77-30

THE HYDRA USERS MANUAL

Department of Computer Science
Carnegie-Mellon University

August 15, 1977

Edited by Andrew Reiner & Joseph M. Newcomer

This work was supported by the Defense Advanced Research Projects Agency under contract No. F44620-73-C-0074 and monitored by the Air Force Office of Scientific Research.

Preface

This manual is a descendent of the Hydra Songbook, which was conceived, written, and disseminated by Brian Reid to fill the void left by the lack of user documentation for Hydra. Like the Songbook, this document is intended to contain those things that are of general interest to the user community.

The User's Manual is published in two parts. The first contains those chapters that are of interest to general users, such as the introductory material and the chapters on the Command Language. The second contains reference material on the various subsystems. Most users will find the first part sufficient for their needs.

We would like to thank the many people who contributed material to this manual. Without their efforts and cooperation, the User's Manual could not have contained the amount and variety of material that it does. Also, we would like to thank Dan Klein, who provided comments on an earlier draft of this document.

Andy Reiner
Joe Newcomer
August 1977

Part I: Introduction to Hydra

Part I: Introduction to Hydra

1. The C.mmp User Community Forum

There is a bulletin board hanging in the C.mmp room whose purpose is to hold graffiti and bulletins from Hydra users to each other. If you feel the urge to use the PDP-10's to air your gripes, any mail sent to HY97 [N810HY97] will be posted on the gripe board if relevant.

1.1. Feedback

This document will be updated periodically, using information gleaned from (a) experience, (b) complaints, (c) the Hydra bulletin board, and (d) the Hydra people. A copy of it will always be on file in the C.mmp room, and some mechanism for its distribution will be set up.

1.2. Schedule of Availability

The C.mmp schedule can be found on the bulletin boards in the terminal room and the C.mmp room. This information is also in the file SCHED.DOC[N811HY97].

Signup sheets and an explanation of the signup system are posted in the C.mmp room.

1.3. The system news

There are two files on the Hydra "user" area (on the PDP-10): NEWS.DOC[N810HY97] and SYSNEW.DOC[N810HY97]. NEWS contains one-line blurbs about changes to the system or support programs. SYSNEW.DOC goes into much gorier detail and may point to more extensive documentation.

1.4. The System Area

Most software development for Hydra takes place on the PDP-10. The Hydra group attempts to make as much data as possible available on the Hydra user-systems area, account number [N811HY97]. Significant here are two files, DOC.DOC[N811HY97], the root of the documentation file tree, and REQ.DOC[N811HY97], which documents all BLISS/11 "require" files. The people responsible for Hydra documentation, under the watchful eye of Joe Newcomer, usually maintain a properly current list in DOC.DOC of any documentation which seems to make sense.

2. Introduction

This chapter is intended to provide a new user of the C.mmp hardware and the Hydra operating system with enough information to be able to perform useful work within a short amount of time. The main emphasis will be upon how to use the system rather than the why's or wherefors. It is assumed that the reader has experience with other computer systems and need only be instructed in the different structure of Hydra/C.mmp. Unfortunately, Hydra is a system that is quite different from standard systems so tasks which may be simple on other systems may appear difficult or awkward to perform on Hydra. Conversly, tasks which are "impossible" on other systems may be trivial (or at least possible) on Hydra.

Hydra has embodied in it several ideas and concepts which are not seen in most "standard" operating systems. First, and perhaps most importantly, C.mmp is a multiprocessor. Hydra allows the user to exploit this architecture (by supporting multiprocessing), or to ignore it (by providing a general purpose time-sharing system). Secondly, there is the notion of policy/mechanism separation in Hydra. You will (if you haven't already) hear people referring to the *Kernel*. The kernel is the part of the Hydra system that manages the (physical and virtual) resources of the machine. Indeed, the kernel has sometimes been described as an "extension" of the PDP-11 [1] instruction set. However, the key word here is manages. The kernel merely maintains the common pool of resources, such as memory (both primary and secondary) and processors. The kernel makes no attempt to implement a policy under which these resources are disributed to users. Those processes that do provide policy decisions are implemented at the user level. This provides great flexibility in design of user facilities. (Theoretically, every user on the system can be living on a different operating system). Finally, Hydra utilizes a Capability-based protection system. Because an understanding of this protection system is fundamental prerequisite to using Hydra, the following sections describe capability- based systems and capabilities.

2.1. Protection in Operating systems

Protection in a computer system can conceptually be viewed by considering two classes of entities and the allowed interactions between them. The classes of entities are:

> The active elements in the system, such as 'users' or processes
> Passive or 'acted upon' elements, such as files or I/O devices.

One can view the allowed interactions as being defined by a 2-dimensional matrix- one dimension being the active elements, the other the passive ones, with entry *(i,j)* containing the operations that active element *i* is permitted to perform

1 The processors used in the C.mmp hardware are PDP-11s

upon the passive element *j*. Note that each element *(i,j)* may contain zero or more operations. For example, an active element (a user) may be permitted to perform operation X (read) on some passive element (a file).

Due to the sparseness and uniformity of the allowed actions and the potentially large size, it is impractical to store actual protection information as a matrix. Instead the information is stored with the elements in one of the dimensions. In an *Authority* based protection system the information is stored with the passive elements (e.g., files). Each passive element has associated with it an access list, which is a list of names of active elements and their allowed access modes. This takes full advantage of the sparseness of the matrix and is in part the way TSS/360 and Multics handle protection. Typically, however, additional facilities are used to handle the uniformity. If active elements are divided into a few large groups (such as project groupings or an 'everyone' group) then access may be permitted to groups of active elements. The TOPS-10 monitor on the Dec PDP-10 uses this method exclusively, dividing the user community into 3 classes of groupings: a user, a user's project and everyone else, restricting access to files according to which classification the relationship between the accessing user and the owner of the file falls into.

In a *Capability* based protection system the access information is associated with the active elements. Each active element has a list of capabilities which define the passive elements it can manipulate. A single capability contains:

> an indication of the passive element it represents (the name or address of the passive element)
> a list of rights describing the how the possessor of the capability may manipulate the object.

Here lies a fundamental difference between capability and authority based systems. In TOPS-10, say, when you create a file, it is assumed a priori that you have ownership of the file. In Hydra, nobody "owns" any object, just capabilities to objects. However, a user can assume functional ownership of an object if he posesses the sole capability to it.

As might be expected, users cannot be allowed to directly make or modify capabilities. Instead, only indirect manipulation through the operating system can be permitted. There must also be restrictions upon how new capabilities may be created (and what rights they have) and how names are to be added to access lists.

2.2. Hydra Objects

The Hydra kernel implements a generalized capability based protection system. Everything in the system- programs, files, memory, etc. is represented be entities called *Objects*. An object consists of two parts:

> A linearly ordered list of capabilities, called the *C-list*, indicating which other objects in the system this object (or its manipulator) may access.

> A *data-part*, which contains arbitrary data.

Either the C-list or data-part may be empty.

2.2.1 Object type

Every Hydra object has a *type* attribute. This notion of type is analagous with that of data type found in programming languages such as Alphard, CLU, and Simula. Object types in Hydra fall into two major classes - User defined types, and Kernel types.

Kernel types are those that are defined within the kernel. They tend to embody the basic "building blocks" of the system. Examples of kernel types:

PAGE	The PAGE object is the basic unit of storage in Hydra (both incore and on secondary storage). Its C-list is empty, and its data-part (which is not accessable to users) contains information about the location of a PDP-11 page of memory.
PROCEDURE	A PROCEDURE object is somewhat analogous to a binary load module file in a conventional system. It contains capabilities for the PAGE object(s) that contain the actual binary program to be run and other information. Like a load module file, it itself is not dynamically executed, but instead contains the necessary data to be loaded into memory at run time. This definition of a PROCEDURE object is not strictly accurate, but the analogy holds.
LNS	An LNS (Local Name Space) is the object that is dynamically excuted at run time. An LNS is *incarnated* from a PROCEDURE object and, like most Hydra objects, may have capabilities for other objects in its C-list. In the case of an LNS incarnated from the Algol 68 PROCEDURE, one of these objects might be the source file and another might be the object file. Again, the analogy here is more important than the (very loose) definition of an LNS

User defined types are object types that are defined outside of the kernel. Examples of user defined types are:

SuperFile	A superfile provides the same kinds of services that are provided by a file in a standard system.
Catalogue	Catalogue objects perform the same function that a directory provides on a PDP-10, with a major exception. The PDP-10 directory system can only store files, while a Hydra a Catalogue has the ability to provide access to an object of any type once it is given a capability for that object.

2.2.2 Operations on objects

2.2.2.1 Kernel Calls

The Kernel provides a set of operations that one might perform on objects, called *Kernel calls*. Kernel calls tend to be operations that apply to all objects, regardless of type. Examples:

(Assume that capability X references an object)
$Clength(X) Returns length of object's C-list
$Dlength(X) Returns size of object's data-part

2.2.2.2 Typecalls

Every user defined object type has associated with it a set of procedures that can be performed on objects of that type. These procedures are known as *Typecalls*. It is only through these typecalls that a user can manipulate the internal representation of user-defined objects.

2.2.2.3 Subsystems

A user-defined type and the typecalls on that type define a Hydra *Subsystem*. One of the major purposes of a subsystem is to provide user functions that access the internal representation of objects without actually letting the user directly access this representation. Consider the following:

It would be unwise to allow users to be able manipulate the internal representation of, say, a SuperFile whenever they wanted to edit one. Therefore, the SuperFile subsystem is constructed so that it has the sole power to create new SuperFiles. Upon request, the SuperFile subsystem will create a SuperFile, and return a capability for it- without the rights necessary to alter it internally. Subsystems, however, have *rights amplification* abilities. Upon being passed the users capability for an object (in this case a SuperFile), the SuperFile subsystem can create an identical capability, but with the additional rights necessary to perform the requested task. In this particular case, when the user wants to edit his SuperFile, he passes his capability for it to the SuperFile subsystem, along with an Edit request, and the subsystem takes care of internally manipulating the SuperFile.

This concept of rights amplification is fundamental to the construction of a protected subsystem in Hydra. The details are discussed in the Hydra Kernel Reference Manual.

2.2.3 Capability Paths

As mentioned above, Hydra objects may contain capabilities that point to other objects in the system. These objects, in turn, may have capabilities for other objects, and so on. Let us define &CAPA to be a capability for an object X. We may then talk about the capability &CAPA.1, which is the first capability in object X's C-list. Likewise, &CAPA.1.4 is the fourth capability in the object referenced by the first capability in the object referenced by &CAPA. It is left as an exercise to the reader to recite the name of &CAPA.1.4.2 in one breath or less. In this last example, the capability you finally come up with is the _target_; &CAPA.1.4 is called the _pretarget_, and &CAPA.1 and &CAPA are called _steps_. Paths can go to arbitrary depths.

You should be aware that &CAPA, &CAPA.1, &CAPA.1.4, ... are all _capabilities_ that reference _objects_. It sometimes gets confusing when dealing with capability paths to keep track of objects and capabilities.

2.3. For More Information

This has only been a brief, informal discussion of the C.mmp/Hydra system. For more information, try these sources:

> Proceedings of the FiFth Symposium on Operating Systems Principles, (SOSP5), November 1975. There are three excellent papers in this Proceedings about Hydra.

> The Hydra Kernel Reference Manual. You should get hold of a copy of this sooner or later. It discusses most of the topics covered in this chapter, and in much greater detail.

3. C.mmp: the Hardware

A quick summary of the hardware

Processors:

* DEC PDP-11 family with slight CMU alterations
* Maximum of 16 processors
* Different models can be used concurrently (e.g. 11/20 and 11/40).
* Currently operational:
  > 5 PDP 11/20 processors;
  > 11 PDP 11/40 processors, all with CMU-developed writable micro-store.

Relocation Units:

* CMU designed and built
* Interfaces Unibus to crossbar switch's processor ports
* Relocates processor and I/O addresses
* Provides address and data parity generation and validation

Crossbar Switch:

* CMU designed and built
* Switches 16 processor ports to 16 memory ports on a single-word request basis.
* Allows up to 16 simultaneous memory accesses and arbitrates request conflicts.
* Provides manual control of all 256 crosspoints, permitting arbitrary system partitioning.

Memory:

* 16 CMU-designed and built ports, each housing up to 1 Mword.
* Currently operational: (1.16 Mwords total)
  > 12 ports of AMPEX 1865 core storage, 648 Kwords total; (250 nsec access, 650 nsec cycle, with overlap)
  > 5 ports of EM&M MOS memory, 640 Kwords total; (400 nsec access, 400 nsec cycle)

Interprocessor Control:

* CMU designed and built
* Provides interprocessor interruption at 3 priority levels.
* Allows each processor to start, continue, and stop an arbitrary subset of the 16 processors.

Time Base:

* CMU designed and built
* Provides global time source with resolution to 4 usec.

Peripherals:

* Standard PDP-11 devices are plug-to-plug compatible.
* Currently operational:

  > 5 Mwords swapping storage
  > 60 Mwords disk storage (moving head)
  > 6 DECtape drives
  > High-speed links to PDP 10 and front-end terminal handler
  > Interface to ARPA network
  > Line printer
  > Miscellaneous small peripherals

* Awaiting software development:

  > 300 Mwords disk storage (RP06/3330-like moving head)
  > Magnetic tape drive

4. Major Subsystems

This is a short description of the major user subsystems on Hydra.

> A **Catalogue Subsystem**, which holds, manipulates, and prints catalogue objects.

> A **Process Subsystem**, which is the interface between users and the scheduler (PM1). Provides such functions as process-create, process-kill, process-start, etc.

> A **Device Allocation Subsystem**, which interfaces the user with the I/O hardware available. Provides a certain amount of resource manipulation and control. A large amount of developmental work is pending on the I/O Subsystem.

> A **Command Interpreter Subsystem**, which decodes and interprets commands typed by the user at the terminal. The command language is a remarkably powerful Algol-like language, which has recursion and macro-definition capabilities. There is also a facility for creating and storing Command Objects, which means your command sequences can be stored away for another day.

> A **Job Subsystem**, which controls access to the system and allocation and accounting, and which interfaces to the user through a Job Monitor (JMON).

> A **Utility subsystem**, whose major components for the moment are a version of SIX12 for debugging BLISS programs, and various time, date, and core-image-creation things.

> A **Device Allocation Subsystem**, which provides for dynamic assignment of physical devices (DECtapes, Line Printer, etc.) among several users.

> A set of **File subsystems**, which provide users with higher-level file constructs, e.g., lines, records, random-access I/O, etc. and provide a uniform interface to a number of interesting file types (e.g., SOS and Undifferentiated Byte Stream (UBS)).

5. The Hydra User Environment

5.1. Logging On/Off

To log on:

1) Give the front end the "C" command to connect to C.mmp
2)You will now recieve one of the following responses:

> The message "Welcome to C.mmp/Hydra" followed by some other messages, and eventually the JMON prompt "@".
> The message "Reconnected to C.mmp/Hydra. Hit <return> to get the prompt "@".

3)If you received the prompt ">" instead of "@" in the above step, then you are already at the Command Language. If not, then type "CL" to get there.
4) Now type "LOG()". It will prompt for necessary information. If this is your first login, answer the request for the password with a carriage return, and you will be asked to supply one.

To log off:

1)Type "KJOB" to the Command Language prompt ">"
2)After receiving the "@" prompt again, type "K"

5.2. Introduction to the Command Language

The Hydra Command Language is the vehicle of communication between Hydra and a user at a terminal. Common practice in documenting interactive systems is to itemize the 'available commands' rather than talk about a language. However, thinking of it as a language allows us to assume a consistent syntax, document the syntax, then move on to a discussion of how the Commands behave.

This document is intended as a command language primer for people generally familiar with the structure of the Hydra environment (i.e. objects and capabilities). It should be sufficient to enable a novice to sit down at a terminal connected to Hydra and do something interesting if not useful. In conjunction with other documents, you may even be able to do something useful.

5.3. Basic Command Language Structure

The basic form of the language is that of a general, expression-oriented, block structured programming language. The syntax resembles that of BLISS in a general way, with features added or altered to allow access to specific Hydra features, such as the Catalogue subsystem and the facilities of Policy Module 1 The language follows the BLISS philosophy insofar as every "statement" is an expression which can return a value. We will use the terms "expression" and "statement" somewhat interchangeably in discussions of the command language; it should be understood that the "statement", as a non-value-returning entity as in Algol-60, PL/1, etc. does not exist. All "statements" return values, and as we shall see, some return multiple values. Although all the facilities described below are part of the command language subsystem, we will occasionally just say that "Hydra" does this or that. It doesn't really matter at this stage. Likewise, we may occasionally mention the "command interpreter" or CI, which is the program which actually parses and executes command language expressions.

5.4. Using the Terminal

The command interpreter indicates that it is waiting for input by typing a ">" character on the terminal. The user is then invited to type in a command language expression (statement), which will be evaluated (executed). Every expression evaluates to an integer, which is printed on the terminal prior to the next prompt (">") or to a capability in which case "capa" is printed prior to the prompt.

Several expressions may be put on the same line by separating them with semicolons. A carriage return typed at the terminal is treated as a semicolon for convenience. To avoid extra long lines, if a carriage return is immediately preceded by a "↑", both the "↑" and the carriage return are ignored. Then the statement may be continued on the next line. The above does not apply inside of quoted strings. Strings may contain any character except the string terminator. In stored programs, the conventions regarding carriage returns and "↑" are somewhat different. We will discuss those conventions when we get to stored programs.

The normal terminal control characters (control-H (rubout), control-U, etc.) are available to correct input as it is typed.

