SL230 - A SOFTWARE LABORATORY

INTERMEDIATE REPORT

W. Corwin W. Wulf

May 1972

Carnegie-Mellon University

Pittsburgh, Pennsylvania

This work was supported by the Advanced Research Projects Agency of the Office of the Secretary of Defense (F44620-70-C-0107) and is monitored by the Air Force Office of Scientific Research.

ABSTRACT

This report describes the resources and data structures of SL230 (Software Laboratory 230) and the designing of SL230 modules and systems. SL230 is a simple, multiprocess, operating system used to create an environment suitable for the construction of experimental programming systems for educational and research uses.

INTRODUCTION

The similarity between many of the components of various systems programs has often been noted but seldom exploited. Lexical analyzers and syntax analyzers, for example, occur in all compilers and to some extent in assemblers, editors, command interpreters, etc. Yet they are generally re-written for each such system (translator-writing systems, or compiler-compilers, have been the one exception to this practice). This situation is especially annoying to two groups of people to whom the present report is primarily aimed: (1) the researcher who would like to quickly fabricate a system in order to experiment with a single aspect of it in depth and (2) the instructor who would like to assign programming problems on some aspect of systems programming but which only make sense in the context of a complete system. To illustrate this point, consider the researcher (or student) who would like to (is assigned to) investigate various compiler optimization strategies on the tree-representation of a program. To do this, lexical analysis, symbol table and space management, parser, tree-generation, and I/O functions must first be written. None of these is essential to the project at hand, and collectively they may be sufficiently effort-consuming to make the project impractical.

This report describes the intermediate results of a project to design a software laboratory (SL230) suitable for the study of software systems.

