the given base. Characters are placed starting at BUFFER in the order the digits are generated, which is the REVERSE of the order they should be printed. Character placement stops before the generation of the first leading zero; thus the number zero will output no digits. Returns the position where the leading zero would have been placed; thus <value returned> minus BUFFER gives the number of characters generated.

This routine is not intended for the average user; it does the hard part of number conversion, and is intended to be called by higher level conversion routines. It calls the unsigned divide routine DIVUNS.

20.4.2 OUTNUM(BCB,VAL,BASE,WIDTH)

A routine for general purpose output of integers in arbitrary bases (2 to 36 inclusive). The integer VAL is converted to ASCII characters in base BASE and written to the buffer. The absolute value of WIDTH is the minimum size of the number in characters. A WIDTH of less than zero will add leading zeros if necessary; a WIDTH greater than zero adds leading blanks; a WIDTH of zero is equivalent to a WIDTH of one. For bases larger than 10, the letters "A" through "Z" are used for the extra digits. If the number is negative, a minus sign is written just preceding the first nonblank position; the minus sign consumes one place of the required WIDTH. Calls NUMSTRING and OUTCHR.

20.4.3 OUTOU(BCB,VAL)

VAL is written to the buffer as an unsigned six-digit octal number with leading zeroes. This is especially useful for writing out addresses, signal values and bit masks. It calls NUMSTRING and OUTCHR.

This page intentionally left blank

20.5. Clock Manipulation and Output

Module TIME Maintainer: David Lamb

The module TIME contains routines for manipulating four-word clock values. In particular, there are routines for outputting the clock values as date and time. There is also a lower-level routine for doing the computation needed to convert clock values to date and time.

Two time scales are used with these routines: the master clock time scale followed by the hardware, and a scale where midnight (00:00) the morning of January 1, 1964 is the zero clock value. The routines CALCTIME, ZCALCTIME, GETCLK, and ZGETCLK can be remembered thus: GETCLK does a $GETCLOCK kernel call to fetch a clock value; ZGETCLK calls GETCLK and the adjusts the result to the January 1, 1964 time scale. CALCTIME takes a clock value as its argument; ZCALCTIME takes a corrected clock value. GETCLK should be remembered by its similarity to the kernel call. One calls CALCTIME with the result of GETCLK, and ZCALCTIME with the result of ZGETCLK.

Those routines which format dates and times for output take the address of a Buffer Control Block as their first parameter. They follow the blissreg1 linkage convention, wherein the first argument is passed in register 1. The standard require file HYDUSR.REQ has all the declarations necessary to accomplish the proper linkage.

20.5.1 CALCTIME(RESULT,CLOCK)

CLOCK is the address of a four-word uncorrected clock value. RESULT is the address of a vector large enough to hold all the fields of the result computation. The file TIME.REQ contains definitions of symbols to be used in accessing fields of the result. The field names are summarized in the following table.

TimeYear	number of years since the base year
TimeLeapYear	true if the year is a leap year
TimeDayOfWeek	Sunday = 0, Saturday = 6
TimeDayOfMonth	0-30
TimeDayOfYear	0-365
TimeMonth	0-11
TimeHour	0-23
TimeMinute	0-59
TimeSecond	0-59
TimeSixtieth	0-59
TimeMicrosecond	0-999
TimeMillisecond	0-999
TimeVecSize	Number of words needed for this vector

As conventions for printout of dates and time vary greatly, this low-level routine is provided for users who wish to dispense with the standard output routines and write their own.

CALCTIME calls ZCALCTIME and CLOCKADJUST .

20.5.2 ZCALCTIME(RESULT,CLOCK)

CLOCK is the address of a four-word block containing a corrected clock value. RESULT is as for CALCTIME. Behaves exactly as does CALCTIME, save for its convention for the clock argument (in fact, it is called by CALCTIME). ZCALCTIME calls the four-word arithmetic package and DIVUNS.

20.5.3 CLOCKADJUST(CLOCK)

CLOCK is the address of a four-word master clock value. Corrects the clock value to the January 1, 1964 scale.

20.5.4 GETCLK(CLOCK)

CLOCK is the address of a four word block. Places in the addressed block the current time in master clock format (least significant word first), with the processor bits zeroed.

20.5.5 ZGETCLK(CLOCK)

CLOCK is the address of a four word block. Places in CLOCK the current time, adjusted for the January 1, 1964 time scale. ZGETCLK calls GETCLK.

20.5.6 OUTMS(BCB,CLOCK)

BCB is the address of a buffer control block. CLOCK is the address of a four-word clock value. CLOCK is interpreted as a number of microseconds, and printed as seconds in the form "s.mmm" (three places after the decimal point). It is assumed there are less than 32768 seconds. This routine is intended for printing the differences between two clock values.

OUTMS calls OUTNUM, OUTCHR, and the four-word arithmetic package.

20.5.7 OUTDATE(BCB,CLOCKVEC)

BCB is the address of a buffer control block. CLOCKVEC is the address of a vector holding the result of a call to CALCTIME. Prints the date in the form "DD MMM YY" with no extra leading or trailing blanks. If the year should happen to be less than 10, it is given a leading zero. If the day should be less than 10, it is given a leading blank. Months are abbreviated to the first three letters. OUTDATE calls TIMEDATE.

20.5.8 OUTTIME(BCB,CLOCKVEC)

BCB is the address of a buffer control block. CLOCKVEC is the address of a vector holding the result of a call to CALCTIME. Prints the time in the form "HH:MM:SS" with no extra leading or trailing blanks. HH is printed as a 24-hour clock time. A leading zero is provided for each of HH, MM, or SS that is less than 10 in value. OUTTIME calls TIMEDATE.

20.5.9 OUTDT(BCB,CLOCKVEC)

BCB is the address of a buffer control block. CLOCKVEC is the address of a vector holding the result of a call to CALCTIME. Outputs time and date, in that order, in the format used by OUTTIME and OUTDATE, separated by a blank. OUTDT calls TIMEDATE.

20.5.10 OUTGDT(BCB,CLOCK)

BCB is the address of a buffer control block. CLOCK is the address of a four-word uncorrected clock value. Outputs the time and date derived from this clock value via OUTDT.

20.5.11 OUTCDT(BCB)

BCB is the address of a buffer control block. Outputs the current date and time in the form used by OUTDT. Calls OUTGDT.

20.5.12 TIMEDATE(BCB,CLOCKVEC,MASK)

Internal name TIM004

BCB is the address of a buffer control block. CLOCKVEC is the address of a vector set by CALCTIME. MASK is a 16-bit mask specifying control information. TIMEDATE prints the date and time information held in CLOCKVEC as directed by MASK.

MASK consists of a number of one- and two-bit fields. The file TIME.REQ provides symbolic names for these fields and for values which can be held in these fields. The following table summarizes the field names and values they may hold.