The remainder of this document essentially just discusses the various kinds of expressions you may type, and what you can expect to happen when you do. Except for a few special examples, all output typed by the command interpreter is underlined.

5.5. Variables

Like any good programming language, the Command Language has entities called variables which can be used in a straightforward manner. All variable names begin with an "&" character (to distinguish them from Catalogue names, described in section 5.8) which may be followed by any number of characters (upper and lower case characters are treated as the same characters). As of version 1.25 of the CL (expected to be released during the summer of 1977) the first 127 characters are significant. In previous versions, only the first 10 characters are significant.

There are three types of variables in the Command Language:

> WORD variables are just 16-bit integer variables.
> WORDVEC variables are one-dimensional integer vectors or string variables.
> CAPA variables hold Hydra capabilities.

5.5.1 WORD variables

As mentioned above, WORD variables are 16-bit integer variables. Their usefulness will be demonstrated in subsequent examples.

5.5.2 WORDVEC variables

WORDVEC variables do double duty as both integer vectors and string variables. The statement

```
>WORDVEC &A.5, &B.6
```

declares &A to be a vector of five integer variables or ten characters, denoted &A.1, &A.2, ..., &A.5, and declares &B to be a string of twelve characters or six integer variables. The following example illustrates the use of WORDVEC variables. Note that string accesses use "substring" notation; the pair [*a*, *b*] following a WORDVEC variable signifies the string starting at the *a*th character and *b* characters long.

```
>&A.2 ← 6                 % Assigns 6 to the second word of the vector. %
6>&B[2,3]←"ABC"           % Assigns "ABC" to the second through fourth
                          characters of the string. %
ABC>
```

5.5.3 Simple Examples

Here is an example of a terminal session, which will demonstrate some command language expressions.

```
>%The percent sign delineates comments
C:If carriage returns is entered before second delimiter,
C:the command interpreter prompts with "C:"%

>2+2*4                          % The regular arithmetic expressions are
                                available %
10>
>WORD &I,&J                     % This declares two simple integer variables %
0>
>&I ← 6                         % Simple assignments are possible, %
6>&J←&I+7                       % as are more complicated ones. %
13>
>TYPE "&J=",&J,"?M?J"
                                % You can type things out, but remember to
                                type the CRLF,
too (?M?J) %2
&J=13
0>                              % The type statement itself has value 0
                                %
>TYPE "&J=",&J,"
S:"                             % You can put the CRLF in the string
                                itself, in which case the command
                                interpreter prompts with "S:" for more
                                of the string.%
&J=13
0>
>while &I lss &J do (TYPE "A"; &I←&I+1)
                                % Iteration and compound expressions;
                                return value is that of last execution of
                                the compound expression. %
AAAAAAA13>
>If &I eql 13 then 1 else 0     % Conditional expressions%
1>
>&I eql &J                      % In relational expressions, 1=true,
                                0=false %
1>
```

[2] Control-X, for any character X, is written ?X. See section 5.10.

5.5.4 CAPA variables

CAPA variables may hold *capabilities*. Capabilities are obtained by retrieving them from Catalogues or by invoking *typecalls* or *procedures* which return capabilities. The Command Language also provides facilities for accessing capabilities via *paths* in objects. If CAPA variable &C contains a capability for an object, then the expression &C.2 represents the second capability in the C-list of the object. Expressions of this type are called *object walks*, and may appear on either side of an assignment operator. Object walks may go to any depth (i.e. &C.1.1.2.5), and may involve arbitrary Command Language expressions, such as &C.&J.3 or &C.(3*&J), where &J is assumed to be a WORD variable.

Section 5.6.1 has an example showing the use of CAPA variables.

A CAPA variable may be thought of as an LNS slot in the commahd interpreter LNS. The implications of this will be understood in the context of executing Kernel calls (K-calls) from the command language.

5.6. Invocations

The Command Language provides facilities for invoking several different kinds of functions:

1. Hydra Kernel Calls (K-calls).

2. Predefined Typecalls.

3. General Hydra procedures.

4. Command Language programs ("Commands").

5.6.1 Kernel calls

Kernel calls, or K-calls, are invoked by the standard names defined in the Hydra Reference Manual and the BLISS-11 require files. The kernel calls look the same as they do in the reference manual. Since paths are evaluated before the kernel call is executed, certain kernel calls such as $INTERCHANGE do not have the desired effect. For example, $INTERCHANGE(A.1,B.1) does not interchange the first capabilities in the C-lists of A and B. Instead, A.1 and B.1 are evaluated and the resulting capabilites are interchanged, effectively a no-op.

The Command Language form of Kernel K-call arguments is related to the form described in the Reference Manual as follows:

> Where the K-call requires a capability specified by a simple index or a path, use a CAPA variable, an object walk, or a Catalogue walk in the Command Language invocation.

> Where an integer value (other than an address) is required, use a Command Language expression which evaluates to an integer.

> Where an address of a block of data is called for, use a WORDVEC variable without an access specification. Be sure that the WORDVEC is long enough for the kernel call or you may get a fatal error.

Kernel calls return numeric values (called *signals* when they are negative) as described in the Reference Manual. They never return capabilities explicitly, but they may have side effects which cause capabilities to be stored in slots specified in the argument list. The following example may clear things up a bit.

```
0>CAPA &B                   % Declares a capability variable %
0>$MAKEUNIVERSAL(&B)        % Kernel call - creates a universal object and
                            returns a capability for it in &B %
0>$CLENGTH(&B)              % Returns length of C-list of Univ object, %
0>                          % Which turns out to be 0, naturally enough. %
>&B.3 ← &B                  % We store a capability (&B) into the third slot
                            in the C-list of the Univ obj.  Circularity! %
Capa>                       % Indicates the result was a capability %
>$CLENGTH(&B)
3>                          % The length is now three. %
```

5.6.2 Procedures (predefined and user defined)

All parameters to Hydra Procedures (which include typecalls) must be capabilities, and thus are supplied in the Command Language in the same way as capability arguments to Kernel calls, i.e. use capability variables, object walks, and Catalogue walks. \$STACKDATA($a_1;a_2,...,a_i$) is a Command Language function which upon appearing in the argument list of a Procedure invocation, takes its arguments (integer or string expressions), puts them in a data object, and passes a capability for that data object to the procedure. The use of \$STACKDATA is illustrated below.

One difference between Hydra procedure invocations and Kernel calls is that a procedure may return a capability in addition to the normal numeric value returned by everything. This is the reason for a slightly different syntactic structure in the Command Language which is used to store the capability returned from a procedure. As an example, consider the predefined procedure \$MAKECMD. It takes no parameters and returns a capability for a Commands object. A sample invocation goes like this:

```
>CAPA &CMD
0>&CMD=$MAKECMD()
#0>
```

In this example, note that an equals sign was used for the assignment instead of a left-arrow. This is to demand the assignment of the returned capability rather than the returned integer value, which was #0. To help in handling the two return values (integer and capability), the Command Language has predeclared a WORD variable &RETVAL, to which is assigned the returned value of every procedure invocation. Thus the above statements could be followed by

```
>WORD &RET
0>&RET ← &RETVAL
```

if this value was needed.

The equals-sign is in fact more complex than has been indicated, but the advanced features will not be described here. They may be found in Chapter 20 in all their gory detail.

The concept of *subsystem* in Hydra is defined as

> A set of objects with a given *TYPE*
> A set of explicitly defined operations, called *TYPECALLS*, on objects of the given type

So, each type subsystem provides a set of typecalls to manipulate the objects of that type. Currently, the command language predefines the typecalls for the JOB, CONNECTION, CATALOGUE, and PM subsystems. The predefined typecalls are similar to the Bliss macros for those subsystems except for those that return capabilities. In those cases, omit the return slot from the list of arguments of the call. The return value will be the value of the procedure. For instance, the Job subsystem has a typecall $LOGIN, to connect a Catalogue to a job [3] The form listed in Chapter 22 is:

$LOGIN(RETSLOT, &JOBPROCESS, &TTYPORT)

where RETSLOT is a capability for the returned Catalogue and the other two parameters are predefined Command Language variables (see section 20.5). In the Command Language, use the following form:

RETSLOT = $LOGIN(&JOBPROCESS, &TTYPORT)

For additional information on Typecalls, see the chapter on the specific subsystem.

3 The LOG() command utilizes this typecall. The object "LOG" itself is a Commands object - see section 5.6.4.

5.6.3 General Procedures

General user-defined procedures may also be invoked from the Command Language. In this class of procedures are included not only any procedures the user might create herself, but most of the system utility programs such as editors, compilers, etc. The only difference between these invocations and invocations ofpredefined typecalls is that general procedures do not have their names predefined in the command interpreter's symbol table, so capabilities for the procedures must be stored in Catalogues or capability variables. As an example, consider the procedure OBINFO, which is used to gain information about an object. The following command will invoke the obinfo procedure.

```
>&SYSDIRECTORY.PUBLIC.HARBISON.OBINFO(OBJECT, &TTYPORT)
```

There will be more examples later. Procedures at the end of object and Catalogue walks may be invoked, e.g. "&A.2.4()".

5.6.4 Commands

A Commands is an object holding one or more command language statements, which can be invoked by users just as if it were a procedure. As such, it is useful for those tasks too difficult or long to do directly at a terminal, but too simple to require a Hydra procedure. Often, Commands are used to insulate the user from unpleasant features of the command language. Most procedures have one or more Commands associated with them to allow such things as default parameters.

Like procedures, capabilities for Commands are stored in Catalogues or CAPA variables. They are invoked in the same way, and will take integers, strings, or capabilities as parameters.

Commands objects, like most user-defined Hydra objects, have a number of typecalls associated with them to provide useful operations upon objects of type COMMANDS. The most useful ones are listed below. For a full list, see section 20.6.

(Assume &Ret and &Cmd have been declared as CAPA variables, and &CMD references a Commands object.)

&Ret = $Makecmd()	creates a new Commands object, and returns a capability for it to &Ret.
&Ret = $Copycmd(&CMD)	creates a copy of the object referenced by &CMD, and returns a capability for it to &Ret.
$Editcmd(&CMD)	edits the text in the object referenced by &CMD with the C.mmp teco editor. Commands objects can also be edited with SOS.

$Listcmd(&CMD)	Lists the object referenced by &CMD on the line printer located in the C.mmp room.
$Printcmd(&CMD)	prints the object referenced by &CMD on the terminal.

5.6.4.1 &PARMS and &CDOPARM

There are two special variables associated with each Commands object upon invocation.

&PARMS points to an object of type UNIVERSAL that contains, in its C-list slots, the parameters that were passed to the Commands object in the invocation. For example, if &GORP is a capability a Commands object, then upon the invocation

```
        &GORP(Capa1,Capa2)
```

&PARMS.1 would contain Capa1 and &PARMS.2 would contain Capa2.

Note that Commands objects take no explicitly defined number of parameters.

Sometimes it is useful to have a list of capabilities available to a Commands object without having to pass each explicitly as a parameter. Each Commands object may have "implicit" parameters stored in its C-list, and made available at run time through the &CDOPARM variable. These are equivelent to the concept called "inherited capabilities" described for Hydra procedures. The following example shows how to manipulate this list:

```
>capa &U
>$Makeuniversal(&U)
0>&U.1←&Capa1
Capa>&U.2←&Capa2              %Store the required capabilities in the C-list of
                              &U%
Capa>$Writeclist(&GORP,&U)    %Copies the capabilities from the
                              C-list of &U to the C-list of &GORP%
#0>&U=$Readclist(&GORP)       %Returns a Universal containing the
                              contents of &GORP's C-list %
```

The following is an example of a Commands which invokes the TECO procedure.

```
Begin                                   %The Commands is enclosed by a
                                        block to release temporary
                                        variables on block exit%
CAPA &1PG;                              %Declare a capa variable to hold
                                        the page
                                        object.  The name is intended to be
                                        unusual
                                        to avoid conflict with user defined
                                        names. %
If $CLENGTH(&PARMS) LSS 1 then
                                        %&PARMS is a universal object
                                        holding
                                        capabilities for the parameters of
                                        the Commands.
                                        $CLENGTH(&PARMS) is the number
                                        of
                                        parameters. %
   (Type 'Page:';                       %If no parameters were passed,
                                        prompt
                                        the user for the object. %
    &1PG ← INTERPRET()                  %Get a capability for a page from
                                        the terminal.%
                                        %Note that ()'s instead of BEGIN
                                        END were used to enclose this
                                        block.  Since ()'s do not affect the
                                        symbol table, they are more
                                        efficient. %
   ) else &1PG ← &PARMS.1;              %If a parameter was passed, use
                                        it.%
&CDOPARM.1.TECO(&TTYPORT,&1PG)

                                        %&CDOPARM
                                        contains a capability for the
                                        inherited capabilities for this
                                        Commands object.
                                        In this case, the first capability in
                                        the C-list
                                        of the Commands is a Catalogue
                                        which
                                        contains a capability for the TECO
                                        procedure.
                                        The level of indirection was
                                        provided to simplify the task of
                                        replacing the TECO procedure with
                                        a new one.%
End                                     %End of command object.  Since no
                                        ";" followed the invocation of
                                        TECO, the value of the command
                                        object is the value of the
                                        invocation.%
```

5.7. Macros

Macros are available to allow the user to do something to make the terminal environment more attractive. The macro definition facility allows any identifier, variable, or control character to be defined as a macro. The definition looks like

```
        MACRO name = stringwithoutdollarsign $
    or
        MACRO name(parmlist) = stringwithoutdollarsign $
```

If a line starting with "MACRO" does not terminate with "$", the command language prompts on the next line with "M:" for more of the macro text.

Consider the following examples:

```
>macro univ(X) = $Makeuniversal(X)$
0>capa &W
0>univ(&W)                          %&W now contains a capability for a new
                                    Universal%
0>macro fillslot(A,B,C) = univ(A); A.C ← B$
0>capa &Z
0>fillslot(&Z,W,5)                  %A new universal is created, and &W is
                                    put in its fifth C-list slot%
0Capa>                              %Notice how each expression in the macro
                                    returned a value%
```

5.8. Catalogues

The Catalogue subsystem provides the facility for long-term storage of capabilities for objects, where long-term means beyond a single terminal session. In this sense, it serves a purpose similar to that of the file system on the PDP-10. Note that all of a user's context is lost in the case of a system crash, so it is valuable to store anything which represents a fair amount of work in its creation in the Catalogue system, even if it already exists in a CAPA variable.

This section only gives an introductory discussion of the Catalogue subsystem. For a complete description, see Chapter 19.

5.8.1 Predefined Catalogues

There are a number of Catalogues available to the user upon logging into the system. Capabilities for these Catalogues are automatically placed in predefined CAPA variables[4].

[4] These are not the only predefined variables in the Command Language. For more, see section 20.5

5.8.1.1 &USERDIRECTORY

Every user has a private Catalogue, accessable through the CAPA variable &USERDIRECTORY. [5] Like all Catalogues, it maps print names into stored capabilities. To list your Catalogue, use the predefined typecall **$CATALIST:**

```
≥$CATALIST(&USERDIRECTORY,&TTYPORT)
```

The parameter "&TTYPORT" is a predefined CAPA variable that contains a capability for the Hydra PORT object connected to your terminal. You will find that your Catalogue contains (among other things, perhaps) a Commands object called PROFILE. This Commands object is invoked automatically after you have logged in, providing you with a number of useful macro definitions, among other things. See section 5.9 for more on Profiles.

Because several names are used frequently, the following macros have been put in the standard profile

```
USER = &USERDIRECTORY

SYS = &SYSDIRECTORY

UT = &SYSDIRECTORY.UTILITIES

PUB =&SYSDIRECTORY.PUBLIC
```

The latter three will be discussed in the next section.

Because capabilities residing in &USERDIRECTORY are so frequently used, it is the default Catalogue. That is, if an object walk does nt begin with a CAPA variable or a Catalogue, &USERDIRECTORY will be used. Example:

```
&CAPA ← &USERDIRECTORY.TEST
```

is equivelent to

```
&CAPA ← TEST
```

5 At one time the functions of the Catalogue subsystem were performed by the Directory subsystem, therefore the name &USERDIRECTORY instead of &USERCATALOGUE.

5.8.1.2 &SYSDIRECTORY

&SYSDIRECTORY contains a capability for the system Catalogue. This Catalogue contains other Catalogues, command objects, procedures, and templates that are of interest to the general user community.

There are two useful Catalogues, called UTILITIES and PUBLIC, that reside on &SYSDIRECTORY and are of interest to most users.

UTILITIES contains procedures and Commands objects of general interest, such as the MAIL subsystem.

To list UTILITIES, type

```
≥$CATALIST(&SYSDIRECTORY.UTILITIES,&TTYPORT)
```

To invoke a Commands object or procedure that resides on a Catalogue other than the &USERDIRECTORY Catalogue, specify the object walk. If the object takes no parameters, you must still specify an empty parameter list (i.e., suffix it with "()")

To invoke the MAIL Commands object, for instance, type

```
≥&SYSDIRECTORY.UTILITIES.MAIL()
```

The PUBLIC Catalogue provides a means for users to make selected objects accessable by other users. It contains a Commands object called MAKEPUBLIC, and a Catalogue for each user who has created one. To get a public Catalogue, invoke the following Commands object:

```
&SYSDIRECTORY.PUBLIC.MAKEPUBLIC()
```

Like everybody else, you can access things in your public Catalogue through the PUBLIC Catalogue. However, you also have access to it through a new capability &USERDIRECTORY.MYPUBLIC, created by the above Commands object.