The Physical Model

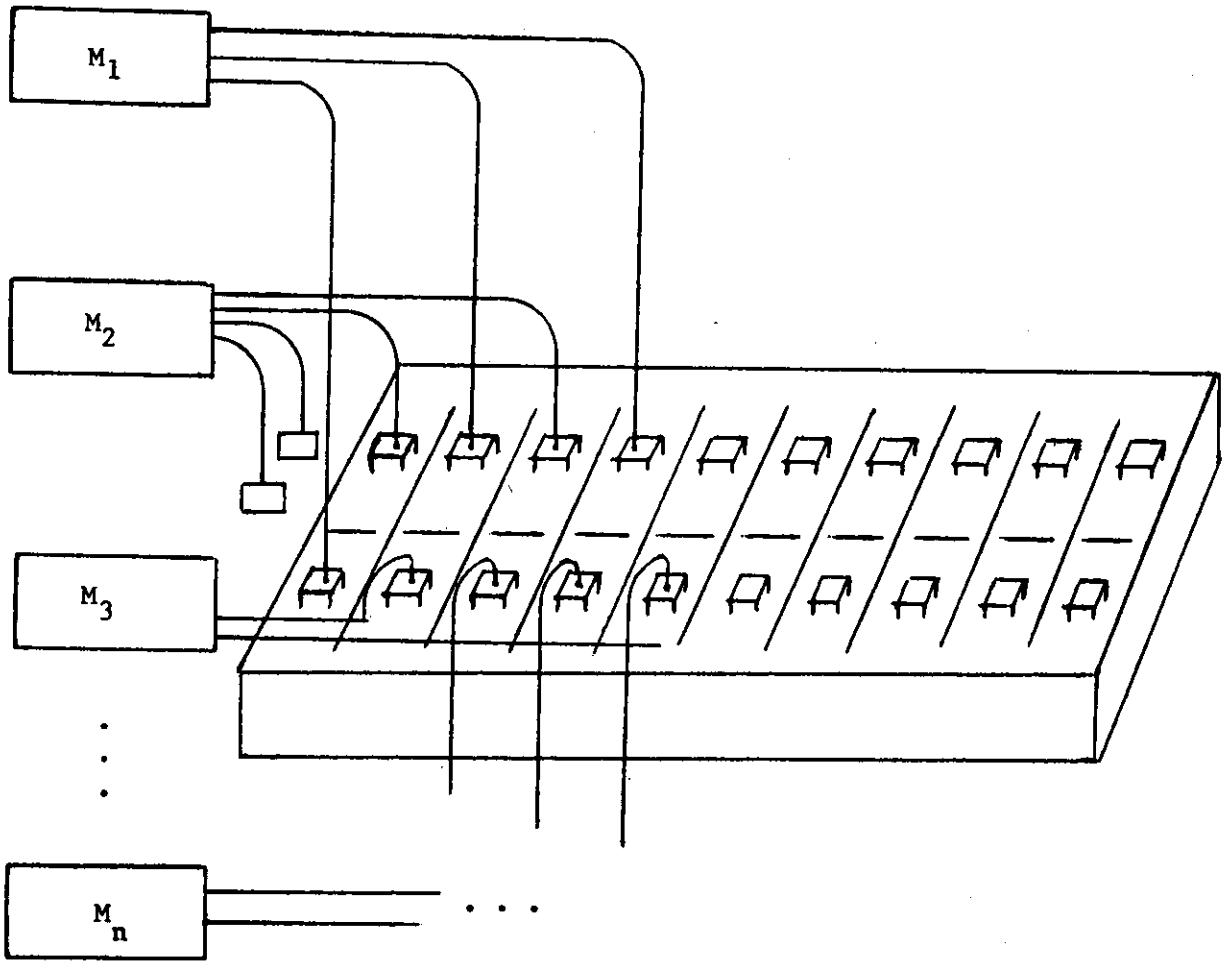


FIGURE I

THE PHILOSOPHY

The objective of SL230 is to create an environment within which researchers and students may experiment with the construction of software systems. The system accomplishes this by providing a large number of functional "modules" together with a mechanism for flexibly interconnecting them in various ways. The philosophy of the system is a software analog of the hardware "macro-modules" of Clark [1] and "register-transfer-modules" of Bell [2]. Much of the philosophy for the approach described below is due to Krutar [3]; key ideas were borrowed from Habermann and Jones [4] and from many discussions with Per Brinch Hansen.

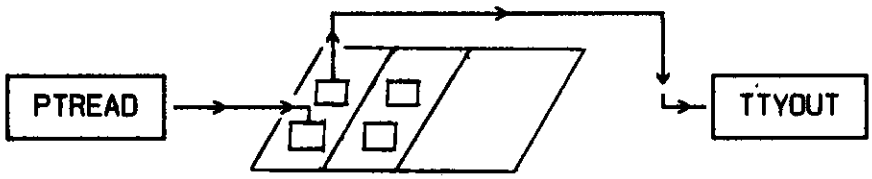
The philosophy of the SL230 environment results from consequences of a particular physical model. The concepts implied by that model are essential for the user to understand that environment. That model is:

> A (user) system is constructed from a number of component modules. The module is a functional unit receiving signals (data) from one of a number of wires, cables or ports, performing some operations and (possibly) generating output signals on other cables (or ports). The cables connected to a module are fitted with standard male/female connectors so that the output of any module may be directed to the input of any other module by an appropriate interconnection of their cables. Rather than direct interconnection, a special "patch panel", similar to

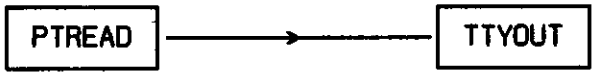
an old-fashioned telephone switchboard, is provided to facilitate the interconnections. Figure I illustrates this model.

In this model modules do not know to whom or to what they are connected. Internal names are used to reference ports for receiving and sending information and the actual supplier or receiver is specified externally by the particular cabling pattern established by the user. This fact, coupled with the "standard connector" assumption, permits the substitution of a module for a functionally equivalent one (or network of ones) at any time.

The use of the system is best illustrated by a simple example. Suppose one wished to construct a program to read text from a paper-tape reader and print it on the teletype. Modules exist for reading (characters) from the paper tape reader (PTREAD) and writing (characters) on the teletype (TTYOUT) -- they can be interconnected as follows:



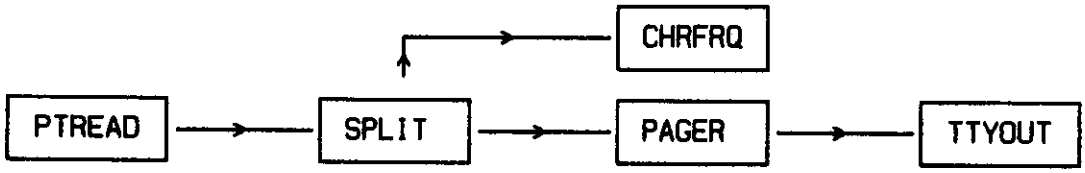
Suppressing the patch panel helps to clarify the diagram in more complex examples, this configuration may be drawn simply as:



Now suppose one would like to add pagination of the output. Further,

suppose there is a module (PAGER) which accepts input and passes it along to its output, but also looks at each data item for a special end-of-line (EOL) character and, after the nth occurance, inserts a special upspace-the-paper (form-feed) character. If the original connection is broken and reconnected as shown below, the desired pagination will result.