OTimeOrder	Specifies the order of the time and date portions
SetDateFirst	
SetTimeFirst	
SetNoDate	
SetNoTime	

OTimeWeekday	Direct printing of the day of the week.
SetNoWeekday	
Set3Weekday	print the first three letters of the day of the week, followed by a blank.
SetFullWeekday	print the full name of the day of the week, followed by a blank
SetCommaWeekday	print the full name of the day of the week, followed by a comma and a blank.
OTimeMonth	Direct printing of the month
Set3Month	print the first three letters of the name of the month.
SetFullMonth	print the full name of the month.
SetArabicMonth	print the month as an arabic number
SetRomanMonth	print the month as a Roman numeral
OTimeMZero	Directs whether a numeric month has a leading zero
SetMZero	print a leading zero if the month is less than 10
SetMNoZero	print a leading blank if the month is less than 10.
OTimeDZero	Directs whether the day of the month has a leading zero
SetDZero	print a leading zero if the day is less than 10.
SetDNoZero	print a leading blank if the day is less than 10.
OTime2Year	Directs whether the year is printed as YY or 19YY.
Set2Year	print the year as YY, with a leading zero if less than 10.
Set4Year	print the year as 19YY.
OTimeSeparator	Directs how the day, month, and year are separated.
SetSepBlank	separate the three fields with blanks
SetSepDash	separate the fields with dashes
SetSepSlash	separate the fields with slashes
SetSepComma	separate the fields with blanks, following the day-month pair with a comma.
OTimeDayFirst	Directs whether the day or month gets printed first.
SetDayFirst	
SetMonthFirst	
OTime12Hour	Directs whether the 24-hour clock or 12-hour clock is used
Set12Hour	print times on the 12-hour clock, followed with A.M. or P.M. Noon prints as 12:00 N, midnight as 12:00 M.
Set24Hour	print times on the 24-hour clock.
OTimeColon	Directs whether the hour and minute fields are separated by a colon
SetColon	
SetNoColon	
OTimeSecond	Directs printing of the seconds
SetNoSecond	
SetSecond	print seconds, preceded by a colon.
SetMillisecond	in addition, print milliseconds preceded by a period
SetMicrosecond	in addition, print microseconds

20.5.13 OUTCLOCK(BCB,CLOCK,MASK)

Internal name TIM005

BCB is the address of a buffer control block. CLOCK is the address of a four-word uncorrected clock value. MASK is as for TIMEDATE. This routine is a version of TIMEDATE that takes a clock value instead of a CALCTIME vector as input. It calls CALCTIME and TIMEDATE.

This page intentionally left blank

20.6. IOBCB/FormString Interface

Module BCBTYP Maintainer: Richard H. Gumpertz

The routine BCBTYP.M11[N810RG02]/A is used together with the BLISS-11 macro package called BCBTYP.R11[N810RG02]/A. These macros provide a convenient way of doing ASCII output on Hydra from BLISS-11 routines using the FormString convention (cf. 19.3). It is assumed that they will be used with the IOBCB package (cf. 19.2) and that any appropriate initialization has already been done.

The BCB arguments listed below are passed in register 1, via a BlissReg1 linkage. This should be of no concern to the user, however, if he uses the suggested require file. There is a linker command file called BCBTYP.LMD[N810RG02] which includes the necessary object files.

Wherever the macros described below have a parameter named FORMAT, that parameter must be a literal string (in single or double quotes).

20.6.1 Special definitions

Routines called via the "&*@" escape mechanism are invoked by a call that looks like

```
BlissReg1(Arg,BCB,CharsPrintedSoFar,ArgLstPtr,PrefixVal)
```

where the arguments are like those defined in section 19.3.7. ARG is the address of the routine being called; it must be a blissreg1 routine, taking arguments BCB, CHARSPRINTEDSOFAR, and so on.

The control code "&$" simply causes the BCB flush routine to be invoked.

20.6.2 Type(BCB,Format,Arg1,...,Argn)

This macro will output to the buffer the string resulting from conversion by FormString of the rest of the arguments. FORMAT should be a legal BLISS-11 string. The macro will automatically create the necessary uplit, including a CarriageReturn-LineFeed and "&$" at the end. Like FormString, Type has as its value the number of characters outputted.

20.6.3 TypeNoNewLine(BCB,Format,Arg1,...,Argn)

This macro is identical to Type, except that no CarriageReturn-LineFeed or "&$" will be appended to the format string. This is useful for input cue messages and for output which must go on one line but is generated by several different output calls.

20.6.4 FormattedType(BCB,FmtAddr,Arg1,...,Argn)

This macro takes the address of the format string rather than the literal character string. This can be used either when the format is not a constant or to save plit space when the same format is used more than once. Since BLISS-11 currently does not "pool" plits, this facility can be used to avoid the generation of multiple identical format string plits when the same format is used in several different places. Code for this might look like:

```
bind MyFormat = uplit(asciz $string( ... , "&/&$"));
...
FormattedType(BCB1, MyFormat, ... );
...
FormattedType(BCB2, MyFormat, ... );
```

20.6.5 BCBTyArray(BCB,FormatAndArgsArray)

FormatAndArgsArray is the address of a block of storage. The first word of the block contains a pointer to the asciz format string, and the succeeding words contain the values Arg1,...,Argn.

20.6.6 BCBTyStack(BCB,Argn,...,Arg1,FmtAddr)

This routine probably will never be called directly -- it is used by the macros above. It is listed here only for completeness.

20.7. Integer Conversion

Module BCBNUM Maintainer: Richard H. Gumpertz

The module BCBNUM.M11[N810RG02] contains two routines for converting 16-bit integers to strings.

The routine DivUns must be loaded with BCBNUM.

20.7.1 BCBNum(BCB,Number,Radix,MinDigits)

BCB is a pointer to the Buffer Control Block to which output is to be done. Note that BCBNum uses a BlissReg1 linkage; therefore this argument is passed in R1. NUMBER is the number to be converted. It is assumed to be unsigned (i.e. in the range 0-65535). RADIX is the radix into which the number is to be converted. The characters "A" through "Z" are used for radices between 11 and 36. Note that BCBNUM will recur infinitely (causing who knows what havoc) if the radix is 1! MINDIGITS is the minimum number of digits to be produced. If the number requires more digits than MINDIGITS, however, none will get lost.

The value of a call to BCBNUM is the number of digits output.

20.7.2 BCBSNm(BCB,Number,Radix,MinDigits)

This entry is identical to BCBNum except that signed integers are printed. For negative numbers, the "-" is counted as a digit when considering MINDIGITS.

21. Higher Level Modules

The modules described in this section contain functions the user is likely to wish to call. Chapter 22 describes lower level modules that may be called by the ones described here.

21.1. Simple User Initialization

Module HYDUSR Maintainer: David Lamb

The modules HYDUSR and HMAIN provide fairly simple initialization for user programs. HYDUSR consists of a set of subroutines. HMAIN is a standard main module which calls routines in HYDUSR, and expects the user's main entry point to be a global routine named HENTRY. Thus HMAIN corresponds to the old HYDUSR module. It uses the I/O package (see 19.2).

This package expects certain LNS slots to contain certain capabilities. The LNS slots are specified in the file HYDLNS.REQ. The only one that is truly critical is the IO port slot used by HYDINIT: slot 4, which is given the symbolic name IOPORT. Certain other routines uses slots 2 and 3, which should point to &SYSDIRECTORY and &USERDIRECTORY, respectively. The standard command objects place the appropriate capabilities in the appropriate slots (see 7.4).

Certain own variables in this module must be local to the process. The standard command objects for running programs loaded with HYDUSR ensure that this happens.

The "debug" and "code" sizes for routines in this module may not differ; this is becauses certain routines must be surrounded by SWITCHES NODEBUG declarations.

The use of this package is described in greater detail in chapter 7 on "getting started with BLISS11".

21.1.1 HARDERROR()

This routine is invoked via an interrupt linkage and is not intended to be called explicitly; it traps all errors that go through **$ERRPC**. It prints a diagnostic message on the teletype, including the PC and SP where the error occurred. If SIX12 is present it calls SIXCMD with argument #1000; otherwise it does a **$SUSPEND** with all zeroes as arguments. It calls OUTSTR, OUTCRLF, OUTOU, and BCBFLUSH.

21.1.2 HSIGNAL()

This routine is invoked via an interrupt linkage and is not intended to be called explicitly; it traps HYDRA signals. It prints the signal value (after stripping the high order bit), the PC and SP at the place of the error, and the top eight words of the stack. If SIGDATA is nonzero, it is also printed. If SIX12 is present, SIXCMD is called with argument #1001. In all cases, HSIGNAL returns the signal value. It calls OUTSTR, OUTCRLF, OUTOU, OUTCHR, and BCBFLUSH.