USER.MYPUBLIC and PUB.DOLMATZ both point to the same Catalogue (provided that your name is Dolmatz), but USER.MYPUBLIC posseses more rights it. If you want to insert a capability into your public Catalogue, you must perform an assignment using USER.MYPUBLIC:

```
≥USER.MYPUBLIC ← GRUNT
```

The above assignment stored a capability for the object pointed to by

&USERDIRECTORY.GRUNT (Remember - if an object walk specification doesn't begin at a Catalogue, &USERDIRECTORY is used as default) into the users public Catalogue. The assignment &SYSDIRECTORY.PUBLIC.DOLMATZ ← GRUNT will fail [6].

Everything that is put in the PUBLIC Catalogue is protected against deletion through any object walk going through &SYSDIRECTORY.PUBLIC. You must, however, protect against modification. This can be accomplished by using the *rights restriction facility* in the Command Language. To restrict rights, append the capability being assigned with an octal expression indicating which rights are to be restricted. For most purposes, it will be sufficient to restrict $MODIFYRTS (octal #20000).A good discussion of kernel rights can be found in The Hydra Kernel Reference Manual. An example of rights restriction follows:

```
>&USERDIRECTORY.MYPUBLIC← &PUBOBJ[,#20000]
```

This restricts the kernel right $MODIFYRTS in the capability passed to the Catalogue. For additional information about the rights-restriction facility, see section 20.3.10. One time when you should *not* attempt to restrict $MODIFYRTS is when the object to be assigned is of type CATALOGUE (i.e., a Catalogue) or DIRECTORY. If $MODIFYRTS is restricted on a capability pointing ot an object of either of these types, it will be impossible to list the contents of the Catalogue (or directory). Anyway, every capability in it will be implicitly protected from deletion because every object walk will go through &SYSDIRECTORY.PUBLIC.

5.8.2 Using the Catalogue Subsystem

Catalogues in Hydra have a directed graph structure, and can contain capabilities for an object of any hydra type (including other Catalogues) Therefore, it is possible to structure your user Catalogue to suit your needs. For instance, your top-level user Catalogue may contain an entry for a Catalogue named ALGOL, in which all your ALGOL68 programs and other related objects are kept.

Be aware that a capability for a particular object can be stored under more than one name in the same Catalogue (perhaps with different rights for each name!). On the subject of names, bear in mind that the name given to an object (called the *print name*) is of use ONLY for specifying that capability in the Catalogue. [7]

6 If it doesn't fail, congratulations! You've found a new bug.

7 The global name of an object (a unique 64-bit bit pattern) is assigned to each object upon creation. More on global names can be found in The Hydra Kernel Reference Manual

5.9. Profiles

As you may have noticed by now, doing things in Hydra requires quite a bit of typing. For example, to run the Algol68 compiler, one must make the following procedure invocation:

```
        ≥&SYSDIRECTORY.PUBLIC.ALGOL68.CMDS.ALGOL68()
```

The way to avoid all this typing, of course, is to declare a macro

```
≥MACRO ALG68 = &SYSDIRECTORY.PUBLIC.ALGOL68.CMDS.ALGOL68$
```

and then type

```
        ≥ALG68()
```

There is a special Commands object in your userdirectory called a *profile*. Every new account receives this profile when it is created. It is invoked every time you log on, and defines macros for a number of useful procedures.

To see what is in your profile, use the macro (defined in the profile) "print":

```
        ≥PRINT(PROFILE)          %Since PROFILE is on &USERDIRECTORY, there
                                 is
                                 no reason to specify this Catalogue in the object
                                 walk%
```

The word "HELP" has been defined as a macro. Type it and see what happens.

The size of the standard profile has been kept to a minimum. It is expected that subprofiles will be made available to special classes of users with particular needs (i.e., Bliss users, Algol68 users).

5.10. Input and Output

The command language provides a few I/O primitives for use mainly in command objects. The TYPE statement was demonstrated earlier in this primer. TYPE will output strings and integers, but not capabilities. An attempt to print a capability will result in "Capa" being printed. It is important to note that the TYPE statement does not provide the so-called "service" of putting in a free carriage-return linefeed. The user must type one out explicitly if one is desired.

To do this, a string containing the carriage-return linefeed must be supplied, and one way is with a macro (what? you skipped the section on macros? Go back and read it now!)

```
        MACRO CRLF="?M?J"$
```

The question-mark before a character is the "control-escape" character, and causes the command interpreter scanner to make the next character its control-shift equivalent. Thus "?M?J" turns into control-M-control-J, which if you didn't already know you will soon memorize as being carriage return and linefeed.

The input primitives are ACCEPT and INTERPRET. The ACCEPT expression, written "ACCEPT()", gets input from the terminal, returning a string of all characters up to, but not including, the carriage return-line feed which terminates the line. (There is an option which includes the "break" character(s) from the input line.) The value of the expression (the string) may be assigned to a string (WORDVEC) variable.

```
>WORDVEC &S.5
0>&S[1,10] ← ACCEPT()
Hello.                    ! We typed this line
Hello.>                   ! This is the return value, typed by the CI.
                          ! Note that no CRLF is included.
>TYPE &S[1,10]
Hello.0>                  ! The CI typed this line
                          ! The 0 is the return value
                          ! of the TYPE expression.
```

To cause inclusion of the carriage return linefeed, use ACCEPT(0). The default case, i.e., ACCEPT(), is the same as ACCEPT(1).

INTERPRET is more complicated than ACCEPT. It accepts an input line, and then feeds the line to the command interpreter, which executes it as if it were an expression (which it had better be). The return value is whatever the expression evaluates to, which may be an integer, a capability, or a string. You should note that the typed expression is executed in the context in effect at the location of the call to INTERPRET, not in the context of the "outer" levels of the command interpreter. This is important if you execute INTERPRET from within a command object which has defined local variables, e.g.

```
>WORD &I
0>&I ← 99
99>BEGIN
B:WORD &I,&J;                % The "B:" tells you that you are in a block %
B:TYPE "Well?? "             % The ?? means ? will be put in the string. %
B:&I ← 37
B:&J ← INTERPRET()
B:TYPE &J,"
S:"                          %The "S:" tells you that you are in a string. %
B:END
Well? &I                     % The block is executed here; we type input to
                             INTERPRET %
37                           % This is what is typed by the TYPE statement %
0>TYPE &I,"?M?J"
99>
```

This dynamic binding of names should be familiar to users of systems such as LISP or APL.

5.11. Pointers to more information

This primer is neither complete nor strictly accurate. More printed information may be found in the command language manual, which includes a complete syntax and list of features. It can be found in Chapter 20and on CMDINT.DOC[N810HY97]. Also available from time to time are updates on the Command Language. They are found on CL??.DOC[N810HY97] as they are available. The ?? refers to the version number of the command language. Unfortunately, even these documents leave something to be desired, and the only really satisfactory way of learning the command language is to use it and watch others do the same.

Good luck.

6. Terminals and C.mmp

6.1. Connecting a terminal to C.mmp

6.1.1 Front end terminals

Normally, user connections to C.mmp are made via the Front End. The Front End multiplexes terminals to several of the deparmental computers: the PDP-10's (the A system, the B system, or the D system), computer nodules (sic), the AI research systems, or to C.mmp (the C system). Most CMU computer users should be familiar with the Front End system by now, but a quick review wouldn't hurt.

The Front End is a PDP-11/40, physically located in the machine room behind the line printer, to which most of the terminals in the building are connected by means of direct cables. Some dialup ports (and soon all but Datel dialup ports) connect to the Front End. When a terminal is first connected to the Front End, or when its system is first brought up, the Front End must determine the speed of the terminal. One must therefore type a known character, namely ↑C, several times, while the Front End tries to interpret it at various speeds. When it finds a speed at which the incoming character looks like a ↑C, it prints a signon message and asks you to select a Host (H for help). To connect to C.mmp, type 'C'.

When Hydra receives a connection from the Front End, it generates a 'virtual terminal' and crosspatches you to it. All characters received by the Front End are transmitted directly to C.mmp. When a Front-End escape character (↑←, control backarrow) is received by Hydra, it breaks the connection to the Front End. If you break a C.mmp connection and then re-establish it, you will get reconnected to your own job.

The link from the Front End to C.mmp is a 4800-baud ASLI (Asynchronous Line Interface) which connects via processor 'C'.

6.1.2 The teletypes in the machine room

Almost no one uses the Teletype^{tm} terminals in the C.mmp room; after all, who wants a 110 baud upper-case-only noisy terminal which probably doesn't work all that well anyway, when nice 1200 baud video terminals are available? Therefore, one needs no information about how to mangle the patch panel to convince a Teletype to talk to C.mmp; anyone who is so desperate as to require C.mmp when the Front End is down is already a hacker and knows. Therefore, the material formerly in this section has been deleted.

6.1.3 Connecting from the Graphics

There is one CMU "Graphics Wonder"^{tm} terminal that is electrically connected to C.mmp. In order to get it and C.mmp talking to each other you have to do some handwaving. First, it is necessary to understand a little bit about the graphics keyboard. This bizarre keyboard is called a 'Stanford Keyboard', and was perpetrated on the public by the university of the same name.

The Stanford keyboard has (in addition to keys in all the wrong places), a number of "shift" keys. The key labelled "SHIFT" shifts lower case letters to upper case letters. The locking key marked "upper case" is the shift-lock for this key. In order to obtain the special character that is printed on the top of each key, you must use the "TOP" shift key. The "CONTROL" shift key does exactly what a control key always does. However, to talk to the graphics monitor, you use the "META" key, or both the "META" key and the "CONTROL" key together. These will be denoted, respectively, as <M>C or <MC>C, where C is some character. These characters are interpreted directly by the graphics monitor and not transmitted to the host computer.

First it is necessary to have the graphics operational. If typed characters do not echo on the graphics, it means that (1) the PDP-10 is down; wait for it to come back or (2) the graphics monitor is down; reload it. <M>CALL sometimes works if the graphics monitor is not too badly screwed up; if nothing happens, push the little red button on top of the keyboard. This reboots the graphics monitor from the PDP-10. It is not necessary that you have a job logged in on the PDP-10.

Now, if you have just walked up to the graphics, or if you have rebooted it, you must reload a program which enables communication to C.mmp. To do this, you type the command on the PDP-10:

```
INI @CT[N810HY97]
```

During the initialization nothing much appears to happen; this is because the program is being loaded into the graphics monitor, which doesn't display much. After it is loaded, it will connect to C.mmp, and assuming Hydra is running (what? You didn't check that first?) you may now proceed to use the graphics as any other C.mmp terminal.

To return to the PDP-10, type <MC>R <M>T; to switch the connection back to C.mmp, type <MC>R <M>C.

6.1.4 Other kinds of terminals

Hydra understands certain physical characteristics of terminals (no, not that they are blue or white or have funny keyboards...) such as how to move the cursor left, how to clear a video terminal screen, etc. Mostly, these characteristics are "wired in" to the Kernel, which has clever little tables in its device support routines. Someday, the user may be able to specify this, but right now there is so little need for such a feature that Kernel tables are compiled right in. The devices which the Kernel currently understands are:

Model 33 Teletripe^{tm}
Texas Instruments "silent 700" series
Superbee II
Minibee IV
Infoton
Datamedia 2000
DEC LA36 DECwriter

Normally, the Front End informs Hydra which type of terminal is connected; however, its information may be incorrect, so the user has the option of respecifying this information after logging in.

6.1.5 Terminal Keyboard Characters

Certain characters typed at the keyboard have particular effects upon the system. Most of these are what one expects if one is familiar with our PDP-10s, but a brief review is included here for those who are not, or who don't remember all of the functions:

* Control-H or Delete: erases the previous character; if the terminal can backspace, the cursor will be moved left; for video terminals the previous character will also be erased.
* Control-U: cancels the current line (equivalent to a series of control-H characters). On video terminals an attempt is made to erase the entire line.
* Control-I: Tab, moves the cursor to the next tab stop. Tab stops are set every 8 positions. If the device cannot actually tab, a number of spaces are printed to move to the next tab stop.
* Control-K: Just like typing the "break" key.
* Control-L: On video terminals, clears the screen. On hardcopy terminals, just prints a few blank lines (three, we think).
* Control-O: clears output. Current output is flushed and future output is rejected. This mode is reset by typing another control-O or by sending a READ request down to the terminal.
* Control-S: temporarily suspends output. Output stops almost instantly, and is continued by typing:
* Control-Q: resumes output suspended by control-S.
* Control-R: this is supposed to retype the line, but right now it does nothing. Try it; perhaps retype will be implemented by the time you read this.
* Control-T: Gives the status of the Kernel, Terminal Multiplexor, or user job. During Kernel initialization it will print out what the Kernel is doing; if it prints Cold Start Pass 1 or Frigid Start Pass 1 or Frigid Start Pass 2 you have time to go up to the 8th floor vending machines and get some (alleged) food. A Cold Start or Frigid Start will require on the order of 20 minutes. Network users can't get these reassuring messages because the Network software is not running during these periods, so the connection will probably time out. Once the terminal multiplexor is up, it intercepts the control-T and will print out its version number or give some other useful status information. Note that

control-T forces the Terminal Multiplexor to check the Job Monitor it has created to talk to you; if the Job Monitor is dead a new one will be created. When in doubt, try control-T. Eventually, control-T will be reported to the job monitor which will tell you something about the job; this is not yet in but we think we know how to do it, so try it.

* Control-underscore (or control-backarrow, depending upon what your terminal keytop shows for this code). Temporarily suspends the front-end or Network link and returns the terminal to the front end or local host. A reconnection to C.mmp will give the user the same job in more-or-less exactly the same state as before the break (typeahead is not guaranteed to be preserved).
* Control-caret (or control-uparrow, on older terminals). Permanently breaks the front-end connection. The Job Monitor and all the subjobs are deleted. Reconnecting to C.mmp will give a completely new terminal connection including a new Job Monitor. None of the status of the old connection remains, and therefore a new terminal type must be re-established. Usually the Front End will do this correctly, providing it knows the correct terminal type.

6.2. The Hydra Terminal Multiplexor

The Hydra Terminal multiplexor subsystem is the part of Hydra that controls your terminal. Very low-level control, such as processing of rubouts, echoing of characters, worrying about which characters are breaks, etc., is handled by the kernel for reasons of efficiency. The Terminal Multiplexor is the next higher level of control. It takes charge of connecting your terminal to various processes, buffering, and connecting you to the command interpreter (which is in fact just another process).

In order to properly understand the nature of the Terminal Multiplexor and what it can do for you, you must first know something about Hydra Ports and Hydra Connection Objects. Ports are very well documented in the Kernel Manual, but there is no particular written documentation about connection objects. We shall attempt to fill that void until such time as official documentation becomes available.

6.2.1 Connection Objects

Data is transported around the innards of Hydra via ports. These ports are connected to each other by means of port connections. Ports are a very general mechanism, which may be used for I/O, messages, semaphores, etc. A particular use of ports which is of primary concern to the beginning C.mmp user is that of terminal ports.

In Hydra, every real device looks like a Port, i.e., disks, tapes, DECtapes, terminals, etc. However, in the case of terminals, no user ever actually obtains a capability for a device or connects directly to a terminal. Instead, an intermediary known as the Terminal Multiplexor sits between the device and the user, and users connect to the Terminal Multiplexor port. The Terminal Multiplexor handles all of the message traffic to the device, and provides a great deal of power which would not be available if the device were connected directly to the user's process. Since a user can have many processes, the terminal Multiplexor allows the user to talk to whichever one is of interest at the moment.

Output to a terminal is from the user's Port to the terminal "device" (which is really simulated by the Multiplexor). When a process sends a message over a port, the SEND operation will fail unless the appropriate output channel of that port is connected to something. If the user's terminal is connected to process A, and process B sends a message over its terminal port, that message must go somewhere. One of the functions of the Terminal Multiplexor is to provide that "somewhere" at the receiving end of terminal ports which do not currently have terminals attached to them. The Terminal Multiplexor provides a function (the TALK function) which is used to specify which port's output is actually transmitted to the terminal.

The connections between <u>ports</u> and <u>the Terminal Multiplexor</u> are represented by Connection Objects. If you "make" a connection; i.e. if you connect a port to the Terminal Multiplexor, then as a byproduct of doing so, you have created a Connection Object which represents that connection. The routing of Terminal traffic around the Terminal Multiplexor is specified not in terms of the ports, but in terms of the connection objects representing the connection of those ports to the Terminal Multiplexor.

6.2.2 Terminal ports

In your program, input and output to the terminal are programmed as input and output requests to a port. The creation of the port and its connection to the Terminal Multiplexor are normally performed automatically by the programming system runtime support: in BLISS, the HYDUSR runtime initialization creates and connects a port; in L*, it is part of the system initialization. Only those BLISS programmers who choose not to use HYDUSR, and assembly language programmers, need worry about terminal connections.

6.2.3 Coming attractions

If you have more than one process active in your "task force", it is useful to be able to connect your terminal to each or any of them. These connections may be made directly from the terminal to the process in question, or may be made through a chain of ports and connection objects. In either case, hitting the command break (↑K) character causes the Terminal Multiplexor to break the connection closest to the terminal and reconnect the terminal to the command interpreter. If there is a chain of ports linked beyond this first level, that chain will not be broken but will not be accessible either. At the command interpreter, a $TALK(<connection object>) may be executed to reconnect the terminal to some other procedure.

Connections also have some properties of the device associated with them. For example, if you are talking on a given connection and change the device status, such as turning off echo, then that condition will be remembered for that connection. When "break" returns you to a previous connection, the state of the terminal for that connection will be reset so echo may be turned back on (if it was turned on in that connection) and whenever you return to the connection in which echo was turned off, the echo will be turned off. This is all accomplished by reading the device status out from the Kernel just before a connection is broken, and writing it back just as a connection is re-established. The first time a connection is established, the device status is read out, so a new connection "inherits" the characteristics set in its creating process.