Suppose further that it is desired to get a character frequency distribution in the text while the printing is going on. If a module (CHRFQR) to do this exists, the following configuration might be created:



In this configuration, 'SPLIT' is a simple module which, when it receives input, replicates that same input on each of two output ports. Much more complicated configurations could be built in this manner but this example has served to illustrate the general philosophy.

Of course, software modules are not physical objects; they do not have tangible cables dangling out of them. The patchboard does not have a physical existence either. The acts of connection and

reconnection are not accomplished by physical acts but rather by commands typed on a terminal. The precise syntax of these commands is defined in the command language interpreter module (CLI) and may change as more attention is paid to the human engineering aspects of the system (which is considered to be a crucial aspect of the whole project). However, the structure of these commands is intended to reinforce the conceptual model presented above; thus the commands mimic the things one would expect to do to modules physically wired together -- for example: connections may be made or broken at any time, the complete "wiring list" may be displayed or individual wires traced, the signals flowing along a particular cable may be monitored, etc.

IMPLEMENTATION AND RESOURCES

The system model presented in the previous section might be implemented in any one of a number of ways -- each module could have a subroutine or co-routine structure, for example. It was decided to construct each module as an asynchronous (sequential) process. The cabling and patchboard are implemented as a "mailbox" message buffering system. The system is implemented in two pieces: (1) a small "kernel" which includes space management, process management, and message handling primitives, and (2) the modules.

The command language (CL) for using SL230 is implemented as a set of modules using the mechanisms provided by the kernel. It is in no way different from, or more privileged than modules assembled by the user. This construction philosophy permits the CL to be easily modified, permits different versions of the CL for different users, and permits the CL to be easily adapted to various configurations and needs. Finally, the CL, being constructed from modules itself, forms an advanced example of the use of the system and is discussed in a later section on current systems and modules.

THE KERNEL

The kernel consists of a small number of data structures, accessors, and routines for manipulating the structures. The data structures used in the kernel are instances of a smaller number of "classes" of structures (objects, lists of objects, semaphores, and vectors).

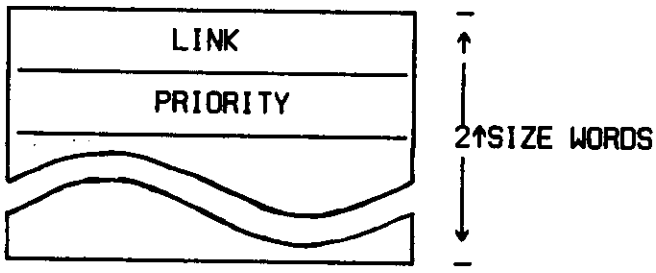
The routines in the kernel are constructed such that each performs an operation appropriate to a class of structures on any instances of a member of that class. This operation is never performed by any other routine. This is a working definition of the term "clean" used earlier. It should be noted that this definition of clean conflicts with similar ones proposed elsewhere [7] in that it implies a strong functional interdependency. It was chosen in favor of a data semantic interdependency because of the clarity and modifiability it affords.

The kernel has been purposely kept small (the entire kernel consists of less than 200 PDP-11 instructions) allowing (1) the design and implementation to be iterated, (2) the kernel itself to be an object of study in a systems programming course, and (3) a usable subset of the total system to be used on a minimal (4K) PDP-11 configuration.

The following is an English description of the data structures and their associated manipulative routine supplied by the kernel.

(1) objects

An "object" is a data structure which is composed of 2↑N ($1 \leq N \leq 16$) words, two of which contain a link field (objects are frequently chained together on lists), and a priority field (when on a list, objects are always in priority order).



All system objects have system defined names associated with the offsets from the base address of the object. These names are always used when accessing the areas of an object and are given beside the locations in the diagrams of the objects (each block in a diagram represents one word).