21.1.3 SETSIGNAL(FLAG)

This routine sets an internal flag used by HSIGNAL. If FLAG is nonzero, signals will print out a message on the teletype; otherwise no message will be printed. It returns the old value of the flag. This is intended for situations in which signals are sometimes errors and sometimes expected. Sections of code which expect signals can be bracketed with code like

```
LOCAL temp;
temp ← setsignal(1);
 ... code which might cause a signal ...
setsignal(.temp)
```

The flag is turned on by default.

21.1.4 HYDINIT()

This routine should normally be the first thing called from the user's program. It sets up $ERRPC and $SIGPC to trap to routines HARDERROR and HSIGNAL, respectively. If SIX12 is loaded it calls INIT612 to do appropriate setup.

HYDINIT calls OPENCHAN.

Note that HYDINIT is called by HMAIN, so user programs making use of HMAIN need not call it.

21.1.5 HYDFINISH(RETVAL,RETCAPA,RESTRICT)

This routine should be the last one called. If SIX12 is present it calls RET612; otherwise it does a $SUSPEND. The three parameters are passed to RET612 or the $SUSPEND, whichever is appropriate. Note that HYDFINISH is called by HMAIN, so user programs making use of HMAIN need not call it if they exit by returning to HMAIN.

This page intentionally left blank

21.2. Standard Main Module

Module HMAIN Maintainer: David Lamb

HMAIN is not a callable routine, but a main program that calls HYDINIT, provides input/output control blocks for teletype input and teletype output, calls the user's HENTRY routine, then calls HYDFINISH.

The "debug" and "code" sizes for routines in this module may not differ; this is becauses certain routines are surrounded by SWITCHES NODEBUG declarations.

21.2.1 MAINCALLER()

The sole purpose of this routine is to provide an entry point with debug linkages set. It calls SIXCMD with argument 0 if SIX12 is loaded with the program. It then calls HENTRY, the user's entry point. It contains an enable block to trap any signals that the user does not trap himself; this block prints a diagnostic message and calls SIXCMD with argument #1002.

This page intentionally left blank

blank page

21.3. User Error Reporting

Module USRERR Maintainer: David Lamb

21.3.1 USERERROR(SIGVAL)

Internal name USR001

SIGVAL is a 16-bit integer. This routine provides an interface between routines which wish to report errors and the user's Hydra signal handler. It examines the $SIGPC field of the LCB and calls the addressed routine (if it exists), faking a Hydra signal with value SIGVAL. It then returns SIGVAL as its value (provided, of course, that the signal handler returns). If no signal handler is provided, it simply returns its argument.

This page intentionally left blank

21.4. Hydra Directory Manipulation

Module DIRLIB Maintainer: David Lamb

The module DIRLIB contains routines for performing directory operations under HYDRA PM0. It uses GETARG from the string manipulation package . Each of these routines except DIRNAMES and DIRWORD corresponds to a directory base call; it is possible that the base calls will cause HYDRA signals.

The routines all take as a parameter an asciz string which contains a directory path. A directory path is a sequence of identifiers separated by dots. There may be up to ten identifiers in a path. Each identifier is truncated to ten characters if it is longer.

21.4.1 GETDIRECTORY(SLOT,DIRECT,STRING)

SLOT is the index of an empty LNS slot. DIRECT is the index of an LNS slot containing a directory capability. STRING is the address of an asciz string. Place into LNS slot SLOT a capability for the entry named in STRING, accessed through the directory in slot DIRECT. Returns the value returned by the **DIRGET** base call. Calls DIRNAMES.

21.4.2 PUTDIRECTORY(DIRECT,SLOT,STRING)

DIRECT is the index of an LNS slot containing a directory capability. SLOT is the index of a non-empty LNS slot. STRING is the address of an asciz string. Create an entry with a name given in STRING, accessed via the directory DIRECT, and copy the capability in SLOT into it. If an entry of the same name already exists, it will be replaced. Any directories in the path which do not exist will be created automatically. Returns the value returned by the **DIRPUT** base call. Calls DIRNAMES.

21.4.3 DELDIRECTORY(DIRECT,STRING)

DIRECT is the index of an LNS slot containing a directory capability. STRING is the address of an asciz string. Delete the entry named in STRING from DIRECT. Returns the value returned by the **DIRDELETE** base call. Calls DIRNAMES.

21.4.4 RENDIRECTORY(DIRECT,OLDNAME,NEWNAME)

DIRECT is the index of an LNS slot containing a directory capability. OLDNAME and NEWNAME are addresses of asciz strings. OLDNAME contains a full directory path; NEWNAME contains a single ten (or fewer) character name. Renames the bottommost level of the old name to the new name. Calls DIRNAMES.

21.4.5 DIRNAMES(DEST,SOURCE,MAX)

DEST is the address of a block of storage. SOURCE is the address of an asciz string. MAX is an integer, the number of bytes in the block addressed by DEST. Moves the directory path from SOURCE to DEST, formatting it as desired by the directory procedures. The routine returns the number of words in the resulting packed block, or -MAX if there were too many levels of directory path in the given string. This routine is probably not too useful for general users; it is called by all of the directory routines. It calls DIRWORD.

21.4.6 DIRWORD(DEST,SOURCE,LEN)

DEST is the address of a block of storage. SOURCE is the address of an asciz string. LEN is an integer, the number of bytes in the block addressed by DEST. Moves up to LEN characters from SOURCE to DEST, terminating when it reaches a null character or "." in SOURCE. If less than LEN characters are moved, pads the rest of the destination block with nulls. This routine is probably not too useful for general users; it is called by DIRNAMES. It calls GETARG.

21.5. Single Precision Floating Point

Module BLFP Maintainer: Bill Dietz

BLFP, the BLISS/11 compatible floating point software package, contains routines for doing floating point arithmetic, I/O, and conversion in the BLISS/11 environment. The routines are written in PDP11 assembly code and were primarily converted from routines contained in FPMP (Floating Point Math Package) a DEC package. The routines now available include only single precision (2 word) add, subtract, multiply, divide, square root, sine, cosine, comparison conversion, and I/O Standard DEC Floating Point format is used.

The file CBLFP.R11[N830P006] defines all the symbols needed to use BLFP. The files BLFP.OBJ[N810P006] and CFPIO[N810P006] should be included in the load module. Note that BLFP is not yet fully integrated with the current I/O package; it uses an old I/O system, PORTIO.

BLFP uses two pseudo Floating Point Registers. These are called the FLAC (Floating Point accumulator) and the OPRD (Operand). They are currently bound to locations #100-#106 and #110-#116 respectively. These locations are compatible with current C.mmp policy system software. The double operand routines use the contents of these two "Registers" as operands and return the result in the FLAC.

```
FLAC ← FLAC operation OPRD
```

The single operand routines use the contents of the FLAC and return the result to the FLAC.

```
FLAC ← operation FLAC
```

A set of BLISS macros have also been written to make the use of the package easier. The macro calls take the form Op(A,B,C), or Op(A,B) or Op(A). Op is a floating point operation and A, B and C are the addresses of the operands. The form is always

```
A ← B op C
```

or

```
A ← op B
```

Addresses are used in the macros due to the BLISS stack usage.

The macro set also contains a group of six comparison macros. These macros return either 1 for true or 0 for false depending on the result of the comparison. For example FNEQ(A,B) will return 1 if the operand at address A is not equal to the operand at address B.