There are plans and promises to beef up the Terminal Multiplexor. Some of the features that are planned are the sorts of things that you might find yourself saying "It sure would be nice if we had _____". Even though the User's Manual is not supposed to be a dream catalog, the future plans for the Terminal Multiplexor are included here just to show you what you are missing.

> A LISTEN function, that will allow the Terminal Multiplexor to listen to many processes at once. You may see printed at the Terminal any output from any of the ports to which you LISTEN. Obviously, the keyboard can only be connected to send data to one of them; the TALK function will still control this. LISTEN is currently available to L* users; it is implemented in the L* kernel rather than the Terminal Multiplexor.

> A KILL function, to get rid of connection objects when they are no longer needed. Connection objects all vanish when the system is restarted, so until such time as system restarts become rare, nobody is going to worry too much about KILL.

> A TALKBREAK function. This will behave exactly like a TALK, save for the way it behaves when you hit break (↑K) at the terminal. If a connection chain is made with a series of TALK calls, then a break request will break the one closest to the terminal. If a connection chain is made with a series of TALKBREAK calls, then a break request will break the one farthest from it

7. Programming for Hydra

At the moment there are four ways you can write a program for C.mmp to run under Hydra; i.e., there are four different "programming systems" available. Of these four, two are primarily batch systems (BLISS-11 and Algol-68), one is primarily a conversational system (L*), and the third (assembler) is pretty much like everybody's assemblers everywhere.

7.1. Assembler

You may program C.mmp in PDP11 assembly language. This is not a good way to learn about Hydra, because all of the K-calls were set up so that they could be called conveniently from BLISS-11; as a result, some of the calls look a little clumsy and are obscure at best. George Robertson has put together a collection of assembler macros which provide the bindings and definitions and calls of Kernel facilities. He reports that with some exceptions such as PATH and WALKn, the assembler macro calls are quite reasonable, and he offers an L* kernel coded with them to prove it. (Those daring souls among you can find this file as KMACS.M11[N810HY97]. See also the .REQUIRE directive in MACN11). The Assembler runs on the PDP-10. There is no assembler running under Hydra.

7.2. BLISS-11

C.mmp may be programmed in BLISS-11. BLISS-11 is an "implementation language" which was developed at CMU. The compiler runs on the PDP10, and produces an output file suitable for processing by the PDP11 assembler. Hydra itself is coded entirely in BLISS-11, so the compiler has obviously been worked over and checked out to a certain extent.

7.3. L*

C.mmp may be programmed in L*. The L* kernel handles most of the nuisance details of running a program, such as setting up port connections and talking to the PDP10 link, etc. The Hearsay-II system for C.mmp is being implemented in L*, and the SOS text editor already operational there is implemented in L*. A handler and data recorder for the Audio Spectrum Analyzer (a piece of Speech hardware) was implemented in L* on C.mmp and is currently in production.

For more information on L*, see LSCDOT.DOCA110LC00]/A

7.4. Algol-68

One of the newest features of Hydra is an Algol68 implementation designed by Peter Hibbard, Paul Knueven, and Bruce Leverett. Several attributes of the language make it suitable for the C.mmp/Hydra environment:

1. It is one of the very few high-level languages to support multiprocessing control structures, e.g. **par begin** ... **end** and semaphores. Applications written in Algol on C.mmp will therefore be able to use all the processing power of C. without explicit interaction with the Kernel or Policy Modules.

2. The crucial ideas of TYPE and PROCEDURE in Hydra are closely paralleled by the **mode** and **op**(erator) concepts in Algol68. Thus Hydra users will find Algol68 natural and vice versa.

3. Finally, the C.mmp implementation is well suited for minicomputers. It resulted from an implementation designed by Peter Hibbard for an English minicomputer. Thus, although Algol68 is considered difficult to implement in America, the implementation on C.mmp is remarkably mature.

It is intended that Algol-68 will be the primary user system. Unlike BLISS-11 and Assembler, Algol-68 does not require the use of the PDP-10 to produce an operational program. Programs are created, compiled, linked, etc. on C.mmp/Hydra.

8. Introduction to Algol 68

The C.mmp Algol 68 system accepts a PAGE, UNIVERSAL of PAGEs, or SuperFile as input, compiles this source text and invokes the run-time system which executes the program. Eventually this compile-and-go system will give way to a compile-link-and-go system which supports separate compilation and program libraries.

8.1. Preparing source input using SOS

A version of the SOS editor exists on Hydra. The PDP-10 SOS Manual serves as the user manual for the SOS editor. However, only a subset of PDP-10 SOS is available. In particular, the Copy, Transfer, Find, Substitute and Justify commands are not implemented.

A new file may be created and edited using SOS by typing: Create(). You will be prompted for a file name. Your reply should be a name of one to ten letters and/or digits. (Other characters may be used, but some are rather dangerous.) SOS starts in insert mode.

An old file may be edited using SOS by typing: Edit(). Your reply to the prompt for a file should be the name given when you created the file or a null reply (i.e. carriage return) in which case the file most recently edited or compiled will be used.

SOS may be used to examine a file in read-only mode by typing: Read().

Additional information about SOS may be found in Chapter 9. Problems with and complaints about the editor should be sent via MAIL to A110LC00 on the PDP-10.

8.2. Preparing source input using TECO

A version of the TECO editor exists on Hydra. It is roughly a subset of its namesake on the PDP-10. A new file may be created and edited using TECO by typing: Make(). The prompt for a file name should be answered with a name consisting of one to ten letters and/or digits. An old file may be edited by typing: Teco(). You will be prompted for a file name. A null response causes the last file edited or compiled to be used.

8.3. Specifying files and file names

If you have created any programs using SOS or TECO, your catalogue now has an entry called Algol. This is itself a catalogue and there is an entry in it for every program you create. When you use one of the standard commands, such as "Edit" or "Teco", and it prompts you for a file name, it looks up that file name in your Algol Catalogue. If you want to deal with a file that isn't in your Algol catalogue you can do that, too; the only requirement is that you specify completely, using the conventions of the CI, how to access the file. For instance, if your user catalogue has an entry called Test, and Test is a catalogue with an entry called Prog which is your program, then when "Edit" or "Teco" or whatever prompts you for a file name, you should type: Test.Prog. If your program isn't even on a catalogue but is in a "Capability variable" called &prog, then you should type: &prog.

You can bypass the prompt by passing the proper indication of your program directly to the command, as a parameter. For instance, you might type:

```
Edit(Algol.prog)
  ! Equivalent to:
  ! Edit()
  ! Source file: prog
Edit(Test.prog)
  ! Equivalent to:
  ! Edit()
  ! Source file: test.prog
Edit(&prog)
  ! Equivalent to:
  ! Edit()
  ! Source file: &prog
```

Notice that these commands "remember" what the last file you edited, ran, etc. was. That is, if you respond to the prompt by not typing anything but carriage return, it is as if you typed the name of the last file you edited or ran. The commands do this by maintaining a "Capability variable" called &currentfile. At any given time, this variable is set to the file currently in use, or the last file that was in use. Ordinarily you don't need to know this, but for the curious, that's how it's done.

Every catalogue entry (indeed every object in the Hydra system) has a type. Files created by SOS are of type SUPER FILE; files created by Make() are of type UNIVERSAL. You'll notice that the type of each file is printed out when you get a catalogue listing (see below) of all your files. SUPER FILE cannot be edited with TECO, and UNIVERSALs cannot be edited with SOS, in case you were wondering.

8.4. Compiling and executing a program

First, type: Alg68(). The following prompt will be typed:

```
Source Input:
```

The response should be the name of the file to be compiled and executed. The dialog continues with the prompt

```
Listing Device:
```

This is the first of several "option prompts" which are part of the standard dialog. For each option prompt, there is a set of possible replies, and if you forget any of these, you can have the system type out the whole set, by replying with "?". Any reply may be abbreviated to only its first few characters, enough to distinguish it from all other replies in the set. A reply must be followed by a carriage return.

Currently there is only one possible reply to the 'Listing Device' option prompt. This is "TTY", indicating that a compilation listing is to be typed at the user's terminal. A null reply (i.e. a bare carriage return) indicates that the listing is to be suppressed. In the future, more replies (e.g. "LPT") will be available.

Next, there is another option prompt:

```
        Compiler Option:
```

This prompt is repeated after each reply, until the user types a null reply (carriage return). A list of the available replies may be found in Section 8.5.1. After the null reply to this prompt, the compilation begins.

When the compilation is complete, a message stating the program's code and data sizes (in number of words) is typed on the terminal. If no compilation errors occurred, the run-time system is called. First, however, there is another option prompt:

```
        Runtime Option:
```

Like the previous option prompt, this one is repeated after each reply, until the user types a null reply. A list of the available replies may be found by typing ". One of the replies is special: 'STANDOUT'. This brings on the 'Standout Option' option prompt, by which means the user specifies what is to be the nature of the standoutchannel, the channel on which the standard output file is opened. The available replies are:

LIST	It is a file of the LPT subfile type, which is listed as soon as it is closed.
TYPE	It is the same as Consoutchannel, the channel for output to the user's terminal.
SAVE	It is a file of the SOS subfile type. More about this later; this is the default.
DELETE	Not implemented yet.

The run-time system then greets the user with some reassuring message and commences program execution. After the program finishes, another reassuring message appears at the terminal indicating that execution is complete. If the user has earlier specified the 'SAVE' Standout option (or has not specified an option-- this is the default), the accumulated output from the program using standoutchannel is now available as a file of the SOS subfile type, and the system gives the 'Final Standout Option' option prompt. You may specify that the file be typed on your terminal, listed on the line printer, saved in your userdirectory under the name 'Standout', or thrown away (it is biodegradable).

If you run a program more than once without editing it, you can shorten the compilation-execution sequence considerably, by avoiding more than one compilation. To do this, first type: Com68(). You should do this whenever you have done some editing and are ready to try out the program again. This runs the Compiler, which produces an Object Program from your program. If you have a subcatalogue called Object on your userdirectory, the Object Program will be put there; otherwise it will be put on your userdirectory. To actually run this program, type: Run68().

8.5. Terminating Algol 68

If your program gets into an infinite loop, or some other mishap befalls it, press the BREAK key (Control-K), and then type Control-C. Ordinarily, this will cause the program to be interrupted and forcibly stopped. You can also interrupt the compiler this way, if you want to.

8.5.1 Compilation Switches

Some of the compilation options listed below are not of interest to users; these are marked with an asterisk. Of the others, currently all are available as pragmat-items as well. For instance, if your program contains the pragmat

```
10:: lower ::
```

the remainder of the program (at least up to the next pragmat) will be assumed to use the "lower" stropping convention.

DEBUG*	Enable compiler debugging mode of operation.
GHOST*	Mumbo jumbo.
LISTING	Produce compiler source listing output.
LOWER	Use lower stropping convention.
NAKED*	Hocus pocus filiocus.
NODEBUG*	Disable compiler debugging mode of operation.
NOGHOST*	Disable mumbo jumbo.
NOLISTING	Suppress compiler source listing output.
NONAKED*	Disable hocus pocus filiocus.
NOWARNINGS	Do not output warning messages.
POINT	Use point stropping convention.
RES	Use reserved word stropping convention.
UPPER	Use upper stropping convention.
WARNINGS	Output warning messages.

8.5.2 Execution Switches

DEBUG	Enable run-time system debugging mode of operation. This is of some limited usefulness to users. The user is prompted for a set of 'Flags', that is, a number; this number is treated as

	if it had been passed as the second argument to a call on Systrace (see documentation elsewhere) in the program.
PROCESSES	The user's program is to be run on more than one process. The user is prompted for a number of processes, which may be from 1 to 16. Note that this is not a number of physical PDP-11 processors, but of HYDRA processes; and that at any time the user's program may make use of fewer processes than the specified number, or it may be written as if it could make use of more; in either case it should still run and produce correct results.
SPEEDUP	Consult the implementors before using this switch.
STANDOUT	See the explanation in the Algol 68 user's manual.

8.5.3 Error Reporting

A compilation error is indicated by the printing of a vague message, the line containing the error, and a line containing a position indicator beneath the first character of the most recently seen input lexeme at the time of the error. A position indicator is simply a digit printed in the appropriate position. If more than one error occurs in a given line, the error messages, first to last, are associated with the position indicators, left to right, respecting the following rule. If many errors occur at the same position, the digit printed indicates the number of errors which occurred there.

Syntax errors will normally cause some portion of the input to be ignored. For the most part, any ignored characters are printed on the listing with equal signs beneath them.

No semantic processing of the program is done after the first compilation error occurs. Naturally, the run-time system is not called if an error is found.

There are also abnormal situations which are not quite as severe as errors. These cause a warning message to be printed in the same format as an error message except the fact that it is a warning is noted. Warnings do not cause semantic processing to stop.

Run-time errors are normally indicated by the printing of an accurate statement of what the problem is. This is accompanied by an indication of the source program line which was being executed when the error occurred.

8.6. Temporary Restrictions

8.6.1 Things You Care About

- Soft balancing of identity-relations is not implemented.
- Transput of structured, *long int* and *long real* values is not implemented.

- Some operators are not implemented . In particular no operators which involve any of the modes @B[long int], @B[long real], or @B[long compl] are implemented.

- Some standard and particular prelude identifiers are not implemented

- Program code size is restricted to 4096 words. This should be able to acçomodate 200 to 300 source lines of program.

8.6.2 Things You Do Not Care About

- Full and correct mode equivalence check is not implemented. Do not be worried about this. The mode equivalence algorithm is almost totally correct. An appropriate prize will be awarded to the first user discovering two equivalent modes which are not recognized as such.

- Determination of necessary environ based on use of applied-mode-indicant as actual-declarer is not implemented completely.

8.7. For more information

Some of the restrictions mentioned should be disappearing in the latter part of 1977. For up to date information, see HYDA68.XGO [L150AL72]

Part II: Utilities

9. SOS on C.mmp

The C.mmp SOS subsystem is composed of two parts: the SOS subfile system, and the SOS editor. The first part of this chapter describes the SOS editor. The second part describes the SOS subfile system with the assumption that the reader has read the C.mmp File System documentation.

9.1. C.mmp SOS Editor

A subset of SOS is now available on C.mmp. The SOS manual for the PDP10 version of SOS can be used as a manual for this version also. This document lists the differences between the two systems. Some of the differences will be removed in the near future, and others may remain for some time. If you encounter any differences not reported here, please report them by sending mail to [A110LC00].

9.1.1 DIFFERENCES

Differences are listed in the order that they will be removed:

1) No transliteration on input and output.

2) Line numbers are only 15 bits (i.e., max line = 32767, or 327 lines with INC=100).

3) Copy, Transfer, Find, and Substitute are not yet implemented.

4) Associative line numbers are not yet implemented.

5) Line mode justification is not yet implemented (and may not be for some time.) (Alter mode justify is implemented.)

6) File name references not implemented:

```
=FILE
←FILE=filename
Cnnn←filename
Wfilename
Efilename
```

9.2. Utility Command Objects and Procedures

A number of command objects have been supplied that will aid in the use of SOS files. If you have any suggestions for additional command object utilities, please send your suggestions to [A110LC00].

In the Commands object calls below, it is assumed that the Command Language macro

```
PUB=&SYSDIRECTORY.PUBLIC
```

has been defined. (For more information about the Command Language, see Chapter 20.

9.2.1 PUB.SOS.EDIT(&file)

enter the SOS editor with arbitrary file &file. If &file is not an SOS file, it is converted to an SOS file before entering the editor, then converted back after leaving the editor,

9.2.2 PUB.SOS.READ(&file)

enter SOS editor in read-only mode with file &file.

9.2.3 PUB.SOS.EDIT()

will prompt for file name (File=). Your response can be exactly as for SOS on the PDP10. That is, filename/N filename/R filename/N/R or filename$ where $ is an altmode. For the last case, a new file is created and the filename supplied is used only as the file print name; the created file is returned after exiting from the editor. For this case, the call should be: &file = PUB.SOS.EDIT() in order to retain the capability for the new file. Note that filename in the other cases is actually the catalogue entry name in the user's catalogue.

9.2.4 PUB.SOS.STATUS()

prints some status information about the SOS monitor. If it indicates that the monitor has stopped, please get in touch with George Robertson. The normal state is Port Wait unless someone is actually doing transput, in which case the state will be Running.

9.2.5 &file = PUB.SOS.FTPSOS()

will transfer a file from CMU-10A across the ARPA-net (assuming the NCP is running). It prompts for a file name, which must be fully specified. E.g., temp:file.ext[a110lc00] Note: line numbers are not transfered, so you must make sure that they have been stripped from your PDP10 file.

9.2.6 &file = PUB.SOS.DTASOS()

will transfer a file from dectape (DT11 format). It will prompt for processor number, unit number, and file name. The file name must be six characters (padded with spaces if necessary) followed by dot (.) followed by three characters. The file should have been placed on the dectape with DT11 using the /A switch (ascii). The file should not have PDP10 line numbers in it.

9.2.7 &file = PUB.SOS.UTOSOS(&univ)

will produce an SOS file from a universal object (&univ) of pages.

9.2.8 &univ = PUB.SOS.SOSTOU(&file)

will produce a universal object of pages from an SOS file.

9.2.9 PUB.SOS.SOSDTA(&file)

will put an SOS file onto a DT11 dectape.

9.2.10 PUB.SOS.SOSFTP(&file)

(Not Yet Implemented) will transfer an SOS file to CMU-10A.

9.3. C.mmp SOS Subfile System

The SOS subfile system provides support for creating, copying, querying, editing, reading, and writing line numbered text files. The line numbers are "sticky", and remain with the file until they are altered by editing the file. The most common file read and file write operations are implemented so that the user need not be aware of the line numbers; making it possible for homogeneous I/O between files with different subfile representations. Each of the allowable operations will be illustrated below using command interpreter expressions where possible.