The routines for manipulating objects are:

a) get (n) allocate memory for an object of size 2↑"n" and return its address.

b) release (a,n) deallocate the space for an object whose address is "a" and size is n. The value of "release" is undefined.

c) copyold (a,n,b) copy the contents of an object whose base address is "a" and size is 2↑"n" words into an object whose base address is "b"; exactly 2↑"n" words will be copied. Return the base address of "b".

d) copy (a,n) create an object of size 2↑"n" and make its contents identical to those of "a"; return the address of the new copy.

e) link (a,h) link the object whose base address is "a" on to the list whose header address is "h". The object will be linked into the proper priority position on the list. Return the address of "a".

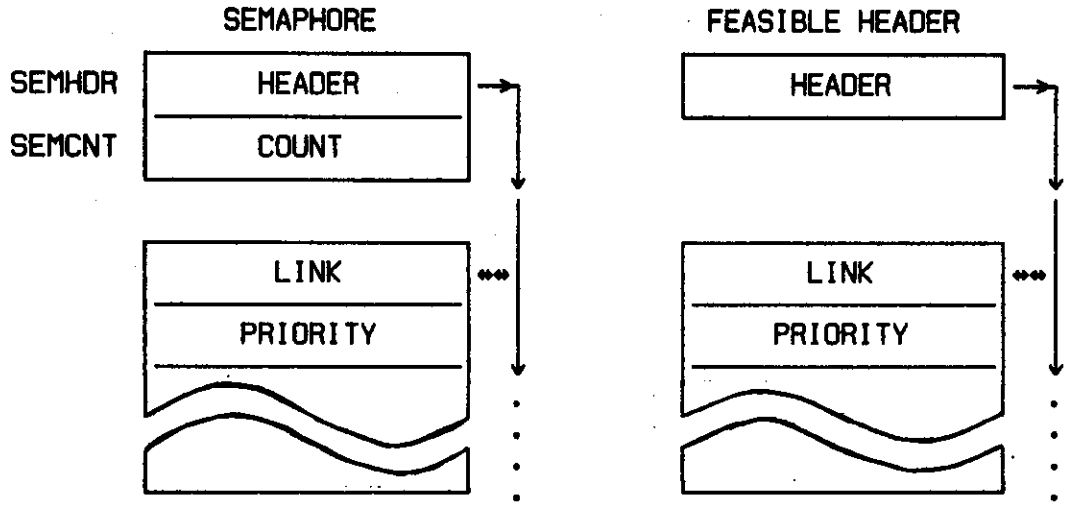
f) delink (h) remove the first object, that is the highest priority one, from the list whose header address is "h" and return the address of this object.

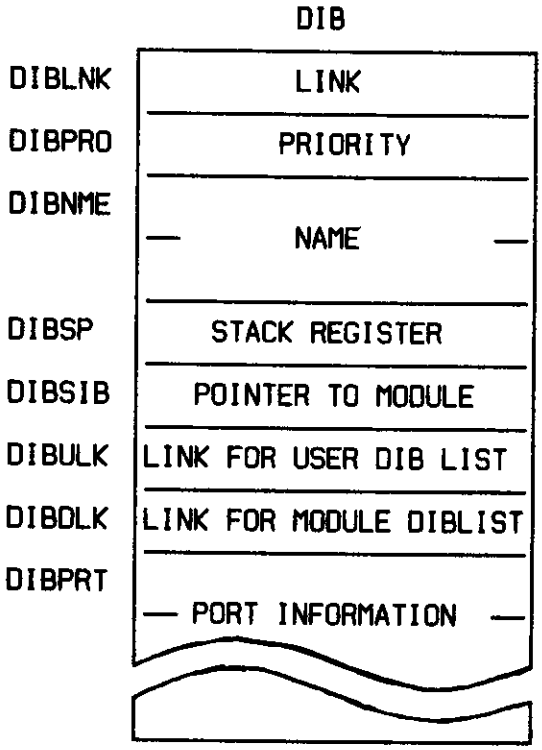
g) swap (h1,h2) delink the first object of the "h1" chain and link it onto the "h2" chain; return the

address of the swapped object.

(2) The "feasible" list, semaphores, and synchronization

A particular class of objects are called "DIB's", Dynamic Information Blocks. DIB is the name given to what has been called a "process description" in other systems, and contains relevent state information for a process. The "feasible" list is a chain of all the DIB's for processes which are ready to run. All other processes are "pending on a semaphore" and these DIB's are chained on a list associated with that particular semaphore. The reader is assumed to be familiar with Dijkstra's P and V primitives and their use for process synchronization [6].