The functions currently implemented in the macro set are:

Operations:

FADD(A,B,C)	Addition	A ← B + C
FSUB(A,B,C)	Subtraction	A ← B - C
FMULT(A,B,C)	Multiplication	A ← B * C
FDIV(A,B,C)	Division	A ← B / C
FSQRT(A,B)	Square Root	A ← Sqrt B
FSIN(A,B)	Sine	A ← Sin B
FCOS(A,B)	Cosine	A ← Cos B
FNGT(A)	Negation	A ← - A
INTR(A,B)	Greatest Integer Function	

Comparisons:

FEQL(A,B)
FNEQ(A,B)
FGTR(A,B)
FLSS(A,B)
FGEQ(A,B)
FLEQ(A,B)

Conversion:

FLOAT(A,B)	
FIX(A)	Returns integer

There are no real I/O routines in the Basic package. There are however, routines which convert an ASCII string to a floating point number and convert a floating point number to an ASCII string. Three BLISS routines have been written to make these conversion routines work as I/O routines. One routine, CONV(X,Y), takes a plit (X) and converts the ASCII to a floating point number and places it at Y. The second routine FIN(X) waits for input at the tty, converts it and places it at X. The last routine FOUTN(X) converts the number located at X and prints it on the tty. For C.mmp users, LFOUTN(X) has been included. LFOUTN(X) uses the line printer as the output device. These I/O routines along with a package error routine are included in a separately compiled BLISS/11 module. These routines are not compatible with HYDIO at this time; they may eventually be replaced by a variant of the ALGOL68 runtime package.

21.6. Unsigned 16-bit Divide

Module DIVUNS Maintainer: Richard H. Gumpertz

The module DIVUNS.M11 ([N810RG02]/A and [E130II00]/B) is a highly optimized divide routine which works with 16-bit unsigned numbers. (The standard BLISS-11 divide operation is signed, which yields different results.)

Division by 0 yields quotient 0, remainder=dividend. One may, if one wishes, thus consider a 0 divisor to be equivalent to #200000 (1 bigger than the biggest unsigned 16 bit number). If one is more concerned with division by 0, one should check the divisor explicitly in the calling routine.

21.6.1 DivUnsigned(R0=Dividend,R2=Divisor)

This call cannot be exactly described in BLISS-11. Arguments are passed in in R0 and R2; values are returned in R0 and R1. The quotient is returned as the "value" of the call (in R0) and the remainder is returned in R1. Since BLISS-11 assumes no registers are changed by a routine other than R0, it is not really "kosher" to call DivUnsigned from a BLISS-11 program. Therefore a group of macros have been written which do this call safely from a BLISS-11 program; they may be found in DIVUNS.R11 ([N810RG02]/A and [E130II00]/B) and are documented below.

21.6.2 UnsignedQuotient(Dividend,Divisor)

This macro returns the unsigned quotient DIVIDEND/DIVISOR as its value.

21.6.3 UnsignedRemainder(Dividend,Divisor)

This macro returns the unsigned remainder of DIVIDEND/DIVISOR as its value.

21.6.4 UnsignedDivide(Dividend,Divisor,Remainder)

REMAINDER is the name of a register variable or the address of a storage location. This macro returns the unsigned quotient DIVIDEND/DIVISOR as its value. In addition, it has the side-effect of setting REMAINDER to the remainder of the same division.

This page intentionally left blank

21.7. Four-word Integer Arithmetic

Module FOURWD Maintainer: Tom Lane

The routines in the module FOURWD[N810PM99] do arithmetic on four-word unsigned integers, such as clock values. Generally they take two arguments, both being addresses of four-word blocks of storage, perform some binary operation and leave the result in the block pointed to by the first argument. Thus

```
RTN(A,B)
```

behaves very much like

```
A ← .A operation .B
```

The four-word block has the least significant bits in the first word (the word with lowest address) and the most significant in the fourth. If you use these routines on clock values, you must zero the processor number field in the fourth word. None of these routines returns any useful value.

The routines are summarized in the following table

ADD4(A,B)	Add B to A
SUB4(A,B)	Subtract B from A
DIV4(A,B)	Divide A by B
MOD4(A,B)	Replace A with its remainder when divided by B
MUL41(A,B1)	Multiply A by the one-word quantity (not address) B1
DVMOD4(C,A,B)	Divide A by B, leaving the quotient in A and the remainder in C

This page intentionally left blank

blank page

21.8. Multiple precision arithmetic

Module ARITH Maintainer: Richard H. Gumpertz

The module ARITH ([N810RG02]/A and [E130II00]/B) and the require file ARITH.R11 comprise a small package of macros and routines which produce efficient code for manipulating multiple precision integers. It is anticipated that more calls will be added to this package as the need arises. Please submit requests for such additions to the author.

In keeping with the basic PDP-11 architecture, and unlike the PDP-11 floating point operations, It is assumed that multiple precision values are stored low order byte first. For example, a 19 bit integer at location X has .X<0,1> as its low order bit and .(X+2)<2,1> as its high order bit.

The routines are called via BLISS register linkages and so are fairly fast. Note that the Compare macros produce inline code and so do not require that ARITH.OBJ be loaded. Because of the strange linkage types used by the other routines and the possibilty of change to inline code, it is strongly suggested that the user use the require file provided.

21.8.1 Compare64(relation, X, Y)

This macro returns as its value either true (1) or false (0) as determined by the pseudo-BLISS11 expression

```
.X<0,63> relation .Y<0,63>
```

where `relation` is one of the keywords eql, neq, lssu, lequ, gtru, gequ, eqlu, or nequ. Note that X and Y are the addresses of 64 bit values, not the values themselves. If X and Y are complex expressions, they may get evaluated more than once because the macro refers to each of them in several places. The signed relations lss, leq, gtr, and geq will not be treated properly and so should be avoided. If there is sufficient demand, this may be fixed.

21.8.2 Compare60(relation, X, Y)

This macro is identical to Compare64, except that 60 bit values are compared. This is useful for comparing Hydra microsecond clock values without including the processor number in the comparison.

21.8.3 Compare32(relation, X, Y)

This macro is like Compare64 but for double word (32 bit) integers.

21.8.4 Add32(Src, Dst)

This routine adds the 32 bit integer at location SRC to that at DST and stores the result in location DST. The value of the call is either 0 or 1 reflecting the carry out of the high order bit of the result.

21.8.5 Sub32(Src, Dst)

This routine subtracts the 32 bit integer at location SRC from that at DST and stores the result in location DST. The value of the call is either 0 or 1 reflecting the carry (borrow) out of the high order bit of the result.

21.8.6 Bump64(n, Dst)

This routine adds the 16 bit unsigned integer N to the 64 bit value stored at location DST. The value of the call is either 0 or 1 reflecting the carry out of the high order bit of the result.

21.8.7 Bump32(n, Dst)

This routine adds the 16 bit unsigned integer N to the 32 bit value stored at location DST. The value of the call is either 0 or 1 reflecting the carry out of the high order bit of the result.

21.8.8 Down32(n, Dst)

This routine subtracts the 16 bit unsigned integer N from the 32 bit value stored at location DST. The value of the call is either 0 or 1 reflecting the carry (borrow) out of the high order bit of the result.

21.8.9 Dif32Limited(X, Y)

This routine computes the value

```
.X<0,31> - .Y<0,31>
```

and returns this as the 16 bit value of the call. If the result was negative (i.e. `.Y<0,31>` gtru `.X<0,31>`), however, the value returned is 0. Similarly, if the result would exceed 65535 (i.e. #177777, the greatest 16 bit unsigned integer), then the value returned is 65535. In other words, the result is forced to be inside the range 0-65535.

21.9. Dynamic LNS Slot Management

Module LNSLOT Maintainer: David Lamb

The set of routines in the module LNSLOT implement a runtime LNS slot allocator for programs running under HYDRA. The routines maintain an own table describing which slots are in use and how many times each is referenced (see SHRSLOT below). This table must be addressable whenever an LNS allocation routine is called. called, and which is not shared by any other LNS which will be making use of the routines.

21.9.1 INITSLOT()

This routine initializes the LNS slot allocator. Initialization involves finding the $LNSLENGTH of the LNS, marking all slots below that point as having one reference, and marking all slots above that point as free.

In some cases it may not be necessary to call INITSLOT. If the page containing the owns for the allocator is local to the running LNS and each invocation starts with a fresh copy of this page, then the first call to any allocation routine will call INITSLOT (the routines check a flag in the own page). Explicitly calling the initialization routine is intended for situations where there will be only one active copy of the procedure at a time but each copy uses the same page without copying. It is not necessary to call INITSLOT if you are using the standard HYDUSR command objects to run your program.