For more details, see one of the following BLISS require files:

FILES.REQ[N810HL00] - for operation codes, reply types, query codes, and message formats,

FILEUS.REQ[N810GA10] - for typecall indices, file system signals, and file system auxiliary rights,

SOS.REQ[A110LC00] - for SOS signals and error codes.

9.3.1 File Creation

There are two ways to create a new SOS file. The first is the file system create typecall:

```
0>capa &file
0>&file = typecall(pub.sos.sosfile, 2, pub.sos.sosfile)
%where pub = &sysdirectory.public%
```

The second is through the SOS editor:

```
0>capa &file
0>&file = pub.sos.edit()
File=newfile$
*
   <series of SOS commands>
*e
```

where $ = altmode. Note that the file name used (newfile) is used only for the print name of the generated file (retrievable through a query operation) - the file is returned as a capability.

9.3.2 File Copy

File copy works as described in the file system documentation.

9.3.3 File Query

In addition to the information supplied by the file system, the SOS subfile system will supply the number of Hydra-page objects needed to represent a file.

9.3.4 File Edit

The file system Edit Typecall will get you into the SOS editor (for an SOS file). It requires a data object with editor options encoded. If the user uses the standard command objects (described below) he need not be concerned with the format of this data object. The format is:

word 0, bit 0 - /N (no line numbers printed)
word 0, bit 1 - /R (read-only)
word 0, bit 2 - $ (new file - enter insert mode at 100/1)

word 0, bit 3 - /KEY (key supplied - position file and print line)
words 1 to n - asciz key if /KEY option (line and page)

The call on the editor for file &file and data (options) object &data is:
0>typecall(&file, 10, &data, &file, &ttycon)

9.3.5 File Open

The SOS subfile system currently supports IdxOpenRead, IdxOpenWrite, IdxOpenSpool, and IdxOpenNew. Update and Append modes are not currently implemented.

9.3.6 File Close

Works as described in the file system documentation.

9.3.7 Operations on an open SOS File

Only message format 1 (pre-allocated key and record positions) is supported. The following operations are supported:

FileRead	sequential read. Will fill your supplied text buffer with as many lines as will fit. It will not split a line across requests, so if the next available line will not fit in the text buffer, you will get an error reply. An SOS-page mark will be translated into a form-feed character. End of file is indicated by a special reply type, and the text buffer will be empty for that message.
FileWrite	sequential write. Will take your supplied text buffer (with arbitrary number of lines and possibly a partial line at the end) and produce sos lines with generated line numbers (increment 100). Null characters are ignored. Bare line feeds are translated into carriage-return-line-feeds. Form-feed will produce an SOS-page mark. A partial line at the end of a text buffer is held onto until the next write request or until a flush or close operation.
FileRewind	position to start of file.
FileToEnd	position past end of file. Not yet implemented.
FileFlush	update data structures on secondary store. Will cause partial line during filewrite series to be moved into the file as a complete line.

FileReadGivingKey sequential read returning key. Not yet implemented. Will read exactly one line into your text buffer and return the line number and page number in an asciz key.

FileWriteGivingKey sequential write returning key. Not yet implemented. Will write exactly one line into the file and return the asciz key generated by the write.

FileSeek position file to specified asciz key. Not yet implemented. Will position to line and page number if they exist, otherwise, one past where they would be.

FileSeekRead seek followed by read. Not yet implemented.

FileSeekWrite seek followed by write. Not yet implemented.

FileInputClear release any held messages.

9.4. Random Comments

There are a few other features that are currently not supported but should come along shortly.

There is no easy way to do the equivalent of ↑C followed by CONTINUE or REENTER during an editing session. At some point in the future, ↑C and ↑K will be trapped in order to provide this function.

If the system crashes during an edit, there is no way to retrieve the temporary file that was being used. Please send comments and suggestions to [A110LC00].

10. TECO

A version of TECO, a subset of the PDP-10 program, exists to run under Hydra. It can edit a virtually unlimited amount of core (over 100 pages). It edits a universal object with pages in it.

There are commands objects in &SYSDIRECTORY.UTILITIES which can interactively invoke TECO.

```
TE()
for pages or
EDITCMD()
 for commands objects.
```

To create a new universal and put text into it, use the following sequence of commands:

```
CAPA &name
$MAKEUNIVERSAL(&name)
&SYSDIRECTORY.PROCEDURES.TECO(&ttyport,&name)
```

You may now edit happily away.

10.1. Hydra-teco commands

The command language of this teco is very similar to that of Teco on the PDP-10 however only a subset of the features exist. In this light only a quick description will be given of the commands which are identical.

Further information may be found by consulting:

1) PDP-10 USERS HANDBOOK
2) CMU Introduction to the PDP-10
3) CMU PDP-10 Teco quick reference guide

These commands are identical on the C.mmp and the PDP-10:

B	0, buffer origin.
Z	character count of the buffer, buffer end.
H	B,Z
nC	skip n characters.
nD	delete n characters.
E	exit.

nFSa$b$	find "a" and substitute "b" (n finds, 1 substitute).
Ia$	insert text "a" into the buffer.
nI$	insert ascii n into the buffer.
nJ	jump to the n'th character in the buffer.
nK	kill n lines.
m,nK	kill characters between buffer position m and n.
nL	skip n lines.
nR	skip n characters backwards.
nSa$	search for the n'th occurence of "a".
nT	type n lines.
m,nT	type characters between buffer position m and n.
;	leave iteration if preceding S or FS fails.
n<>	do commands enclosed in <> n times (forever if no n).
n==	type the value of n in octal.
n=	type the value of n in decimal.
n+m	the value of n + m.
n-m	the value of n - m.
↑R	quote the following character, in this way <ESC> and ↑C can inserted and searched for.
#	like the pdp-10 TECO ↑O meaning take the following number as octal.

In addition the S and FS commands will take a negative argument causing them to search backwards. A backwards S (-S) will leave the buffer pointer at the beginning of the string it finds. If a search fails the buffer pointer is left where it was, as in SOS.

10.2. Q-registers

Teco has q-registers which behave almost identically to those in PDP-10 teco. Internally there are actually two sizes of q-registers but these are completely transparent to the user. This was done in the interest of speed.

There are 36 q-reg's: the letters A-Z and digits 0-9.

The commands available for q-registers are:

nUm	store the integer n into q-reg m
Qm	equal to the integer stored in q-reg m
nXm or i,jXm	store the next n lines into q-reg m or store characters between buffer positions i and j into q-reg m
Gm	put the text in q-reg m into the text buffer at the current pointer position.
Mn	execute the text in q-reg M as a teco command string. For now the user must be careful not to execute an 'I' command with a string longer than 8192 characters. The insert command has not yet been modified to take longer strings.
*m	as the first command in a command string. Store the last command string in q-reg m. this is a loss recovery technique.

10.3. extended features

There are some commands in C.mmp Teco which do not exist on the PDP-10, these are as follows:

1P	type out a map of your text pages at the screen top after every $$
P	type out the page map and turn off page map typeout.
↑W	must be used to "fool" the command interpreter into allowing this pdp-10 construct:

```
FSa$$
FSa$↑W$ must be done instead.
```

Note that this is only necessary in an iteration.

Three SOS-like commands exist:

<↑A>	as the first character in a command string is equivalent to the teco command string '-LT' or go to the previous line and print it.
<LF>	as the first character is equivalent to 'LT' or go to the next line and print it.
<CR>	as the first character is equivalent to '0LT' or go to the beginning of this line and print it.

10.4. implementation details

The command input buffer is 4000 characters long. Teco will warn you when you are less than 150 characters from the end. If you do not heed this warning by typing $$ (executing the command) execution will begin automatically after reading the 3999th character. Then any characters that were ignored as a result of getting input line at a time will be printed.

On exit, teco writes the byte count of the buffer size into the first two words of the data-part of the universal object it edited, least significant word first. Note that teco does not use these numbers itself but places them there for the benefit of other programs.

If ↑K↑K (or ctlprocess(&pros,-1)) is typed it will have various effects depending upon what teco is currently doing. The possible actions are:

a) During typeout: return to command mode.
b) At the beginning of any command execution: return to command mode.
c) Anywhere else: no action is taken.

The inner search loop in teco can search for a character it will never match at approx. .2 seconds per page. This includes time spent shuffling pages.

11. Obinfo: Object information

OBINFO is a Hydra Procedure which permits interactive inspection of capabilities (rights, flags, etc.) and objects (C-list and data-part). It is possible to travel down "paths" through C-lists and Catalogues, looking at capabilities and objects along the way. Special object-type-dependent facilities are provided, for instance the ability to look at the contents of PAGE objects.

OBINFO never modifies the objects it looks at.

11.1. Features

OBINFO prints out kernel rights by name; auxiliary rights are printed by name for those object types known to OBINFO and as a binary bit mask for the others. Typing "RIGHTS ALL" will get you a complete list of the kernel and auxiliary rights for the object being looked at.

OBINFO allows the inspection of an object's data part (rights permitting). Typing "DATA" causes OBINFO to enter a submode in which commands of the form "i!j" (and "n:m") will display the j words beginning with word i (or the words n to m) in a variety of formats. The formats can be changed. Typing a blank line terminates the submode.

OBINFO has a submode allowing inspection of PAGE objects. Enter the submode by typing "PAGE", and commands of the form "i!j" and "n:m" are accepted, where the numbers are byte offsets and counts.

OBINFO can walk paths through the C-lists of objects, including Catalogues. Typing "DOWN 3" will let you look at the object whose capability is in C-list slot 3 of the current object. If the current object is a Catalogue, typing "DOWN FOO" will let you inspect the object stored as entry "FOO" in the directory. This descent may be continued indefinately. To ascend back up the path, type "UP". To find out where you are, type "STACK".

OBINFO returns the capability it is currently looking at when it is exited.

To get a complete list of commands, type "HELP" to OBINFO. Experiment!

11.2. Accessing OBINFO

A capability for the OBINFO procedure may be found in &SYSDIRECTORY.PUBLIC.HARBISON. It takes two arguments: the first is a port connected to a terminal with "&TTYPORT conventions", the second is a capability of any type to be inspected.

Examples:

From the Hydra Command Interpreter:

CAPA &OB &OB←&SYSDIRECTORY.PUBLIC.HARBISON.OBINFO &OB(&TTYPORT, x)
where x is a capability for the object to be inspected.

From a user procedure:

(Assuming a capability for the OBINFO procedure is in LNS slot o, a capability for a terminal port is in slot p, and the object to be inspected in in slot i.)
$CALL(O,o,p,i)

Of special interest is using OBINFO to debug procedures. In this situation, a call of the form

$CALL(O,o,p,$LNS())

is most helpful.

12. PSIGNAL

PSIGNAL is a set of Hydra objects that will attempt to generate English text versions of octal Hydra signal values. It exists as a Hydra procedure, which lives in &SYSDIRECTORY.PUBLIC.Everhart as KSigName, and a Commands object, which lives both in that PUBLIC directory, and in &SYSDIRECTORY.UTILITIES. The Commands object is merely a command-language interface to the procedure object.

The Hydra procedure PUB.Everhart.KSigName takes as argument a DATA object containing two words, and returns a DATA object containing ASCII data; the value it returns is the number of bytes stored in the DATA object. The first word passed to the KSigName procedure is the signal whose interpretation is desired; this must be a negative number for the procedure to function properly. The second word is the Kernel's value for $SIGDATA, and is optional; absence is indicated by a zero value.

As stated, the Commands object serves as an interface between the CL and the KSigName procedure. It may be invoked with between zero and two parameters: if no parameters are passed, it will prompt with "Signal value: " and do an INTERPRET(). If one parameter is passed, it is used as the signal value to be passed as the parameter to KSigName, and the second parameter ($SIGDATA) is defaulted to zero. If two parameters are used, the first is used as the signal value, and the second as the $SIGDATA parameter. In all cases, the Commands object PSIGNAL will OR in #100000 with the signal value. Note that the KSigName procedure itself does not do this.

The recommended method for using the PSIGNAL system is to generate a macro, probably called PSIG, which is

```
macro PSIG = &SYSDIRECTORY.UTILITIES.PSIGNAL $;
```

In this fashion, PSIG is a capability, PSIG() will prompt for a signal value, PSIG(#1234) will decipher #101234 as a Hydra signal value, and PSIG(#1234,#2345) will do pretty much the same thing (at this point).

KSigName is periodically updated to provide the lastest interpretations of Hydra Kernel and user library signal values.

17. Hydra FTP

17.1. Available Commands

The Catalogue &SysDirectory.Public.NCP contains a number of useful Commands objects which perform various file transfer functions between C.mmp and other ARPANET hosts. They all prompt the terminal for the necessary information; none take parameters.

&SysDirectory.Public.NCP.NetCmd	Retreives an ASCII file from a remote host and stores it in a Commands object.
&SysDirectory.Public.NCP.NetPutCmd	Sends the contents of a Commands object to an ASCII file at a remote host.
&SysDirectory.Public.NCP.NetSOSFile	Retreives an ASCII file from a remote host and stores it in a SOS SuperFile.
&SysDirectory.Public.NCP.NetPutFile	Sends an ASCII SuperFile to a remote host.
&SysDirectory.Public.NCP.NetProc	Creates a PROCEDURE object from a ".PAG format" file at a remote host.
&SysDirectory.Public.NCP.NetStore	Sends a "UNIVERSAL of PAGEs" to a remote host.

Note that no facility is currently provided at allow a "LOGIN" at the remote host. For CMU PDP-10s, this means files will be accessed by a job running as [N900AR00]. When shipping data from a PDP-10 to C.mmp this is not usually a problem, because the default protection for files is 5 in the third digit. However, when sending data TO a CMU PDP-10, the file should be created in advance with a protection code whose third digit allows writing by [N900AR00]. Typically <111> is appropriate. The protection <000> is funny, and should not be used without understanding all the implications thereof. If the data being sent is sensitive, it may be protected by pre-creating an empty file with protection <111> and then specifying a filename to C.mmp which ends with <xxx> where xxx is the protection actually desired.

17.2. PageFTP

The procedure at the heart of the above Commands objects is

&SysDirectory.Public.NCP.PageFTP

which is called from the Command Language as follows:

```
word &Code;
```

```
UnivRcvd, &Code = PageFTP(UnivToSend,
                          $StackData(Host, Format, FileName),
                          TTYPort, Job)
```

where:

UnivRcvd If a "RETRieve" operation is performed, then this is the returned capability for a "UNIVERSAL of PAGEs" which contains an array of bytes in successive pages in its C-list. The data-part contains a 32 bit byte count (low order word first) for the data received. UnivRcvd is returned <u>null</u> if a "STORe" or "MLFL" operation is performed.

&Code is the integer value of the procedure call. If the transfer succeeds, it is decimal 200. If the transfer fails, this code is a (possibly ambiguous) description of the cause of the failure. The Commands object call &SysDirectory.Public.NCP.TypeFTPError(&Code) may be used to get a semi-intelligible interpretation of the code.

UnivToSendThis optional parameter should be present <u>only</u> if one desires to send data to another host. It must be omitted otherwise. UnivToSend is assumed to be a UNIVERSAL of PAGEs, similar in format to UnivRcvd. If the byte count is missing, a semi-reasonable assumption will be make on the basis of $CLength(UnivToSend).

Host is the ARPANET host number from (or to) which data should be transferred. The Commands object &SysDirectory.Public.NCP.GetHost may be used to obtain this value from the terminal. Typical values are 78 for CMUA, 14 for CMUB, and 142 for CMUD.

Format indicates the format of the data to be transferred. This is composed of several bit-flags:

Format<0,1>is off for ASCII and on for IMAGE mode. IMAGE mode is actually a special mode for retreiving .PAG, .OBJ, or similar files which are stored on a PDP-10 as two PDP-11 words per PDP-10 word, right justified in each halfword. The transfer is done in 36-bit image mode, with PageFTP removing (or adding) the four padding bits in each PDP-10 word.

Format<1,1>if on, indicates that "MLFL" should be used instead of "STOR" for the transfer. This mode is reserved for use by implementors of mail systems and may change without notice. Anyone who attempts to use this (without contacting Rick Gumpertz first) will be considered a willing victim of whatever nasty behavior happens.

Format<2,1>if on, suppresses typeout on TTYPort.

FileName	is the name of file at the remote host. FileName should contain no NUL (zero) characters as they will be interpreted as the end of the FileName. For CMU PDP-10s it may be any ASCII string acceptable to PIP, such as "ALL:FOO.BAR[C410XX00]<123>". If no PPN is given for a CMU PDP-10, then [N900AR00] will be assumed by that host.
TTYPort	must be a port which, like &TTYPort in the Command Language, has channel 1 connected for terminal output and channel 0 connected for terminal input. This port will be used to indicate the progress of the file transfer along with any error messages received from the other host. If Format<2,1> is on, however, then the only references made to TTYPort will be those made by Six12.
Job	is a Job object from which a subjob may be created. This subjob will be used to register those resources (such as the ARPANET connections) which must be released if the user logs out before PageFTP completes. &JobProcess in the Command Language is such a job.