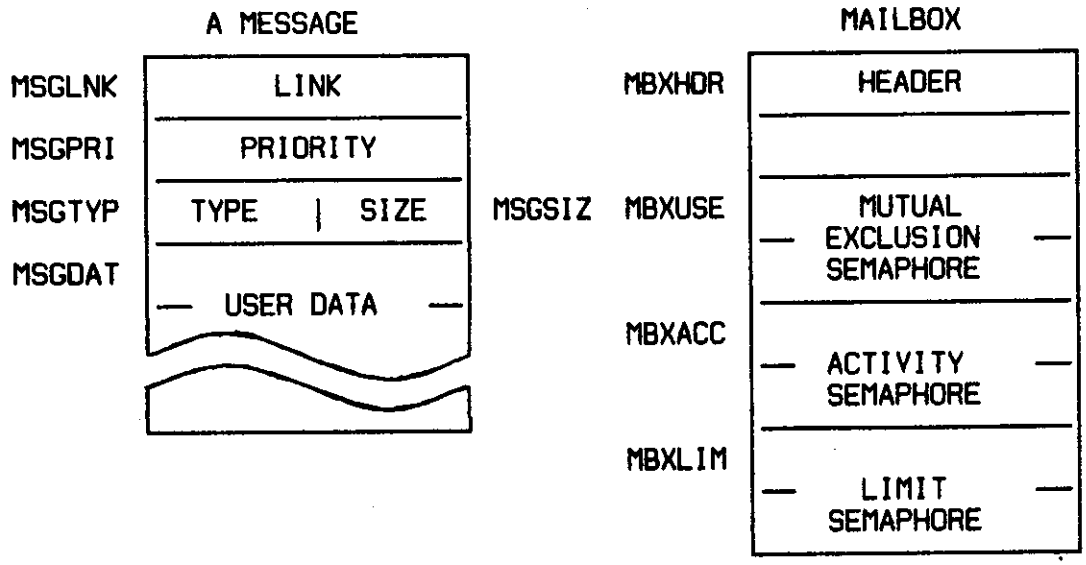
The routines which manipulate semaphores and the feasible list are:

savstart	saves the context of the current process on its stack, saves the stack pointer of the current process in its DIB, and initiates the process whose DIB is at the top of the feasible list by retrieving its stack pointer and restoring its context from the stack.
P (sem) V (sem)	Dijkstra's synchronization primitives.

(3) Messages, Mailboxes, Ports, and Communication

Processes communicate by sending and receiving objects called "messages". Modules do not send messages directly to other modules but rather to "ports". A port is a local (to the module) name for one of the cables in the model -- thus modules are not aware of which other modules they receive messages from nor send messages to; they are aware only of their own local port names.

The patchboard is implemented as numbered a set of "mailboxes" -- data structures which contain (among other things) a (possibly empty) set of messages. Patchboard connections are accomplished by making the "port information" portion of a process's DIB reference a particular mailbox by its number.



The message handling primitives are:

send (m,p) A copy of the message whose base address is "m" will be sent to the mailbox connected to port "p". If the mailbox is currently full the sending process is suspended until space for the message becomes available.

receive (p) Return the address of a message in the mailbox connected to port "p". The message is removed from the mailbox. If no messages are currently in the mailbox the process is suspended until a message is sent to it.

SYSTEM FUNCTIONS

Although the kernel supplies all of the support facilities necessary for the running process, there is a set of functions that

is useful to have in a common area where it may be shared by all the modules. These functions are those which are either performed by many modules (but too simple to be an independent module) or are best performed with more access to system data structures.

The system functions are neither necessary for the operation of the kernel nor do they form a permanently defined set. They exist solely as a convenience for the user.

(1) Process creation functions

The kernel supplies routines to support processes but it does not provide any means to create them or interconnect them. A module could perform these activities but this might endanger the reliability of the system. The process control functions are:

create (a,b,c) create an incarnation of the module whose base address is "a"; give it the name located in the 2 word area whose address is "b" and use the first 7 words of the area whose address is "c" as the DIB's priority and its context values (if c=0, the priority becomes the modules priority and the context values are undefined). Link the DIB on the feasible list and on the system DIB list. Return value is the address of the new DIB.

connect (a,b,c) disconnect the Ath port of the DIB whose base address is "b" from any connection and reconnect it to the mailbox number "c". If "c" is 0, allocate a new mailbox. Return values are the mailbox address and number.