21.9.2 ALLOSLOT()

If there are any free LNS slots, one is selected and marked as "in use" with a reference count of 1; its index is then returned. If no free slot exists, a 0 is returned.

21.9.3 FREESLOT(SLOTNUMBER)

SLOTNUMBER is the index of a slot in the LNS. FREESLOT decrements the reference count for slot SLOTNUMBER. When the count reaches zero, it does a $DELETE(SLOTNUMBER) to clear the slot. It returns no value.

21.9.4 SHRSLOT(SLOTNUMBER,NUMBER)

SLOTNUMBER is the index of a slot in the LNS. NUMBER is a small integer. If SLOTNUMBER is in use, this routine increments its reference count by NUMBER and returns 1 (for TRUE). If SLOTNUMBER is out of the range of valid slots, or is free, or if adding NUMBER to the reference count would overflow the field used to hold the count (8 bits), this routine returns 0 (for FALSE).

The principal use of this routine is to share a slot between moderately independent sections of a program. Your initialization code can call ALLOSLOT and

save the value returned in some global variable. It can then call SHRSLOT to tell it how many sections of code will be using the slot. This allows each section to free the slot independently. When the last of the places sharing a slot frees it, the slot is deleted and marked as free. It is most useful when it is not clear a priori which section will finish with the slot first.

21.9.5 TRYSLOT(SLOTNUMBER)

SLOTNUMBER is the index of a slot in the LNS. Returns FALSE if SLOTNUMBER is out of range or in use. Otherwise it selects the indicated slot and allocates it as in ALLOSLOT.

21.9.6 NUMSLOT()

Returns the number of free LNS slots. Useful for finding if there are enough slots for the sequence of requests you are about to make.

21.10. ASCIZ String Package

Module STRLIB Maintainer: David Lamb

The module STRLIB contains a number of routines for use with asciz strings.

21.10.1 STRINDEX(CHAR,STRING)

CHAR is an ASCII character. STRING is the address of an asciz string. Return the 0-origin index of the first occurrence of CHAR in the STRING, or -1 if none is found.

21.10.2 STRLENGTH(STRING)

Return the number of characters in the asciz string.

21.10.3 COPYSTR(DEST,SOURCE,MAX)

DEST is the address of a block of storage, of length at least MAX bytes. SOURCE is the address of an asciz string. Copy characters from SOURCE to DEST, stopping with the first null in SOURCE, or when MAX characters have been transferred. The high order bit of each byte is left untouched. If less than MAX characters are transferred, a trailing null is written. Returns the number of characters copied.

21.10.4 GETARG(DEST,SOURCE,DELIM,MAX)

DEST is the address of a block of storage, of length at least MAX bytes. SOURCE and DELIM are the addresses of asciz strings. Copy characters from SOURCE to DEST, stopping with the first null in SOURCE, the first occurrence of one of the characters in DELIM, or when MAX characters have been transferred. If copying was stopped by a delimiter, the delimiting character is not copied. If less than MAX characters are transferred, a trailing null is written. Returns the number of characters copied.

GETARG calls STRINDEX.

21.10.5 UPPER(DEST,SOURCE,MAX)

DEST is the address of a block of storage, of length at least MAX bytes. SOURCE is the address of an asciz string. Copy characters from SOURCE to DEST, converting lower case letters to upper case, stopping at the first null in SOURCE or when MAX characters have been transferred. The high-order bit of each byte of the result is turned off. If less than MAX characers are copied, a trailing null is written. Returns the number of characters copied. UPPER(str,str,STRLENGTH(str)) will convert STR to upper case in place.

21.10.6 LOWER(DEST,SOURCE,MAX)

DEST is the address of a block of storage, of length at least MAX bytes. SOURCE is the address of an asciz string. Copy characters from SOURCE to DEST, converting upper case letters to lower case, stopping at the first null in SOURCE or when MAX characters have been transferred. The high-order bit of each byte of the result is turned off. If less than MAX characers are copied, a trailing null is written. Returns the number of characters copied. LOWER(str,str,STRLENGTH(str)) will convert STR to lower case in place.

21.10.7 EQU(STR1,STR2)

STR1 and STR2 are addresses of asciz strings. Compares STR1 and STR2, ignoring distinctions between case. Differences in the high order bit of a character are significant; thus #301 and #101 ("A") are not equivalent. Returns 1 (for TRUE) if they match and 0 otherwise.

21.11. Capability Printout

Module WHATS Maintainer: Guy Almes

The routines in the module WHATS format information about capabilities into the given buffer. These routines use the blissreg1 linkage convention. That is, the first argument is passed in register 1. Users need not normally be aware of this, as the standard require file HYDUSR.REQ has all of the necessary definitions.

21.11.1 WHATS(BCB,SLOT1,SLOT2)

SLOT1 is zero or the index of an LNS slot containing a capability with LOAD rights. SLOT2 is a C-list index. This routine formats information about a capability into the given buffer. If SLOT1 is zero, information is printed about the SLOT2th slot of the running LNS. Otherwise, the SLOT2th slot of the object in the SLOT1st slot of the LNS is examined. WHATS prints the number SLOT2, the auxiliary and Kernel rights of the object, its global name, and its printname. If the capability is null, in place of the global name WHATS prints "TEMPLATE" for a true null and "PARAMETER" for a parameter template.

WHATS calls OUTCHR, OUTSTR, OUTOU, and OUTCRLF.

21.11.2 WHATSIT(BCB,SLOT,MSG)

SLOT is zero or the index of an LNS slot. MSG is the address of an asciz string. WHATSIT prints the contents of MSG followed by a carriage return and linefeed, then calls WHATS to print information about SLOT if SLOT is nonzero. It then walks through the C-list of the object in slot SLOT (or the current LNS, if SLOT is 0) calling WHATS for each item. It calls WHATS, OUTSTR, and OUTCRLF.

22. Lower Level Modules

The modules described in this section are not intended to be used directly by the average user. They are called by modules described in chapters 20 21.

22.1. Interface to DAS

Module DASOPN Maintainer: David Lamb

The routines in DASOPN provide an interface to Sam Harbison's device allocation system. They can be used to establish and destroy connections to devices such as the line printer or DECtape drives. Slot 2 of the LNS, given the symbolic name SYSDIRECTORY, must contain a capability for the system directory (the command interpreter variable &SYSDIRECTORY).

22.1.1 OPENDAS(SLOT,PORT,CHAN,DEV)

Internal name DAS001

PORT is the LNS index of a port capability. DEV is the LNS index of an IO device capability, obtained from the PUBLIC.DAS.DEVICES under &SYSDIRECTORY. CHAN is an integer in the range 0-15, or -1 in which case DAS will find a free channel. SLOT is the index of a free LNS slot. This routine calls the DASCONN procedure to establish a connection with the indicated device via the given port and channel. The object returned by the DAS Procedure is placed in SLOT. If DAS gives an error, OPENDAS will return the negative number indicating the error status. Otherwise it will return the number of the channel used for the connection.

OPENDAS calls GETDIRECTORY, ALLOSLOT, FREESLOT, and USERERROR.

22.1.2 CLOSEDAS(SLOT)

Internal name DAS002

SLOT is the index of the slot used in the call to OPENDAS, which should contain the object returned by DAS. This routine calls the DASDISCONN procedure to break the connection to the device allocated by DASCONN.

CLOSEDAS calls GETDIRECTORY, ALLOSLOT, FREESLOT, and USERERROR.

22.2. Line Printer Open and Close

Module LPTOPN Maintainer: David Lamb

The module LPTOPN provides routines to establish and destroy connections to the line printer. Slot 2 of the LNS must contain a capability for &SYSDIRECTORY, so that a capability for the line printer IO device object can be found.