Refer any comments or questions to Gumpertz@CMUA

Part III: Subsystems

18. Conventions for Typecall specifications

The typecalls in this manual are specified using the same format that is used for kernel calls in the Hydra Kernel Reference Manual. The typecall specifications consist of several parts:

> The typecall name and formal parameter list. These are described in terms of Bliss macros. In some cases the typecalls have been predefined in the Command Language (see section 5.6.2). When a predefined typecall which returns a capability is called from the Command Language, the formal parameter for the returned capability is not used. Instead, the returned capability is the the value of the call. So a call of $Jobname, for instance, would look like

```
&DIndex = $Jobname (Job)
```

instead of

```
$Jobname (&DIndex, Job)
```

> The 'parameters' section. Each parameter is listed, along with the following information:

> A 'path index' or 'simple index' specification. A path index can be a capability path or a Catalogue walk, and a 'simple index' must be an LNS C-list slot, if the call is made from a procedure.
> An (optional) indication of what type of object the capability should be referencing.
> The minimum rights the capability should possess.

The names 'SPath' and 'SIndex' are sometimes used to indicate 'source' capabilities; likewise, 'DPath' and 'DIndex' are used to specify destination capabilities.

> The 'Effect' section describes the effect of the typecall if no signal occurs.

> The 'Signals' section describes specialized signal information for the particular typecall. All chapters contain a list of signals for the object type described.

> 'Result' is the value returned by the typecall assuming no signal occurred. In the case of a Command Language call, this value is returned in the predefined variable &Retval (see sections 20.5 and 5.6.2).

19. The Hydra Catalogue Subsystem

19.1. Introduction

The Catalogue system implements the abstraction of symbolic access to capabilities. Historically, it is a replacement for Bill Corwin's Directory system; many of the Catalogue system's features are merely borrowed from successful Directory system features, while others directly address its more annoying failings. The basic idea is to implement a graph-structured set of nodes. Each node, called a catalogue, is a collection of Entries, each mapping a symbolic name to a capability. The graph-structure arises whenever these entries address other catalogues. Catalogue system operations, implemented as Hydra procedures, serve to manipulate these catalogues and entries. Criticism of the system should be sent to ALMES@CMUA. Several symbols peculiar to the Catalogue system can be found on CATALOGUE.REQ[N810GA10] on CMUA.

19.1.1 Basic Concepts

19.1.1.1 Structure of a Catalogue

A catalogue appears to the user as a set of entries. Each entry has four fields:

1. The Name, one to ten alphanumeric characters. Within the context of a particular catalogue, the name uniquely specifies an entry.

2. The Value, either an object capability or template. The principal function of the system is the mapping of name to entry to value.

3. The Protect boolean which, if true, insures that the entry will not be deleted from the catalogue. It may be thought of as representing $DELETERTS for the value, except that it may be set true or false. It thus provides a function similar to level "2" protection on TOPS-10.

4. The Environment boolean. For Hydra technical reasons, any value capability must have $ENVRTS. If the environment boolean is false, any Lookup through the entry will restrict $ENVRTS in the copy of value returned.

In order to speed the mapping from name to entry, the entries are ordered alphabetically by ASCII representation of the name. That is all the user-visible data structure. Although it borders on being an implementation detail, we do have a semaphore object per catalogue to lock the object during modifying operations. The data are arranged, however, so that the semaphore is never needed during a Lookup operation; consequently a Lookup need never block.

19.1.2 Auxiliary Rights

A detailed specification of the meanings of Catalogue Auxiliary Rights is given later as the operations are specified. A summary is presented here, however, to introduce their names and an informal description of meaning:

CataLookupRts	#001	Controls Lookup and steps in a path
CataEnterRts	#002	Controls Enter
CataDeleteRts	#004	Controls Delete and over-write during Enter
CataListRts	#010	Controls Catalogue List
CataRenameRts	#020	Controls Rename
CataProtectRts	#040	Controls Protect
CataCopyRts	#100	Controls Copy
CataDestroyRts	#200	Controls Destroy

19.1.3 Names, Paths, and Targets

As mentioned earlier, the catalogue system allows a graph structure to be constructed and, for each node in this graph(i.e., catalogue), defines a mapping from name to value capability. These two properties naturally combine to produce the notion of a path name, rather similar to a Hydra path index (cf. Hydra Reference Manual, Section 2.1.2). Thus, the path name &sysdirectory.public.almes.catalogue is a four level path similar to the $Path(3,4,2,1). Adapting Kernel terminology, we will speak of &sysdirectory.public.almes.catalogue as the <u>target</u>, of the catalogue &sydirectory.public.almes as the <u>pretarget</u>, and of &sysdirectory and &sysdirectory.public as <u>steps</u>.

In another important adaptation of Kernel notions, each of the modifying operations of the catalogue system require $MODIFYRTS and CataLookupRts (cf. use of $MODIFYRTS and $GETCAPARTS in the Hydra Reference Manual, Section 2.15.1.2). Further, they require $MODIFYRTS and some operation-specific auxiliary right on the pretarget. Finally, when we speak of an entry being <u>empty</u>, we mean that no entry exists with the name in question, not that some entry exists with a Null template for its value.

19.1.4 Details of the Name Parameter

Most of the catalogue operations take, as a parameter, a Data object containing a path name. There are a few details of such a parameter that are needed for practical reasons. First, the Data object must be no more than fifty-five words long; note that this limits the length of a path name to between ten and fifty-five levels, depending on the length of the various entry names used. The Data object must contain the path name as a series of entry names (of one to ten alphanumerics) separated by periods. If this path name doesn't fill the Data object, then a null character (zero) should follow the path name; any characters following this null are ignored. Finally, we insist that any name parameter be passed with $GETDATARTS.

19.2. Catalogue Operations

In the following specifications, the symbol CataRep will mean any template or object capability of type Catalogue; NamePar will mean any Data object capability meeting the requirements of 19.1.4 above. If the name parameter is syntactically improper, the signal CataErrName is given. If the rights for the catalogue or catalogue entries along the steps or pretarget are insufficient, the signal CataErrRights is given. Other signals are operation-specific and are dealt with individually.

$MakeCatalogue (NewCata, CataRep)

Parameters:

NewCata -	Simple index, empty.
CataRep -	Path index; a Catalogue object reference or template.

Effect:

NewCata is the capability for a newly created empty catalogue, returned with all auxiliary rights, $ENVRTS, $MODIFYRTS, and (unless called confined) $UNCFRTS.

Result:

0

$CopyCatalogue (NewCata, OldCata)

Parameters:

Newcata -	Simple index, empty.
Oldcata -	Path index; Catalogue object reference, $ENVRTS, $UNCFRTS, $MODIFYRTS, CataCopyRts.

Effect:

NewCata is the capability for a newly created catalogue, containing all the entries of OldCata, returned with all auxiliary rights, $ENVRTS, $MODIFYRTS, and $UNCFRTS.

OldCata is a capability for the catalogue to be copied. Note that this operation circumvents the revocability of entries whose Environment boolean is false; CataCopyRts should therefore be guarded rather jealously.

Result:

0

$CataDestroy (Cata)

Parameters:

Cata	- Path index; Catalogue object reference, $MODIFYRTS, CataDestroyRts.

Effect:

Cata won't work any more.

Result:

0

$CataObjinfo (Val, Cata)

Parameters:

Val	- Simple index, empty.
Cata	- Path index; Catalogue object reference, CataListRts.

Effect:

Val is the capability for a newly created Page, which contains information about Cata. The first thirty-two words of the Page contain (1) an $Objinfo block of sixteen words describing the current 'version' of Cata and (2) a word containing the number of entries in Cata, say n. Following that are n blocks of thirty-two words each; the ith such block contains: (1) an $Objinfo block of sixteen words describing the value of the entry and (2) a block of eight words containing the Ascii name of the entry and (3) a word containing the protect boolean in the low-order bit. Val lacks $MODIFYRTS.

Result:

0

$CataLookup (Val, Cata, NamePar)

Parameters:

Val	- Simple index, empty.

Cata - Path index; Catalogue object reference, $ENVRTS, CataLookupRts.

NamePar - Path index; Data object reference, $GETDATARTS.

Effect:

Val is a copy of the value capability of the entry specified by Cata/NamePar. If the Environment boolean of any entry in the path is false, then Val will lack $ENVRTS.

Cata is a capability for a catalogue. Each step or pretarget on the path specified by Cata/NamePar must have $ENVRTS and CataLookupRts. The target entry must be defined.

Result:

0

$CataEnter (Cata, NamePar, Source, Options)

Parameters:

Cata - Path index; Catalogue object reference, $ENVRTS, $UNCFRTS, $MODIFYRTS.

NamePar - Path index; Data object reference, $GETDATARTS.

Source - Path index; object reference or template, $ENVRTS

Options - Path index; optional Data object reference, $GETDATARTS.

Effect:

Options is an optional parameter; if present, it must be a Data capability with $GETDATARTS. If its first word is present and non-negative, it becomes the Environment boolean (default is true). If its second word is present and non-negative, it becomes the Protect boolean (default is false).

Cata is a capability for a catalogue. Each step catalogue must have $ENVRTS, $UNCFRTS, and $MODIFYRTS and CataLookupRts. The pretarget must have $ENVRTS, $UNCFRTS, and $MODIFYRTS and CataEnterRts. If the target is not empty, then the pretarget must have CataDeleteRts (otherwise, CataErrDelete is signalled) and the entry must have its Protect boolean false (otherwise, CataErrProtect is signalled).

Source will become the value of the new target entry.

Result:

0

$CataRename (Cata, NamePar, NewPar)

Parameters:

Cata	- Path index; Catalogue object reference, $ENVRTS, $UNCFRTS, $MODIFYRTS.
NamePar	- Path index; Data object reference, $GETDATARTS.
NewPar	- Path index; Data object reference, $GETDATARTS.

Effect:

Cata is a capability for a catalogue. Each step catalogue must have $ENVRTS, $UNCFRTS, and $MODIFYRTS and CataLookupRts. The pretarget catalogue must have $ENVRTS, $UNCFRTS, and $MODIFYRTS and CataRenameRts. The target must be defined.

NewPar must be a valid NamePar with a single level name. An entry with this name must not already exist in the pretarget. The name of the target entry is changed to be that in NewPar.

Result:

0

$CataDelete (Cata, NamePar)

Parameters:

Cata	- Path index; Catalogue object reference, $ENVRTS, $UNCFRTS, $MODIFYRTS.
NamePar	- Path index; Catalogue object reference, $GETDATARTS.

Effect:

Cata is a capability for a catalogue. Each step catalogue must have $ENVRTS, $UNCFRTS, and $MODIFYRTS and CataLookupRts. The pretarget catalogue must have $ENVRTS, $UNCFRTS, and $MODIFYRTS and CataDeleteRts. The target must be defined and have a false Protect boolean (otherwise CataErrProtect is signalled). The target entry is removed from the pretarget.

Result:

0

$CataProtect (Cata, NamePar)

$CataUnprotect (Cata, NamePar)

Parameters:

Cata	- Path index; Catalogue object reference, $ENVRTS, $UNCFRTS, $MODIFYRTS.
NamePar	- Path index; Data object reference, $GETDATARTS.

Effect:

Cata is a capability for a catalogue. Each step catalogue must have $ENVRTS, $UNCFRTS, and $MODIFYRTS and CataLookupRts. The pretarget catalogue must have $ENVRTS, $UNCFRTS, and $MODIFYRTS and CataProtectRts. The target must be defined.

If $CataProtect is called, the Protect boolean of the target entry will become true; if $CataUnprotect is called, it becomes false.

Result:

0

$CataList (Cata, TTYPort)

Parameters:

TTYPort	- Path index; Port object reference, all auxiliary rights, $MODIFYRTS
Cata	- Path index; Catalogue object reference, CataListRts

Effect:

TTYPort is assumed to be connected to a teletype; things like &TTYPort are apt to work. To this port will be sent for each entry in Cata:

> the entry name

> a star if it's a template

> an 'at' sign if its protect boolean is true

> the print name of its Type

> its kernel and auxiliary rights

> the date and time of object creation

Finally, a count of the number of entries is given.

Result:

0

$DirToCata (NewCata, OldDir)

Parameters:

NewCata-	Simple index, empty.
OldDir -	Path index; Directory object reference, $ENVRTS,$UNCFRTS, and $MODIFYRTS.

Effect:

NewCata is the capability for a newly created catalogue, returned with all auxiliary rights, and with $ENVRTS, $UNCFRTS, and $MODIFYRTS. It is initialized with the contents of OldDir. Any entries of OldDir that had name characters that are illegal in the catalogue system will simply be left out.

Result:

0

19.2.1 Compatibility Procedures

There are procedures in the Catalogue Type object that accept calls in the calling sequence for DirGet, DirPut, DirDelete, and DirRename, munge with the Name Parameter in block notation, and do recursive calls in the calling sequence of $CataLookup, $CataEnter, $CataDelete, and $CataRename. Thus, if you have a catalogue and want to use it with a program that thinks it's a Directory and has the sense to use $Typecall's, then it will work.

Similarly, there are procedures in the Directory Type object that do the dual operation. Thus, if you still have a Directory and want to use it with a program that thinks it's a catalogue, then that will work too.

These two sets of procedures will not be supported indefinately.

20. The Command Interpreter

20.1. Command Language Subsystem

The Policy System 1 command language subsystem provides the interface between the system and a user at a terminal, in the form of an interactive language. The command language allows access to and control over the user's environment as provided by C.mmp, HYDRA and Policy system 1. This includes a mechanism for the invocation of Hydra procedures including all of the Policy System 1 procedures. It also has facilities to execute stored files of commands and to access the catalogue subsystem. Readers who have not used the command language recently should read the Command Language Introduction, CLINTR.DOC (or CLINTR.XGO) [N810HY97].

Users are hereby notified that this document represents a changing system, and therefore may be regarded with some suspicion. A list of known bugs can be found in CLBUGS.DOC[N810HY97], but we as yet have no way of generating a list of unknown bugs. With the co-operation of users and implementors we will be able to keep this document up-to-date, and put incremental changes or bug reports in the NEWS and SYSNEW files.

20.2. Command Language

The following two sections give an informal description of the command language. Section 20.8 contains the complete syntax description in the form of BNF rules. Some of the primitives, such as *identifier*, *numeric*, etc. are (or should be) so well known that their formal definitions are left as exercises for the reader. Informal descriptions are given below.

20.2.1 Lexical Structure

This section describes the results of the lexical analysis and macro expansion section of the command interpreter, describing what the "atoms" of the language are and what they mean. Atom refers to the smallest unit that the command interpreter understands.

The atoms are much the same as in any programming language: numeric quantities, variables, strings, and delimiters. In addition, there are the usual lexical niceties including comments, macro calls and line continuation. These constructs are similar to those of BLISS/11 and are defined as follows:

NUMERIC:	a sequence of characters that can be converted into a binary number. There are two kinds of numeric quantities: decimal and radix. Decimal numbers are the default and consist of a sequence of ASCII numeric characters terminated by a non-numeric character. Radix quantities are indicated by a sharp sign ('#') followed by a sequence of characters valid in the current radix. Radix numbers are terminated by any character that is not in the current radix set. The radix used to convert the input sequence is determined by a command language variable (see section 20.5) and may be set under program control. The full flexibility of radix control for input and output can be determined from studying section 20.5.2.
NAME:	a sequence of characters starting with an alphabetic and containing alphanumerics. Upper and lower case alphabetics are treated as the same characters, except in strings; thus an identifier with only upper case alphabetics is the same as one with the same alphabetics in all lower case or mixed upper and lower case alphabetics.
VARIABLE:	a name preceded by a '&' character.
STRING:	a sequence of characters contained within a pair of

quoting characters, either single quotes or double quotes (', ', or "). Strings may contain any ASCII character. The quoting character may be entered into the string by having two sequential occurrences of it. The double occurrence is converted into one character that is placed in the string. A question mark ("?") causes the character immediately following it to have its 7th bit cleared. Lower case letters also have their 6th bit cleared. This usually is used to convert upper case characters to the control characters. Thus "?A" is control-A, "?J" is control-J (line feed) and "?M" is control-M (carriage return). "??" is just "?". An exclamation mark ("!") causes the 6th bit of the following character to be set, which causes the upper case characters to be converted into their lower case equivalents.

COMMENT: a sequence of characters which are ignored by the command interpreter. A comment is enclosed either by a "!" and a line feed or by two occurrences of "%". The text of a comment is not examined except to find the occurrence of the terminating character. Thus an exclamation point may occur within a "%" type comment without affecting it. A comment must occur at a lexically valid point in the input string, that is, an occurrence of a commenting symbol will cause the termination of a lexical construct such as a name or a number.

DELIMITER: any character that isn't alphanumeric, %, !, blank or contained within quoting characters.

CONTINUATION: If an "↑" followed immediately by a carriage return (CR) or linefeed (LF) is received from the terminal during statement input, the "↑" and the line terminator (CR or LF) are ignored. Scanning continues with the next atom. The effect of the characters is to cause termination of an atom and to suppress the instance of the carriage return delimiter. This action does not apply inside strings or comments. Similar effects inside of stored programs may be obtained when &NOCRLF, a command language control variable, has the appropriate value. See section 20.5 for more details on &NOCRLF.

MACRO CALL: a name which has been "declared" as a macro, followed by its parameters (if any) enclosed in brackets (either "()", "[]" or "<>"). The macro name and parameters are replaced by the macro body on an atom by atom basis. All macros encountered during the scanning of the parameters are expanded as they are encountered. A macro without parameters does not have brackets associated with it. Section 20.3.3 on declarations gives a complete description of macro declarations.

20.3. Command Syntax and Semantics

This section describes the syntax and semantics of the command language. Examples are included which suggest the power and flexibility of the language.