(2) Arithmetic functions

The PDP-11 is a mini-computer and does not supply all the hardware arithmetic functions that are normally used by processes. Rather than every module having its own routines, the most useful are

provided in the system functions (if the functions become available as hardware, the modules can be changed to take advantage of it and/or the system functions can be changed to be more efficient). The arithmetic functions currently provided are:

multpy (a,b)	unsigned integer multiply "a" times "b" and return the double precision result.
mult50 (a)	has the result of "multpy(a,#50)" ("#" denotes octal)
divide (a,b)	unsigned integer division of "a" by "b", returning a quotient and a remainder.
div50 (a)	has the result of "divide (a,#50)".
log2 (a)	calculate the log, base 2, of "a" and round it to the next higher integer. Return value is the log and the difference between "a" and 2↑(first return value).
power2 (a)	calculate 2↑"a" and return it.

(3) Conversion functions

The names in the DIB's and in the modules are in RADIX 50 (allowing 3 characters/word). The conversion functions from RADIX 50 and ASCII are:

conv50 (a)	returns the RADIX 50 value of "a".
conasc (a)	returns the ASCII value of "a".

MODULES

The modules are the basic building blocks of all SL230 systems. As such there are several restrictions placed on the code and several conventions that should be followed when modules are coded. A SL230 module (source file) is composed of 3 parts, (1) documentation (for programmer use), (2) Static Information Block (SIB) and (3) the executable code. A detailed description of the parts follows and an example of a module is given in appendix A.

DOCUMENTATION

The documentation is a description of what the module does and how to sucessfully interface the module to other modules. It gives all the information that is externally visible (message formats, what the module does,...) but it doesn't give the algorithms used or the internally defined data structures. The format has the purpose of insuring that all information necessary for the proper use of the module is available to the user. This forces the modules to be clean (as defined earlier), the internal data is unavailable and it can not be used in assumed connections. The format consists of keywords and descriptions (see the example in appendix A). The following describes what the keywords signify:

MODULE module name as given in the SIB

FUNCTION: English description of the function of the module, exclusive of port information, message formats and algorithms

PARAMETERS: the parameters of the module as given in the SIB, which are: priority, stack size, dib size, number of ports, and module size (approximate)

EXTERNAL: external variables referenced by the module. These are given in the format:
<English name> (<system name>)
example:
TELETYPE INPUT STATUS REGISTER (TØKBS)

PORTS: description of the message format that is sent or received through each port, including what the module uses the port for. The format is:

<port number> <port name> <port function>

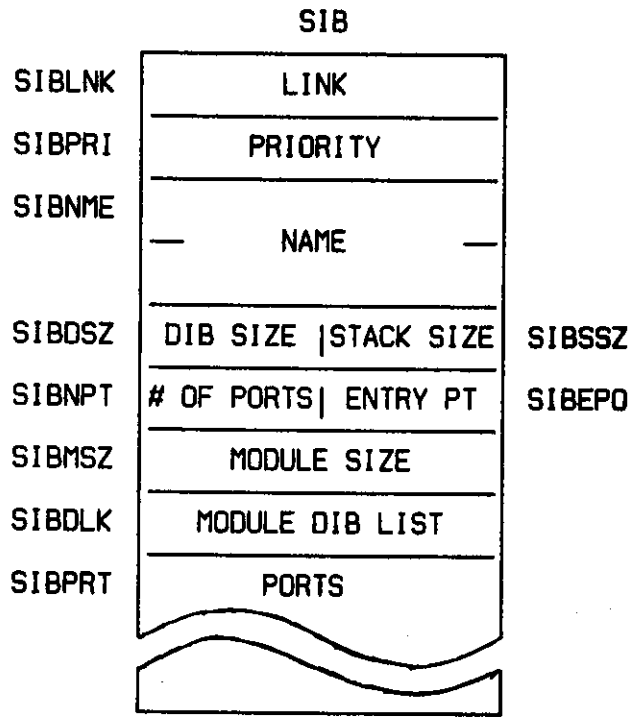
CONNECTIONS: modules that the module is "normally" connected to. The format is:

"CONNECTIONS:
PORT MODULE
<PORT NUMBER> <MODULE NAME>:<PORT NUMBER CONNECTED TO>

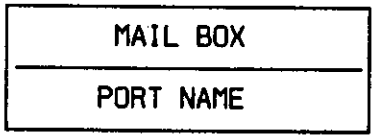
ASSEMBLY: all the files with which the module must be assembled, usually the files which define interface languages (explained in a later section) and assembly time options

SIB

The SIB is the name given to the object that contains the system module data constants for a particular module. All SIB's are linked on the system SIB list (SIBHED) and are available to all users. The SIB is located at the head of a module's code and its format is:



PORT FORMAT



The contents of each field (at load time) must be:

SIBLNK - module size (in bytes). This value is used by the loader to allocate space.