22.2.1 OPENLPT(SLOT,PORT,CHAN)

Internal name LPT001

PORT is the index of a slot containing a port capability. CHAN is an integer in the range 0-15, or -1 in which case a free channel will be found. SLOT is the index of a free slot which will be used to hold the capability returned by the DAS procedure. Establishes a connection to the line printer on the given port and channel; returns the channel number returned by the request to DAS.

OPENLPT calls ALLOSLOT, FREESLOT, GETDIRECTORY, and USERERROR.

22.2.2 CLOSELPT(SLOT)

Internal name LPT002

SLOT is the slot you passed to OPENLPT. Closes the line printer. Calls CLOSEDAS.

This page intentionally left blank

22.3. Message Creation

Module MAKMSG Maintainer: David Lamb

The routines in the module MAKMSG create and destroy port messages. They are called by the higher level portions of HYDIO. Like all routines taking a buffer control block as a parameter, they use the blissreg1 linkage convention. Users need not normally be aware of this, as the standard definition file HYDLIB.REQ, a part of HYDUSR.REQ, contains the necessary declarations.

22.3.1 MAKEMESSAGE(BCB)

Internal name MSG001

BCB is the address of an initialized input/output buffer control. Creates the number of messages specified by the IOBCBNUMMSG field of the control block, on the port and channel specified by the IOBCBPORT and IOBCBCHAN fields, large enough to hold the buffer accociated with the IOBCB plus the header required for the device specified by the IOBCBDEVICE field. For input channels, all messages but the last are sent to the connected device via the **$RSVPMSG** kernel call. For output channels, all messages are queued at the local port with an "OK" status via the **$REQUEUEMSG** kernel call.

22.3.2 KILLMESSAGE(BCB)

Internal name MSG001

BCB is the address of a fully initialized input/output buffer control block. The IOBCBINPUT field is odd (for true) if the first of the messages is being held at the port; this is normally the case for input channels. If this field is even, then all the messages must be received by the **$RECEIVEMSG** kernel call before being destroyed.

This page intentionally left blank

22.4. Low-Level Port Interface

Module IOTRAN Maintainer: David Lamb

The module IOTRAN contains routines which perform very low-level input and output, dealing directly with the Port subsystem. These routines assume that all necessary connections have been set up; for instance, OPENCHAN will set up connections (see 20.2).

The routines in this module use the blissreg1 linkage convention, as do all routines taking a buffer control block as a parameter (see 19.1). You need not normally be aware of this convention, since the require file HYDLIB.REQ, which is part of HYDUSR.REQ, contains all of the necessary declarations.

In addition to the routines described below, this module defines the global symbols READOP, WRITEOP, and HDRSIZE, which contain information peculiar to each of the devices which might be connected to the port.

INMESSAGE and OUTMESSAGE may return a negative number indicating a bad message status. This number consists of #140100 as a base, plus the four bit reply type of the received message. The meanings of these reply types are described in UIO.REQ[N810HY97]. In the case of an error, information describing the error is also placed in the IOBCB; this may be retrieved by calling the IOSTATUS function (see 19.2.1).

22.4.1 ACCEPT(IOBCB,RCVER)

IOBCB is the address of an input/output buffer control block. RCVER is the address of a block of storage at least six words in length. ACCEPT does a $RECEIVEMSG of the next message from the channel and port associated with IOBCB, placing the header returned by $RECEIVEMSG in the block pointed to by RCVER. It translates nonstandard error status codes such as those from the file system to a common form as described in UIO.REQ[N810HY97].

ACCEPT is not intended to be called by the average user; it is called by INMESSAGE and OUTMESSAGE and some I/O routines in other modules. It normally returns the reply type of the received message. In the case of a file system message containing an error code it returns the bottom five bits of the code plus #40.

22.4.2 INMESSAGE(IOBCB)

Internal name IOT002

IOBCB is the address of an input/output buffer control block. Requests a record large enough to fill the buffer, from the port and channel with which the IOBCB is associated.

If multiple buffering is going on (that is, if there are several Port messages

associated with the channel), the message copied into the buffer will not be the one requested. Its length may therefore be the length specified by a previous request. Thus if you wish to do things like requesting single characters from the terminal, you must ensure that only one message is created for the channel. The principal effect of this will be that any error status returned will refer to the status of the received message, which is likely to have been requested previously.

INMESSAGE returns either the value returned by the **$READMSG** call used to read the data into core, or a negative number indicating that an error has occurred. An attempt to read after an end-of-file has occurred is considered an error. On exit the BCB portion of the IOBCB correctly describes the information read in.

INMESSAGE calls ACCEPT.

22.4.3 OUTMESSAGE(IOBCB)

Internal name IOT001

IOBCB is the address of an input/output buffer control block. Writes the buffer to the port/channel combination with which the buffer is associated. Returns the value returned by the **$RSVPMSG** call used to send the data, or a negative status indicator. On exit the BCB describes an empty buffer. Calls ACCEPT.

22.5. File Opening and Closing

Module FILOPN Maintainer: David Lamb

This module references the directory package. This will disappear when **$TYPECALL** works. LNS slot 2, given the symbolic name SYSDIRECTORY, must contain a capability for the system directory (the command interpreter variable &SYSDIRECTORY), so the routines can find the file system procedures.

22.5.1 OPENFILE(PORT,CHAN,SLOT,MODE)

Internal name FIL001

PORT is the index of an LNS slot containing a port capability. SLOT is the index of an LNS slot containinga file capability. CHAN is an integer in the range 0-15, or is -1 in which case any free channel will be used. MODE is an integer indicating the type of opening desired: 0 for sequential read, 1 for sequential write, 2 for update. HYDLIB.REQ defines the symbols SEQREAD and SEQWRITE which may be used to specify this parameter. Sequential update and random I/O are not yet supported. Invokes the appropriate file system procedure to connect the indicated channel of the indicated port to the file monitor and open the indicated file object in the indicated mode. Returns the channel number.

OPENFILE calls GETDIRECTORY, USERERROR, ALLOSLOT, and FREESLOT.

22.5.2 CLOSEFILE(PORT,CHAN,LNAME)

Internal name FIL002

PORT and CHAN are as for OPENFILE. LNAME is a local name associated with the port containing a message which is at least eight words long; the message will be used to tell the file monitor to close the file. Upon return from CLOSFILE the message will have been sent to the file monitor via **$RSVPMSG.**

22.5.3 MAKEFILE(SLOT,TYPE,NAME)

Internal name FIL003

SLOT is the index of a free LNS slot. TYPE is a small integer; it will provide the file type parmeter, and should be 1 for the current file system. NAME is the address of an asciz string, and will provide the 10-character file name for FCREAT. This routine calls the file system FCREAT procedure, supplying it the given type and name fields. A capability for the created file is placed in slot SLOT. Returns whatever value is returned by the **$CALL** to FCREAT.

MAKEFILE calls ALLOSLOT, FREESLOT, GETDIRECTORY, DIRWORD, and USERERROR.

22.5.4 LISTFILE(SLOT)
Internal name FIL004

SLOT is the index of an LNS slot containing a file capability. Submits the file to the spoooler.

LISTFILE calls ALLOSLOT, FREESLOT, GETDIRECTORY, and USERERROR.

22.6. Generalized Formatting -- FrmStr

Module FRMSTR Maintainer: Richard H. Gumpertz

FrmStr (FRMSTR[N810RG02]/A and FRMSTR[E130II00]/B) is a BLISS-11 package (useable on any PDP-11) for formatting strings. It is particularly helpful when used with output routines to produce formatted output using the FormString convention (cf. 19.3).

The modules NUMOUT.OBJ and DIVUNS.OBJ must also be loaded from the same PPN. There are no owns in any of these routines so they may be put in a pure (read only) page when memory protection is available (as on C.mmp). For the convenience of C.mmp users, there is a linker command file called FRMSTR.LMD[N810RG02] which includes the necessary .obj files.