20.3.1 Program

The command language is intended to be used interactively from a terminal. A user types a "program" to the command interpreter which is executed. A program consists of a set of statements in the language. In addition, programs may be stored away as "Commands " and executed as if they were procedures. See section 20.6 for a description of how to create Commands.

```
program ::= statement | program sep statement

sep ::= carriagereturnlinefeed | ;
```

When a line terminator is typed, the input line is analyzed and all complete statements are executed. Each statement is executed as soon as it has been parsed, and before any other statements are parsed. Multiple statements are allowed on a line, separated by semicolons. A statement entered from the terminal may be continued across a carriage return boundary by means of the continuation construct. Note that something of the form:

```
if Booleanexpression then
   DOTHIS()
  else
   DOTHAT()
```

will <u>not</u> work, since the carriage return is a separator; the program must be typed in as:

```
if Booleanexpression then ↑
   DOTHIS() ↑
  else ↑
   DOTHAT()
```

or typed in on a single line. To alleviate this problem, carriage returns are not treated as statement separators in stored programs. See also the discussion of &NOCRLF in section 20.5.

20.3.2 Statement

The *statement* is the smallest expression that is independently executed by the command interpreter.

```
statement ::= conditional | declaration | macrodeclaration | iteration
            | compound | i/o | string | simpleexpression | assignment
            | massignment | empty
```

Each statement is independently parsed and executed before any following statements are parsed. If an error occurs, either in parsing or in execution of a statement, all statements up to the statement causing the error have been executed and no statements after the erroneous ones will be executed. The erroneous statement may have been partially executed and thus some side effects may have occurred. If the sequence "S1; S2; S3" were typed to the command interpreter then S1 would be executed and on its successful completion S2 and then S3 would be executed. If "(S1; S2; S3)" were typed to the command interpreter then S1, S2 and S3 would all be parsed before S1 was executed. In the second case, if S1 were to invoke Commands which defined macros, the macros would not be defined for scanning S2 and S3.

20.3.3 Variables

Declarations allow access to and allocation of the resources of the command interpreter. Resources include core storage, which can be used in the form of *variables* or *macros* and LNS slots which can be used to store capabilities. The form of the name of a variable is defined by the lexical construct *variable*. The purpose of the & is to distinguish a variable from a catalogue name (which is described in section 20.3.10). The scope of a declaration is to the end of the current *block*. For a discussion of static and dynamic blocks, see section 20.3.5.

```
declaration ::= CAPA clnamelist | WORD clnamelist
              | WORDVEC clnamesizelist

clnamelist ::= clname | clnamelist , clname

clnamesizelist ::= clnamesize | clnamesizelist , clnamesize

clnamesize ::= clname specifier

clname ::= &identifier

specifier ::= . p3 | [ stringspecs ]

macrodeclaration ::= MACRO macrodefinition
```

```
macrodefinition ::= macroname = macrobody
          | macrodefinition , macroname = macrobody

macroname ::= macroid | macroid ( formalparms )

macroid ::= identifier | clname | controlcharacter

formalparms ::= identifier | formalparms , identifier

macrobody ::= stringwithoutdollarsign $
```

Variables introduced by the CAPA declaration hold capabilities, WORD variables hold a simple PDP-11 word (i.e 16 bits) and WORDVEC variable is either a linear vector of words or a byte string, being "specifier" words long . Capabilities and simple word variables are accessed by using the variable name (i.e. the construct "clname"). Words in a vector are accessed by specifying the desired word using the qualified name notation. Strings and substrings are specified by enclosing the offset and length of the string within brackets, where the offset is the number of bytes from the first byte to (and including) the byte desired and the length is number of bytes desired in the substring. It is possible to use strings from word vectors as names of entries in catalogues.

There is no claim for elegance made about this implementation of strings; it was expedient, and someday it may be improved so that real variables of type STRING will exist.

The macro definition facility allows any identifier, variable or control character (except carriage return and line feed) to be defined as a macro, giving the user some ability to tailor the system to his needs. Control characters are included so that common commands can be made into simple, single character macros which can be executed with a minimum of typing.

There are several predefined variables which the command language uses to interface to the user. These are listed in section 20.5.

Examples

```
CAPA &A,&B                          % declare two capability variables%
WORD &C                             %declare a simple word variable%
WORDVEC &STR.36, &STR2[30]          % declare a vector of 36 words and
                                    a string of 30 characters %
&STR2[5,6]                          % refers to the string composed of
                                    the fifth thru tenth bytes %
&STR.2                              % refers to the second word of the
                                    vector %
MACRO MAK(A,B)=A=$CREATEPMPROS(B); $STARTPMPROS(A)$
                                    % macro "MAK" creates and starts
                                    a process %
MACRO CRLF=TYPE '?M?J'$             % type carriage return line feed on
                                    the terminal %
MAK(&PROS,&PROC); CRLF; CRLF
```

```
                                        % create process and start it, then
                                        type two carriage return-linefeeds
                                        %
```

20.3.4 Simple Expression

```
simplexpression ::= p10 | simplexpression XOR p10 | simplexpression EQV p10

p10 ::= p9 | p10 OR p9

p9 ::= p8 | p9 AND p8

p8 ::= p7 | NOT p8

p7 ::= p6 | p6 rel p6

rel ::= EQL | NEQ | LSS | LEQ | GTR | GEQ | EQLU | NEQU | LSSU
          | LEQU | GTRU | GEQU

p6 ::= p5 | p6 + p5 | p6 - p5

p5 ::= p4 | p5 * p4 | p5 / p4 | p5 MOD p4

p4 ::= p3 | p4 ↑ p3

p3 ::= unsigned | + p3 | - p3

unsigned ::= p2 | numeric | compound | block | string | invocation | ove
          | statement

p2 ::= clname | clname specifier

specifier ::= . p3 | [ stringspecs ]

stringspecs ::= firstchar | firstchar , | , stringsize | firstchar , stringsize

firstchar ::= simplexpression

stringsize ::= simplexpression
```

A simplexpression is an arithmetic expression. All operations are upon 16 bit binary quantities and result in a 16 bit quantity. The precedence of the operations is implicit in the BNF above. A variable in a simple expression must evaluate to a simple numeric quantity.

Example

```
        &A+24                         ! WORD variable &A plus 24
```

&C.&A MOD 12	% &Ath word in WORDVEC &C mod 12%
&A GTR &C.5	% 1 if &A contains a larger value than the 5th word of &C, 0 otherwise %

When specifying a part of a substring, the specifiers refer to bytes (not words) in a WORDVEC variable. The general form of the specifier is [firstchar , stringsize]. For convenience one of the arguments may be omitted. The defaults are:

Written	Meaning
[first]	[first,1]
[first,]	[first,rest.of.string]
[,length]	[1,length]

where rest.of.string will use the words remaining between the given byte position and the end of the WORDVEC holding the string.

20.3.5 Block and Compound

```
block ::= BEGIN program END

compound ::= ( program )
```

A *block* or *compound* is an arbitrary number of statements separated by semicolons or carriage returns and bracketed by BEGIN - END or (-), respectively. The *compound* is used syntactically to allow the grouping of statements. A compound statement does not determine the scope of variable declarations.

A declaration context level is implicitly defined by a matching BEGIN END pair. A context may also explicitly be entered and exited by the command language functions BLKBEGIN and BLKEND. These functions provide for a dynamic block structure which can nest within the static block structure. The main purpose of BLKBEGIN and BLKEND is to define a local context in which future commands typed in at the terminal will be executed. The BLKEND will undeclare any variables declared in the "inner block"; these have usually been declared by direct typein by the user. Note that for each BLKBEGIN function executed a matching BLKEND function must be executed.

20.3.6 Assignment

The command language allows extensive access to capabilities and numeric quantities, including the catalogue structure. This includes the ability to manipulate capabilities in a manner similar to that of simple numeric quantities. The *ovel* construct below is defined later but indicates an access to the catalogue subsystem or to objects.

```
assignment ::= name ← statement

name ::= ovel | p2
```

The evaluation of an assignment involves evaluation of the right hand side and then, if possible, the storing of the result into the left hand side. Type checking and some type conversion are performed.

Examples

```
&A←21                          ! give &A the value 21

&C.12← &A/5 * &C.3
```

20.3.7 Iteration

Since commands may be executed from sources other than the terminal, the command language has expressions which allow control over program execution.

```
iteration ::= WHILE simplexpression DO statement
```

The value of the simple expression is computed and if its value is "true", the statement is executed and the iteration statement is reevaluated. If the simple expression is "false", the iteration evaluation is complete. The value is considered "true" if its low-order bit is 1, and "false" if its low-order bit is 0. This is consistent with the BLISS interpretation of Booleans. The canonical values of relations are "1" for true and "0" for false. The Boolean operators AND, OR, NOT, etc. are bitwise operations on unsigned 16-bit values. Users of languages in which any nonzero value is considered "true" are urged to keep this definition in mind. The value of the iteration is that of the last evaluation of the statement or -1 if the statement is never evaluated.

Example

```
WORD &A; WORDVEC &C.100

&A←0

While (&A←&A+1) leq 100 do &C.&A←0
```

20.3.8 Conditional

```
conditional ::= IF simplexpression THEN statement elsepart

elsepart ::= ELSE statement | empty
```

The simple expression is evaluated and if it has a value of "true", the statement following the THEN is executed, otherwise the *elsepart* (if any) is executed. The definitions of "true" and "false" are given in section 20.3.7. Since the THEN and the ELSE parts are statements, it is possible to have conditional declarations. Since macros are defined during input scanning, conditional definition of macros by this method is not possible. Commands defining macros may be in a conditional statement. The macros will not be defined during the current input scan. When using conditional declarations, one must be careful to avoid errors. This facility is intended to be used from Commands. For more information about Commands see section 20.6.

Conditionals may be nested to any depth desired, as the BNF indicates. It is not necessary to include an ELSE for each THEN. The "dangling else" ambiguity is resolved by matching each ELSE with the most recent unmatched THEN.

20.3.9 I/O

```
i/o ::= TYPE outputlist | ACCEPT() | INTERPRET()

outputlist ::= output | outputlist , output

output ::= string | simplexpression
```

The three types of I/O statements allow a CI program to communicate with the user at the terminal. The TYPE statement prints on the terminal and allows as parameters anything that evaluates to either a string or a numeric quantity. A string is printed as it appears (including any control characters) and a numeric quantity is printed as a signed decimal. Note that unlike some systems, the TYPE statement does not output a free <crlf>, which gives the user greater flexibility than in those systems which provide such a "service". A note of caution is in order, however. If a user forgets to put a <crlf> at the end of a line, the usual case is that the value of the statement is printed out upon return to the top level. In most cases this is a zero, and since it is concatenated to the current line the value printed appears to be too large. Consider the example:

```
0>TYPE "HI"
HI0>
```

which is fairly obvious; the "0>" is the result of the last statement plus the ">" prompt character. However, a common (and easily made) error occurs in the case of:

```
0>&RETVAL←123
123>TYPE &RETVAL
1230>
```

where the actual value was "123" followed by the "0" for the result of the statement. Be careful! Note: The printing of the result can be suppressed by use of the &PRINTVALUE variable; for details see section 20.5.3.

The ACCEPT function gets input from the terminal up to and including the next carriage return, line feed pair. The input is converted into a string. This string is the value of the statement. ACCEPT may take one optional argument. If the argument evaluates to 0, then the break character (typically carriage return-linefeed) is included in the string. If the argument evaluates to 1, or is omitted, the break character is removed.

The INTERPRET function allows the user to enter a single CI statement from the terminal. This statement is immediately parsed (including macro processing) and executed in the context of the INTERPRET function before control is returned to the command containing the INTERPRET command. The INTERPRET function in effect performs a recursive call upon the command interpreter. The value of the INTERPRET function is that of the statement it evaluates.

20.3.10 OVE - Object Valued Expressions (Capabilities)

The major power and utility of the command interpreter is its ability to deal with Hydra style objects. The command language contains commands for accesses into or out of the catalogue structure or other Hydra objects. The catalogue procedures not implemented syntactically are easily available by name. For information on the current system catalogues, see SYSDIR.DOC[N810HY97].

The following sections describe the valid syntactic forms. Dynamic type and rights checking are employed to validate the semantics of each execution instance.

```
ove ::= ove1 | ove1  rtsrestrict  | invocation | constructor

ove1 ::= ove1 . p3 | dirid | ove1 . dirid | capavariable

dirid ::= identifier | string

rtsrestrict ::= [ rtspecs ]

rtspecs ::= auxrts | auxrts , | ,kernelrts | auxrts , kernelrts

kernelrts ::= simplexpression

auxrts ::= simplexpression
```

An Object Valued Expression (*ove*) is a command language statement that has a Hydra object as its result. This object can be assigned to other objects (or capability variables) or can itself be assigned into.

The simplest form of an *ove* is an object or catalogue "path", which consists of a qualified name path, e.g. 'A.B.C' or '&A.1.2.3'. Names can be used only for lookups in catalogues and numbers only for object paths. To lookup numbers in catalogues, put them in quotes.

Note that each qualifier may be an expression. Names represent themselves, thus the path A.B.C is identical to the path @'A'.'B'.'C'. The @ is necessary to inform the command interpreter that 'A' is a catalogue name, not a string. If the variable &A holds the capability for a catalogue, then the path &A.B.C uses the catalogue selected by &A as the root of the catalogue path. If &A is a WORDVEC, and contains the string 'GORP' in bytes 1 through 4, then the path &A[1,4].B.C is equivalent to the path 'GORP'.B.C or GORP.B.C. The use of self-quoting names is valuable, but leads to problems when "reserved" command interpreter symbols are used; if a user wishes to have a qualifier called EVAL (which is a reserved command interpreter function) the path cannot be specified as MUMBLE.EVAL.GORP since the command interpreter will attempt to evaluate EVAL and issue some error about insufficient parameters; the path must be specified as MUMBLE.'EVAL'.GORP.

The actual value of an *ovel* may be either a capability or a one word numeric quantity. The evaluation of an *ovel* up to the last qualifier must result in a capability. However the last qualifier, if numeric, specifies either a capability, with a positive value, or a word from the data part, with a negative value. 0 is always invalid.

Example

Note that each level of qualification can be an expression, ϵ. If we designate the last expression as ϵ_k, then the expression X.Y.1.ϵ_k has the evaluation:

$\epsilon_k > 0$ - refers to the object referenced by the ϵ_kth capability in the object reached by the path 'X.Y.1'.

$\epsilon_k < 0$ - refers to the $-(\epsilon_k)$th word of the data part of the object reached by the path 'X.Y.1'.

$\epsilon_k = 0$ - generates an error.

All *ovel's* start from either an capability variable or from the default catalogue reference. The default catalogue may be set by the user and initially is the user's top level catalogue, as returned by $LOGIN. It is stored in the predefined CL variable &USERDIRECTORY.

The *rtsrestrict* construct permits the restriction of the "rights" of a capability. The rights of a capability indicate which kernel functions and type-dependent operations may be performed upon that capability and the object it represents. For a detailed explanation of rights see the Hydra Reference Manual. Unfortunately at the moment, August 15, 1977, the symbols which define the rights are not available within the command interpreter. Users who wish to use the rights-restriction construct must first determine the absolute octal values necessary. Pat McGehearty has thoughtfully provided a command object called RIGHTS, which when executed defines the values of the Kernel rights. It currently resides at &SYSDIRECTORY.PUBLIC.PM10.RIGHTS. To determine the values of the rights fields for Kernel objects, see section RTS.REQ[N811HY97].

20.3.11 Invocations

One of the most important functions in the Hydra environment is the Hydra Procedure Call. The Hydra Call allows the access rights of an object to be changed so that secure operations may be performed upon the objects. The CI allows a Call by means of an invocation - which has the syntactic form of an Algol procedure call and is similar to the procedure call from a BLISS program.

```
invocation ::= ovel ( actualparmlist ) | procname ( actualparmlist )

actualparmlist ::= empty | actualparm | actualparmlist , actualparm

actualparm ::= block | compound | ovel | simplexpression

procname ::= cifunctionname | kernelfunctionname | subsystemprocedurename
```

The *cifunctionname* names one of the special command interpreter functions defined in section 20.4.

The *ovel* must evaluate to a Hydra procedure object or a command object. The parameters to the invocation are evaluated to their simplest form, capabilities, numeric quantities or strings. The parameters for a Hydra procedure call are all made into capabilities, i.e. a numeric or a string quantity is made into a data object in order to be passed. Several numeric or string quantities may be concatenated into one data object by using the special command interpreter function $STACKDATA which transforms its arguments into one data object. $STACKDATA is a relatively new function. It is intended to replace CALLDATA (See next paragraph). $STACKDATA does not reverse the order of its arguments. Currently, $STACKDATA only will work as a parameter to an invocation. At some future time, this restriction may be removed.

CALLDATA has a non-obvious perversity which makes its use complex insofar as the subsystem designer is concerned. CALLDATA <u>reverses the order of its arguments</u>. Thus CALLDATA (1,2,3) creates a data object whose first word is 3, whose second word is 2, and whose third word is 1. Warning to users: if a document says to pass arguments as CALLDATA(1,2,3) then do so! Don't think that you have to reverse them. The poor guy who wrote the subsystem already had to worry about that, and he expects a data object with arguments in the order (3,2,1). However, if you plan to call a procedure from a BLISS program, you have to reverse them from the CALLDATA convention.

If the invocation object is of type Commands, a recursive call upon the command interpreter is performed and the object is used as the input source for one or more statements. Those parameters which are not capabilities are converted into data objects. If a string with an odd number of characters is passed as a parameter, a zero byte is added to fill out the word. Two variables of type CAPA are declared: &CDOPARM and &PARMS. Their scope is the command object statement. If the invoking object is of type Commands then &CDOPARM contains a capability for an object with the C-List of the Commands. &PARMS is a universal object which holds the parameters in its C-List. To get data from the data objects in &PARMS, see the section on Multiple Assignments, 20.3.12.