SIBPRI - module priority

SIBNME - RADIX 50 value of the module name.(1)

- - - - - - - - - -

(1) If PALX11 is used, this value must be calculated. If MACX11 is used then the .RAD50 directive may be used.

SIBSSZ - one byte value indicating the size of the stack necessary to run an incarnation of the module. The value is the log base 2 of the number of words in the stack. The minimum value of the field is "5" resulting in a stack of 2↑5 (=32 words) of which 20 words are required for system overhead (hold the context when the process isn't running, interrupt stacking, etc.) leaving 12 words free for the user (subroutine calls, local storage, etc).

SIBDSZ - the log, base 2, of the number of words in a DIB of this module. The minimum value of the field is "3" giving a DIB of 2↑3 (=8 words) but not allowing any ports, or at least "4" (2↑4 =16 words), allowing 4 ports.

SIBEPO - module entry point (offset from the SIBLNK field to the first executable instruction).

SIBNPT - number of ports in the module.

SIBDLK - one word DIB list header, initial value 0. When the module has incarnations, this field is the header for a list of the DIB's.

SIBMSZ - one word size field, initial value, undefined. When the module is loaded this field is given the value of the number of words that the module actually has allocated to it (not necessarily an integer power of 2).

SIBPRT - SIBNPT port entries, mailbox number is 0, the port name is the RADIX 50 value of a 3 character name and may be 0.

CODE

The actual coding practices used in the modules are not important as long as they do not violate certain restrictions and conventions. These restrictions are imposed to insure that the modules function properly in the software laboratory environment and do not harm the system.

The most important set of restrictions centers on the possibility of multiple incarnations of a module. This property of SL230 modules forces all code to be pure and re-entrant. A pure module, in this sense, has two implications: (1) the code must not contain any instructions that alter other instructions in the module and (2) the module must not contain any local storage.

Since local storage can not be located in the module, it must be allocated from core when an incarnation of the module is first started. There are two ways storage may be allocated:

1) from the stack
2) using a new section of core

Both of these methods work quite well and require about the same amount of work.

To allocate from the stack, it is first necessary to start out with a large enough stack. The first instructions (outside of any and all loops) subtract the proper amount from the stack pointer (SP) and save the SP in a register. Throughout the rest of the module the local storage is referenced as indexed on the register.

To allocate local storage from core, a call on GET is done for the required core and the address of the core is saved in a register. It is accessed in the same manner as space on the stack.

Allocating from the stack is the preferable (possibly may become required) method of allocating storage. Since the stack assigned to a process is completely is defined, the system can easily control it. This may become important when it is desired to implement a function to delete an incarnation of a module. The major reason why the delete function does not exist in the current version is the virtual impossibility of deallocating all the space (such as messages, local storage, etc.) an incarnation has. This problem has been given some attention but no suitable (neat and easy) solution had been found that did not involve a considerable overhead in the allocation and deallocation process. (1)

Another restriction on the code is that it must be entirely relocatable. This restriction results from the lack of hardware relocation facilities on the PDP-11. The problems of writing relocatable code are dicussed in the later secton on coding hints.

If the code in a module comforms to these restrictions, then there will be no problems in running it under SL230 (once it is debugged).

- - - - - - - - - - - - - -

1. A feasible solution appears to be one having the module deciding when and if an incarnation can be deleted. This could be done with a system function and a delete bit in each DIB.

Conventions

In the process of coding many modules there are certain coding conventions that have been found to be useful. These conventions do not actually affect the module code (other than in the conceptual manner in which the modules are coded) and apply mostly to coding done in PDP-11 assembly code (PAL11). They are not required but it is adviseable to use them since they do make coding easier.

To facilitate the use of local storage it is desirable to use direct assignments rather than numeric offsets. This allows two things to be done, the format of the local storage may be changed easily and the format is well described for later referencing. At the start of a module, values are assigned to names and a description given of the data. An example of this would be:

```
                    ;  ↑**********************↑
NAME    =0          ;  ↑  RADIX 50 NAME     ↑
                    ;  ↑**********************↑
NUMBER  =2          ;  ↑ FILE NUMBER        ↑
                    ;  ↑**********************↑
REFCNT  =4          ;  ↑ NUMBER OF REFS     ↑
                    ;  ↑**********************↑
```

The data would be acessed by an instruction like:

```
        TST     REFCNT(R5)      ;ANY REFERENCES?
```

It is also desirable to use direct assignments of variables for port assignments. This allows the assignments to be easily changed and provides more information for anyone reading the code.