The module FrmStr implements the FormString convention in a manner which may be easily adapted to almost any PDP-11 environment without having to change the sourec code of the conversion routines. It does this by taking a routine as the destination for the characters. It does successive calls on this routine for each character produced. The calling sequence is as follows:

```
OutRtn(Extra, Character)
```

To allow the routine to be recursively reentrant but still be able to retain some information between calls, the EXTRA argument is simply passed along by FormString. It may be considered to be analagous to the display pointer (used in most implementations of Algol, PL/I, or BLISS-10) which is missing in BLISS-11. Obviously if only a single 16 bit value (which does not change between successive calls to OUTRTN) is needed to be passed around, EXTRA need not be literally a display pointer, which points to the information in question, but rather may be the datum itself. If the routine does not need any such data, it can take advantage of the way BLISS-11 passes arguments and simply assume it has one argument, CHARACTER.

22.6.1 Special definitions

The definition of the calling sequence for "&*@" is:

```
Arg(CharsPrintedSoFar,ArgListPtr,OutRtn,Extra,PrefixVal)
```

where the arguments are like those defined in section 19.3.7. The routine so called should either make calls directly on OUTRTN or call FrmStr recursively.

The control code "&$" causes OUTRTN to be invoked with the argument -1 instead of a character. This may be distinguished because in all other cases the high byte of the character will be zero.

22.6.2 Differences from the FormString standard

FrmStr does not currently support the &*H, &*UH, and &*W control codes. Right justification may be accomplished using the sequence "&#[...&]", which behaves exactly like "&#W&(...&)" should according to the standard.

22.6.3 FormString(OutRtn,Extra,FmtAddr,Arg1,...,Argn)

This macro causes the format pointed to by FMTADDR to be converted to a string which is handed to OUTRTN with successive calls for each character. The value of the call is the number of characters produced.

22.6.4 FmStrArray(FormatAndArgsArray,Extra,OutRtn)

FormatAndArgsArray is the address of a block of storage. The first word of the block contains a pointer to the asciz format string, and the succeeding words contain the values Arg1,...,Argn.

22.6.5 FmStrStack(Argn,...,Arg1,FmtAddr,Extra,OutRtn)

This routine probably will never be called directly -- it is used by the FormString macro above. It is listed here only for completeness.

22.7. Unsigned Integer Conversion

Module NUMOUT Maintainer: Richard H. Gumpertz

The module NUMOUT.M11 ([N810RG02]/A and [E130II00]/B) is an unsigned integer to string conversion routine. It converts a 16-bit unsigned integer to a series of characters.

The routine DivUns must be loaded with NUMOUT.

22.7.1 NumOut(Extra,OutRtn,MinDigits,Radix,Number)

EXTRA is an optional argument to be passed to the user-supplied routine. Since this is the first argument on the stack, it may be omitted if the user-supplied OUTRTN does not use it.

OUTRTN is the address of the user-supplied routine. It is called as:

```
(.OutRtn)(.ExtraArg, Character);
```

which may be considered as simply:

```
(.OutRtn)(Character);
```

if the EXTRA value is not needed by OUTRTN. MINDIGITS is the minimum number of digits to be produced. If the number requires more digits than MINDIGITS, however, none will get lost. RADIX is the radix into which the number is to be converted. Note that radices greater than 10 do not cause letters to be used for the digits, but rather the ASCII codes following "9" (i.e. ":", ";", "<"", "=", ...).. Note also that NumOut will recur infinitely (causing who knows what havoc) if the radix is 1! NUMBER is the number to be converted. It is assumed to be unsigned.

The value of a call to NumOut is the number of digits produced.

23. Summaries

This chapter contains a number of tables which summarize various pieces of information needed by users of the modules described earlier. Most beginning users will simply use the standard command files, in a manner illustrated by the standard template file XXXXXX.LMD. However, more spohisticated users will want to load only those routines they actually need, and eliminate the space wasted by unwanted routines. This becomes even more necessary as a user's programs grow to overflow page boundaries and it becomes necessary to re-arrange routines to fit onto pages.

For each routine, the intermodule reference sections tell which other routines it calls, and which routines call it. You can then look up the routines you wish to use in these tables, find what routines they call, then repeat the process for any new routines discovered in this fashion until no new routines are added. You can then look in the summary of module sizes to find out how big each routine is, and the name of the object file which contains it.

23.1. Summary of Calling Sequences

23.2. Internal Names

Many of the routines described in this document have names longer than the six characters of uniqueness allowed by the current assemblers and loaders. To resolve this, critical routines are given six-character internal names. The routines are referenced everywhere via their long names. The file NAMES.REQ provides macros that define the long names as macros for the short names. This section summarizes the short names.

Long Name	Short Name
CLOSECHAN	CHA002
CLOSEDAS	DAS002
CLOSEFILE	FIL002
CLOSEIT	CHA004
CLOSELPT	LPT002
CLOSETTY	CHA002
INMESSAGE	IOT002
KILLMESSAGE	MSG001
LISTFILE	FIL004
MAKEFILE	FIL003
MAKEMESSAGE	MSG001
NUMSTRING	NUM001
OPENCHAN	CHA001
OPENDAS	DAS001
OPENFILE	FIL001
OPENIT	CHA003
OPENLPT	LPT001
OPENTTY	CHA001
OUTCLOCK	TIM005
OUTMESSAGE	IOT001
TIMEDATE	TIM004
USERERROR	USR001

23.3. Summary of Module Sizes

Following is a table of modules and their sizes. Sizes are decimal numbers, and give the number of words of code and data. Data consists of owns, plits, and globals. Included with each routine is the name of the file containing its source; the default extension is ".B11".

Module	Routine	Code	Debug	Data	File
ARITH		44			
BCBNUM		67			
BCBTYP		579			
BLFP		1663	0		
BUFFER		128	172		
	BCBFLUSH	11	16		BUFFER
	GETLINE	43	54		BUFF05
	INCHR	16	20		BUFF01
	OUTBLOCK	20	28		BUFF04
	OUTCHR	10	14		BUFF02
	OUTCRLF	10	14		BUFF06
	OUTSTR	18	26		BUFF03
DASOPN		108	122	21	
	CLOSEDAS	49	56		DASOPN
	OPENDAS	59	66		DASOPN
DIRLIB		252	290	1	
	DELDIRECTORY	40	45		DIRL05
	DIRNAMES	39	49		DIRL02
	DIRWORD	25	33	1	DIRL01
	GETDIRECTORY	41	46		DIRL03
	PUTDIRECTORY	42	47		DIRL04
	RENDIRECTORY	65	70		DIRL06
DIVUNS		34			
FILOPN		243	270	55	
	CLOSEFILE	36	42		FILOPN
	LISTFILE	49	56	11	FILE02
	MAKEFILE	71	78	11	FILE01
	OPENFILE	87	94	33	FILOPN

FOURWD		209	0		
FRMSTR		544			
HMAIN		107	122	131	
	MAINCALLER	55	69		HMAIN
HYDUSR		334	349	134	
	HARDERROR	108	114		HYDUSR
	HSIGNAL	101	106		HYDUSR
	HYDFINISH	31	31		HYDUSR
	HYDINIT	88	88		HYDUSR
	SETSIGNAL	6	10		HYDUSR
IOTRAN		399	439	12	
	ACCEPT	115	126		IOTRAN
	INMESSAGE	149	164		IOTRAN
	OUTMESSAGE	135	149		IOTRAN
LNSLOT		157	192	62	
	ALLOSLOT	25	31		LNSLOT
	FREESLOT	31	36		LNSLOT
	INITSLOT	15	20		LNSLOT
	NUMSLOT	14	18		LNSL01
	SHRSLOT	39	49		LNSL02
	TRYSLOT	33	38		LNSL03
LPTOPN		52	62	12	
	CLOSELPT	6	10		LPTOPN
	OPENLPT	46	52		LPTOPN
MAKMSG		201	217		
	KILLMESSAGE	53	59		MAKMSG
	MAKEMESSAGE	148	158		MAKMSG
NUMFMT		175	204		
	NUMSTRING	25	35		NUMFMT
	OUTNUM	114	127		BUFF01
	OUTOU	36	42		BUFF01
NUMOUT		40			
OPENCH		315	338	4	
	CLOSECHAN	130	141		OPENCH
	CLOSETTY	21	26		OPEN00
	OPENCHAN	185	197		OPENCH
	OPENTTY	63	71		OPEN00