An invocation can also call command language or Hydra kernel functions. The command language functions are defined in section 20.4 and the kernel functions are defined in the Hydra Reference Manual. Parameters to these invocations are evaluated into strings, numeric quantities or capabilities and passed <u>without further transformation</u>. A special feature of wordvecs allows addresses to be passed to kernel calls. If a variable of type wordvec without any specifier is used in a kernel call, then the wordvec is copied in its entirety to the stack page and the address of the wordvec in the stack page is used in the kernel call. After the kernel call, the wordvec is copied back to its normal location.

Examples

```
Wordvec &A.5;
$GETDATA(&A,TEST,3,5)
```

The above example will get the 3rd thru 7th word of the data part of TEST and put it into &A. For data parts larger than 2 words, the method is more efficient than the TEST.-n construct. A maximum of 512 words are available for this purpose at any given time. Caution: since the CL currently does not check parameters for validity, the unsuspecting user can easily hang his/her CL by an incorrect kernel call. The result of an invocation may be a numeric quantity, a capability, multiple capabilities, or a numeric quantity and multiple capabilities. Kernel and CI functions have a return value which is function-dependent, usually just a numeric quantity. A procedure call returns a numeric quantity and (possibly) several capabilities. This is treated similarly to an object (see Multiple Assignments, 20.3.12) except that a reference can not be assigned to an capability variable. A command object has an explicit numeric value and may return values in global variables as well.

20.3.12 Multiple assignments

In the Hydra environment it is useful to extend the notion of assignment of capabilities from the one to one mapping normally thought of as assignment to the assignment of the constituent parts of an object. This type of assignment would in one operation, assign capabilities from the capability list and data from the data part to variables and the catalogue structure.

```
massignment ::= namelist = ove

namelist ::= name | namelist , name
```

The semantic intent of this assignment is to assign, in left to right order, the components of the *ove* to the named quantities on the left of the '='. Successive words or blocks of words in the data part are assigned to the WORD or WORDVEC variables and successive capabilities from the capability list of the *ove* are assigned to the capability variables or *ove1*'s. If the assignments require more capabilities or data than is available, null words (0) are used for data and null capabilities are used for the excessive capabilities.

In practice, most users are horribly confused about the difference between = and ←. A good rule of thumb is that one should USE "=" WHEN ASSIGNING THE RESULT OF A INVOCATION and USE "←" WHEN ASSIGNING THE RESULT OF A DIRECTORY LOOKUP OR SIMPLE VALUE ASSIGNMENT.

If the *ove* was a simple object then a multiple assignment could be simulated by a series of simple assignments. However if the *ove* were an invocation returning several capabilities then the operations could not be simulated by such a sequence.

Examples

```
CAPA &A,&B;
WORD &W1, &W2;
WORDVEC &V.3
&A,&W1,&V,X.Y,&B,&W2=GORP
                                  % this is the same as the simple
                                  assignments %
&A←GORP.1
&W1←GORP.-1
&V.1←GORP.-2
&V.2←GORP.-3
&V.3←GORP.-4
X.Y←GORP.2
&B←GORP.3
&W2←GORP.-5
```

20.4. Command Language Functions

The Command Language provides several functions and variables which serve a variety of purposes.

CALLDATA(p_1, ... p_n)

Evaluates each parameter p_i, which must evaluate to a simple 16-bit value or a string. These are then used to create a data object containing the parameters in the order $p_n,...,p_1$. Please note this reversal if you are writing a procedure which expects a data object. A string of m characters will occupy (m+1)/2 words in the data object. If m is odd, the empty byte will be a zero.

$STACKDATA(p, ..., p_n) 20.3.12

Same as CALLDATA, except it does not reverse its parameters.

UNDECLARE(p_1, ..., p_n)

Removes the most recent declaration of the specified parameters from the symbol table and returns their storage to the free space list. Its return value is the number of declarations successfully removed.

INTERPRET()

Gets a statement from &TTYPORT and executes it. The value is whatever value is returned by the statement evaluated. Normally, this function is used inside of command objects.

ACCEPT()

Accepts a line of input from &TTYPORT and returns it as a string. If no parameter is specified, or its parameter is 1, then the break character is removed from the string. If the parameter is 0, the break character is included in the string.

EVAL(str)

Evaluates the string-valued parameter as a command interpreter statement.

DECLARATION(str)

If the string-valued parameter is the name of a declared variable, then the value of this function is an integer greater than 0. If the name has not yet been declared, the value is 0. The values are not given here because they might change; if you wish to check the type of a declaration, declare a dummy variable of a known type, and see if DECLARATION of your dummy variable is the same as DECLARATION of the variable you are concerned with.

$VERSION()

Returns the current version as a string.

ERRMESS(val)

If the value of the parameter is an integer which is a valid error message code, the text of the error message is printed. This is useful if full error message printout was suppressed by use of &LISTERROR, see section 20.5.3.

HUH()

Print the last error message in extended form. This also is useful if full error message printout was suppressed by use of &LISTERROR, see section 20.5.3.

BLKBEGIN()

Declare a dynamic context within the current static context; see section 20.3.5.

BLKEND()

Leave a dynamic context declared by BLKBEGIN; see section 20.3.5.

PRCAPA(ove)
Prints appropriate information about the capability obtained from evaluating the ove, as determined from the Kernel WHAT call (see the Hydra Reference Manual).

PRSYM(clname)
Prints some internal information about the variable given as a parameter (no quoting is done, so macros are expanded--- you can't PRSYM a macro name directly). To PRSYM a macro name, use a quoted string with the name in upper case letters. Mostly of value to command interpreter debuggers, but users may find it handy.

GETMEM(address)
If the expression given as the parameter evaluates to an integer in the range (decimal) 64 to 128, the value of GETMEM is the contents of that location in the stack page. If the address is odd, an error message will be given.

PUTMEM(address,value)
If the expression given as the first parameter evaluates to an integer in the range (decimal) 64 to 128, then that location in the stack page is set to the value obtained from evaluating the second parameter. If the address is odd, an error message will be issued.

DELETE(ove)
If ove is a local capability, it will be deleted and the capa released. If it is a path, the last element of the path will be deleted. Note that the kernel function $DELETE(ove) does not have the expected effect. In this case, the ove is evaluted by the command language and the resulting local capa is deleted, effectively a no-op.

TYPECALL(object,parms)
Will do a $TYPECALL according to the type of object, passing parms as parameters.

BASECALL(index,parms)
Will do a $BASECALL of the specified index, passing parms as parameters.

20.5. Predeclared variables

20.5.1 Directories

&SYSDIRECTORY
Variable which contains a capability for the system catalogue.

&USERDIRECTORY
Variable which contains a capability for the user's catalogue. If a catalogue path name does not start with a capability for a catalogue, &USERDIRECTORY is used, e.g., the path 'A.B' is identical to the path '&USERDIRECTORY.A.B'.

20.5.2 I/O control

&TTYPORT
Variable which contains a capability for the initial port which is connected to the terminal.

&TTYCON
Variable which contains a capability for the connection to &TTYPORT.

&RADIX
The radix used for converting numbers <u>without</u> a # symbol on input (initially ten).

&RADIXIN
Current input radix for # type numbers (initially eight).

&RADIXOUT
Current output radix (initially ten). This radix is used for all output conversions of numbers.

20.5.3 Interaction control

&LISTERROR
If set to "true", full error messages will print. If set to "false", only the error message code itself will print. See the ERRMESS function in section 20.4. Its initial value is "true".

&PSIGNAL

If set to "true", a Hydra signal will print a meaningful description as well as the error number. If set to "false", only the number will be printed. Its initial value is "true".

&PRINTVALUE

If set to "true", the value of the last statement executed will print out before the user is prompted for the next input line. If "false" this typeout will be suppressed, and the value must be explicitly printed by the user if required. Its initial value is "true".

&PRTSTACK

If "true", various state information about the interpreter stack is printed when an error occurs. If "false" the printout is suppressed. This is primarily for use by people debugging the interpreter, so its initial value is "false".

&NOCRLF

Controls the effects of carriage return linefeed (CRLF) and ↑ inside of stored programs. If &NOCRLF is 1 then CRLF is not a statement terminator inside of stored programs. Also, ↑ is not a continuation symbol. This feature allows reasonable indentation and spacing inside of stored programs. If &NOCRLF is 2 then CRLF is not a statement terminator inside of stored programs but ↑ acts as a continuation symbol. This feature was provided to allow a transition between the old way and the new way. If &NOCRLF is 0 then CRLF acts as a statement terminator and ↑ acts as a continuation symbol. This is the old way. The default for &NOCRLF is 1. System commands assume that &NOCRLF is 1.

&REDECLARE

Controls the action taken when a new variable is redefined at the same block level as an existing variable with the same name. If it is 0 then delete the old definition at this level, print warning. If it is 1 then delete the old definition and continue (no warning). If it is 2 then stack the old definiton, and continue. If it is 3 ask &TTYPORT whether 1 or 2 is appropriate. The default is 0.

20.5.4 Miscellany

&RETVAL

Value of the last invocation of a procedure or kernel function, or catalogue lookup.

&RETCAPA

# of capabilities returned in the last call.

&CDOPARM
: When a stored program is invoked, the variable &CDOPARM contains a capability for the stored program.

&PARMS
: When a stored program is invoked, the variable &PARMS is a universal object containing the actual parameters of the invocation. See section 20.3.11.

&COPYCPS
: Contains a safe CPS slot for page copying, etc.

20.6. Commands

Commands are a protected type in Hydra maintained by the command language. They provide a means for users to create and save stored programs in the command language. There exist predefined type calls to create and modify Commands.

$Makecmd() will create and return a new Commands object.

$Copycmd(Cmd) will create a copy of Cmd and return it.

$Editcmd(Cmd) will edit a Cmd with the C.mmp teco editor.

$Listcmd(Cmd) will list Cmd on the line printer.

$Printcmd(Cmd) will print a Cmd on the terminal.

$Readtext(cmd) will return a universal with one page in its clist which contains the text of the cmd.

$Writetext(cmd,unv) writes the text of the cmd from the page in the first slot of the clist of unv.

$Readclist(cmd) returns a universal containing the capabilities that would be available in an invocation of Cmd.

$Writeclist(Cmd,Unv) makes the capabilities in Unv the ones that are available in an invocation of Cmd (as &CDOPARM).

See also the earlier section on invocations for more information about Commands.

20.7. Known Bugs

Since the command interpreter forms the users' prime interface to Hydra, it has been worked over pretty thoroughly and most bugs have been removed. However, some creep in with new releases and others go away. (Fortunately it is nowhere near "critical mass", the phenomenon where removing one bug introduces $1+\epsilon$ bugs). Since bugs change faster than this document, the list of known bugs will be kept in CLBUGS.DOC[N810HY97].

20.8. BNF summary

Section	Definition
20.3.11	actualparm ::= block | compound | ovel | simplexpression
20.3.11	actualparmlist ::= empty | actualparm | actualparmlist , actualparm
20.3.6	assignment ::= name ← statement
20.3.10	auxrts ::= simplexpression
20.3.5	block ::= BEGIN program END
20.3.3	clname ::= &identifier
20.3.3	clnamelist ::= clname | clnamelist , clname
20.3.3	clnamesize ::= clname specifier
20.3.3	clnamesizelist ::= clnamesize | clnamesizelist , clnamesize
20.3.5	compound ::= (program)
20.3.8	conditional ::= IF simplexpression THEN statement elsepart
20.3.3	declaration ::= CAPA clnamelist | WORD clnamelist | WORDVEC clnamesizelist
20.3.10	dirid ::= identifier | string
20.3.8	elsepart ::= ELSE statement | empty
20.3.4	firstchar ::= simplexpression
20.3.3	formalparms ::= identifier | formalparms , identifier
20.3.9	i/o ::= TYPE outputlist | ACCEPT() | INTERPRET()
20.3.11	invocation ::= ovel (actualparmlist) | procname (actualparmlist)
20.3.7	iteration ::= WHILE simplexpression DO statement
20.3.10	kernelrts ::= simplexpression
20.3.3	macrobody ::= stringwithoutdollarsign $
20.3.3	macrodeclaration ::= MACRO macrodefinition

```
20.3.3        macrodefinition ::= macroname = macrobody
                    | macrodefinition , macroname = macrobody

20.3.3        macroid ::= identifier | clname | controlcharacter

20.3.3        macroname ::= macroid | macroid ( formalparms )

20.3.12  massignment ::= namelist = ove

20.3.6        name ::= ovel | p2

20.3.12  namelist ::= name | namelist , name

20.3.9        output ::= string | simplexpression

20.3.9        outputlist ::= output | outputlist , output

20.3.10  ove ::= ovel | ovel  rtsrestrict  | invocation | constructor

20.3.10  ovel ::= ovel . p3 | dirid | ovel . dirid | capavariable

20.3.4        p10 ::= p9 | p10 OR p9

20.3.4        p2 ::= clname | clname specifier

20.3.4        p3 ::= unsigned | + p3 | - p3

20.3.4        p4 ::= p3 | p4 ↑ p3

20.3.4        p5 ::= p4 | p5 * p4 | p5 / p4 | p5 MOD p4

20.3.4        p6 ::= p5 | p6 + p5 | p6 - p5

20.3.4        p7 ::= p6 | p6 rel p6

20.3.4        p8 ::= p7 | NOT p8

20.3.4        p9 ::= p8 | p9 AND p8

20.3.11  procname ::= clfunctionname | kernelfunctionname | subsystemprocedurename

20.3.1        program ::= statement | program sep statement

20.3.4        rel ::= EQL | NEQ | LSS | LEQ | GTR | GEQ | EQLU | NEQU
                    | LSSU | LEQU | GTRU | GEQU

20.3.10  rtspecs ::= auxrts | auxrts , | ,kernelrts | auxrts , kernelrts

20.3.10  rtsrestrict ::= [ rtspecs ]

20.3.1        sep ::= carriagereturnlinefeed | ;
```

```
20.3.4      simplexpression ::= p10 | simplexpression XOR p10
                  | simplexpression EQV p10

20.3.3      specifier ::= . p3 | [ stringspecs ]

20.3.4      specifier ::= . p3 | [ stringspecs ]

20.3.2      statement ::= conditional | declaration | macrodeclaration |  iteration
                  | compound | i/o |  string |  simpleexpression | assignment
                  | massignment | empty

20.3.4      stringsize ::= simplexpression

20.3.4      stringspecs ::= firstchar | firstchar , | , stringsize |
                  firstchar , stringsize

20.3.4      unsigned ::= p2 | numeric | compound | block | string | invocation | ove
                  | statement
```

Appendix I: Typecall summary

This appendix summarizes the Typecalls and their parameters and provides a quick reference both to the calls and to their descriptions in the manual.

$CataDelete (Cata, NamePar)	19-6
$CataDestroy (Cata)	19-4
$CataEnter (Cata, NamePar, Source, Options)	19-5
$CataList (Cata, TTYPort)	19-7
$CataLookup (Val, Cata, NamePar)	19-4
$CataObjinfo (Val, Cata)	19-4
$CataProtect (Cata, NamePar)	19-7
$CataRename (Cata, NamePar, NewPar)	19-6
$CataUnprotect (Cata, NamePar)	19-7
$CloseFile (NewFile, OpenFile, Abort)	21-8
$CompareFile (F1, F2)	21-11
$Controlpmprocess (Pros, Data)	23-3
$CopyCatalogue (NewCata, OldCata)	19-3
$CopyFile (NewFile, OldFile, OPrName, Job)	21-9
$DASCONNECT (DIndex, JOB, PORT, IODevice, DATA)	24-3
$DASDisconnect (DIndex, DEVCON)	24-8
$DASGetDC (DIndex, IODevice)	24-5
$DASIODDisplay (DIndex, IODevice)	24-4
$DASIODStatus (DIndex, IODevice)	24-5
$DASMerge (DIndex, IODevice1, IODevice2)	24-5
$DASPConnect (DIndex, JOB, PORT, DEVCON, DATA)	24-7
$DASSConnect (DIndex, JOB, PORT, DEVCON, DATA)	24-7
$DASSelect (DIndex, IODevice, DATA)	24-6
$DASXCONNECT (DIndex, JOB, PORT, IODevice, DATA)	24-4
$Datapmprocess (Pros)	23-5
$Desynchpmprocess (Pros, Data)	23-3
$DirToCata (NewCata, OldDir)	19-8
$DISPLAY (DEVCON)	24-8
$EditFile (File, Port, Job, Options)	21-11
$Getpm (DIndex, Job, Data)	22-2
$Joberror (Job, Data)	22-4
$Jobid (Job)	22-1
$Jobident (DIndex, Job)	22-1
$Jobinsert (Job, SPath)	22-5
$Jobname (DIndex, Job)	22-2
$Jobremove (Job, SPath, Data)	22-6
$Jobseterrport (Job, Port)	22-4
$Jobsjcreate (DIndex, Job,Data)	22-7

$Join (DIndex, Job, Port)	22-3
$Killpmprocess (Pros)	23-2
$Login (DIndex, Job, Port)	22-3
$Logout (Port, Data, Job)	22-6
$MakeCatalogue (NewCata, CataRep)	19-3
$MakeFile (NewFile, OldFile, Job, OPrName)	21-3
$Makepmprocess (DIndex, PMtempl, Pros, Job, arguments)	23-1
$Makeport (Job, Data)	22-5
$ObjinfoFile (NewData, File, Field)	21-10
$OpenFilexxx (NewOpen, Super, Port, Channel, OPrName, Job)	21-4
$Startpmprocess (Pros)	23-2
$STATUS (DEVCON)	24-9
$Statuspmprocess (Pros)	23-3
$Stoppmprocess (Pros)	23-2
$Timepmprocess (Pros)	23-4

INDEX