OPENIT		206	222	7	
	CLOSEIT	18	22		OPENIT
	OPENIT	188	200		OPENIT
STRLIB		174	222		
	COPYSTR	19	25		STRI04
	EQU	42	50		STRI02
	GETARG	29	39		STRI03
	LOWER	29	37		STRI06
	STRINDEX	18	22		STRI01
	STRLENGTH	8	12		STRI07
	UPPER	29	37		STRI05
TIME		833	937	199	
	CALCTIME	24	28		TIME
	CLOCKADJUST	65	74		TIME
	GETCLK	27	32		TIME04
	OUTCDT	15	19		TIME02
	OUTCLOCK	19	23		TIME03
	OUTDATE	8	12		TIME02
	OUTDT	8	12		TIME02
	OUTGDT	17	21		TIME02
	OUTMS	57	63	4	TIME01
	OUTTIME	8	12		TIME02
	TIMEDATE	386	428	154	TIME03
	ZCALCTIME	190	200	41	TIME
	ZGETCLK	9	13		TIME04
USRERR		29	34		
	USERERROR	29	34		USRERR
WHATS		156	178	20	
	WHATS	114	126		WHATS
	WHATSIT	42	52		WHATS

23.4. Intermodule References by Calling Routine

This section is an alphabetic list of modules or routines, together with modules or routines they call.

BCBFLUSH	USERERROR
BCBNUM	DIVUNS
BUFFER	USRERR
CALCTIME	ZCALCTIME, CLOCKADJUST
CLOSECHAN	CLOSELPT, CLOSEFILE, LISTFILE, FREESLOT, ACCEPT, KILLMESSAGE
CLOSEDAS	GETDIRECTORY, ALLOSLOT, FREESLOT, USERERROR
CLOSEIT	CLOSECHAN, FREESLOT
CLOSELPT	CLOSEDAS
CLOSETTY	KILLMESSAGE
DELDIRECTORY	DIRNAMES
DIRLIB	GETARG
DIRNAMES	DIRWORD
DIRWORD	GETARG
GETARG	STRINDEX
GETDIRECTORY	DIRNAMES
GETLINE	BCBFLUSH
HARDERROR	OUTSTR, OUTCRLF, OUTOU, BCBFLUSH
HMAIN	HYDINIT, HYDFINISH
HSIGNAL	OUTSTR, OUTCRLF, OUTOU, OUTCHR, BCBFLUSH
HYDINIT	OPENCHAN
INCHR	BCBFLUSH
INMESSAGE	ACCEPT
LISTFILE	ALLOSLOT, FREESLOT, GETDIRECTORY, USERERROR
MAKEFILE	ALLOSLOT, FREESLOT, GETDIRECTORY, DIRWORD, USERERROR
NUMOUT	DIVUNS
NUMSTRING	DIVUNS
OPENCHAN	ALLOSLOT, FREESLOT, OPENFILE, OPENLPT, INMESSAGE, OUTMESSAGE, MAKEMESSAGE, USERERROR
OPENDAS	GETDIRECTORY, ALLOSLOT, FREESLOT, USERERROR
OPENFILE	GETDIRECTORY, USERERROR, ALLOSLOT, FREESLOT
OPENIT	EQU, STRINDEX, GETARG, OPENCHAN, DIRNAMES, GETDIRECTORY, PUTDIRECTORY, ALLOSLOT, FREESLOT, USERERROR, SETSIGNAL, MAKEFILE
OPENLPT	ALLOSLOT, FREESLOT, GETDIRECTORY, USERERROR
OPENTTY	MAKEMESSAGE, USERERROR
OUTBLOCK	BCBFLUSH
OUTCDT	OUTGDT
OUTCHR	BCBFLUSH
OUTCLOCK	CALCTIME, TIMEDATE
OUTCRLF	OUTCHR
OUTDATE	TIMEDATE
OUTDT	TIMEDATE

OUTGDT	OUTDT
OUTMESSAGE	ACCEPT
OUTMS	OUTNUM, OUTCHR, FOURWD
OUTNUM	NUMSTRING, OUTCHR
OUTOU	NUMSTRING, OUTCHR
OUTSTR	BCBFLUSH
OUTTIME	TIMEDATE
PUTDIRECTORY	DIRNAMES
RENDIRECTORY	DIRNAMES
WHATS	OUTCHR, OUTSTR, OUTOU, OUTCRLF
WHATSIT	WHATS, OUTSTR, OUTCRLF
ZCALCTIME	FOURWD, DIVUNS

23.5. Intermodule References by Called Routine

This section is an alphabetic list of modules or routines, together with modules or routines which reference them.

ACCEPT	CLOSECHAN, INMESSAGE, OUTMESSAGE
ALLOSLOT	OPENIT, OPENCHAN, OPENDAS, CLOSEDAS, OPENLPT, OPENFILE, MAKEFILE, LISTFILE
BCBFLUSH	OUTCHR, INCHR, OUTSTR, OUTBLOCK, GETLINE, HARDERROR, HSIGNAL
CALCTIME	OUTCLOCK
CLOCKADJUST	CALCTIME
CLOSECHAN	CLOSEIT
CLOSEDAS	CLOSELPT
CLOSEFILE	CLOSECHAN
CLOSELPT	CLOSECHAN
DIRNAMES	OPENIT, GETDIRECTORY, PUTDIRECTORY, DELDIRECTORY, RENDIRECTORY
DIRWORD	DIRNAMES, MAKEFILE
DIVUNS	NUMSTRING, ZCALCTIME, BCBNUM, NUMOUT
EQU	OPENIT
FOURWD	ZCALCTIME, OUTMS
FREESLOT	OPENIT, CLOSEIT, OPENCHAN, CLOSECHAN, OPENDAS, CLOSEDAS, OPENLPT, OPENFILE, MAKEFILE, LISTFILE
GETARG	OPENIT, DIRLIB, DIRWORD
GETCLK	ZGETCLK
GETDIRECTORY	OPENIT, OPENDAS, CLOSEDAS, OPENLPT, OPENFILE, MAKEFILE, LISTFILE
HYDFINISH	HMAIN
HYDINIT	HMAIN
INMESSAGE	OPENCHAN
KILLMESSAGE	CLOSECHAN, CLOSETTY
LISTFILE	CLOSECHAN
MAKEFILE	OPENIT
MAKEMESSAGE	OPENCHAN, OPENTTY
NUMSTRING	OUTNUM, OUTOU
OPENCHAN	OPENIT, HYDINIT
OPENFILE	OPENCHAN
OPENLPT	OPENCHAN
OUTCHR	OUTCRLF, OUTNUM, OUTOU, OUTMS, HSIGNAL, WHATS
OUTCRLF	HARDERROR, HSIGNAL, WHATS, WHATSIT
OUTDT	OUTGDT
OUTGDT	OUTCDT
OUTMESSAGE	OPENCHAN
OUTNUM	OUTMS
OUTOU	HARDERROR, HSIGNAL, WHATS
OUTSTR	HARDERROR, HSIGNAL, WHATS, WHATSIT
PUTDIRECTORY	OPENIT

SETSIGNAL	OPENIT
STRINDEX	OPENIT, GETARG
TIMEDATE	OUTDATE, OUTTIME, OUTDT, OUTCLOCK
USERERROR	OPENIT, OPENCHAN, OPENTTY, BCBFLUSH, OPENDAS, CLOSEDAS, OPENLPT, OPENFILE, MAKEFILE, LISTFILE
USRERR	BUFFER
WHATS	WHATSIT

INDEX