The kernel routines and system function, defined earlier in this

report, are accessed, on the PDP-11, by means of the TRAP and EMT instructions (TRAP for kernel routines and EMT for system functions). These instructions allow an argument which is used by the TRAP and EMT routines to determine which routine is being called. Thus, if the SEND routine is number 2 the calling instruction would look like:

```
TRAP    2
```

allowing position independent accessing of the routines.

The system data that a user might need has been defined in the file SYMHED. In addition to defining the hardware registers (R0 - R5, SP, PC, PS, I/O registers), SYMHED defines all the names of the system data structures (SIB, DIB, messages, semaphores) with the identifiers given in this report. SYMHED also defines a mnemonic and gives the relavent information about the parameters and return values for each argument of the TRAP and EMT functions. This allows the kernel routines to be called by their name. By assembling SYMHED with a module all that is necessary to access the send routine is to code the instructon:

```
TRAP    SEND
```

after setting up the parameters.

SYMHED defines the "interface" language between the module and the kernel. An interface language is a defination of the assumptions, structures, commands and conventions that exist between two objects that interact with each other (such as kernel and module, module and module). If a module is written that has a non-trivial message format, a large set of possible commands, or uses a common

data format, it is best to create an interface file like SYMHED. Defining interface languages in a file like SYMHED has a great advantage over putting the assignments at the start of every module. It is easier to access parts of a data structure using mnemonics and it allows the format of the structure to be changed with only the cost of a reassembly instead of a change and reassembly of every module that accesses that structure. Another advantage is having the interface completely defined so other programmers can use it.

Hints

The following paragraphs describe several PAL11 oriented tricks that can be used to ease the job of writing a module.

As someone who closely studied the SIB format may have noticed, some of the information required would be non-trivial for the programmer to calculate. Specifically this is the module size in the location SIBLNK and the module entry point offset in the location SIBEPO. An easy way to get these values is to have the assembler calculate them. The module size is calculated by having a label at the start of the SIB and one at the end of the module (after the last instruction). The start of the module would have an assignment of the form:

MODLNK=<LASTLABEL>-<FIRSTLABEL>

and the first word of the SIB would have the value of "MODSIZ". Alternately, if the labels were "SIB" and "LAST" the start of the SIB could look like:

```
SIB:    .WORD   LAST-SIB        ;MODULE SIZE
```

The same technique can be used for SIBEPO. (see the example in appendix B)

It is sometimes useful to assemble several modules together, so, instead of putting a ".END" at the end of each module, it is usually better to put the ".EOT" directive there. All currently existing files follow this practice and for this reason the file "TAIL" exists. It contains only one line, a ".END" statement.
Example: to assemble the module "DTACON" we find from the documentation that it requires the files "SYMHED" and "DSKCOM" assembled with it. The assembler command string would look like:

```
DTACON,/CDTACON←SYMHED,DSKCOM,DTACON,TAIL
```

NOTE: SYMHED must be the first file in the string since it defines the hardware registers.

The lack of relocation hardware forces all modules to be location independent. When a module is loaded, it can and will be placed almost anywhere in core. On most machines this requirement would place a great burden on the programmer and/or the programming language. On the PDP-11 relocatable code is easy to write, the only problems requiring care are accessing fixed addresses (the PS word, I/O registers, etc) and accessing module information (such as command vectors). On the PDP-11 this requirement is easy to fulfill due to the ability to do indexing relative to the program counter (PC). The

only problems occur when it is desired to access a vector of data within the module (such as a command vector) or when trying to access a fixed location in core (such as the PS or I/O register).

A fixed address can be referenced position independently only by the "deferred autoincrement on the PC" mode. This mode forces absolute instead of relative addressing. The correct and incorrect methods of referencing the PS would be:

```
MOV     @#PS,-(SP)      ;RIGHT (ABSOLUTE)
MOV     PS,-(SP)        ;WRONG (RELATIVE)
          (NOTE: timing is identical)
```

Accessing module information (indexed by a register) involves using the PC to find where the module is located and calculating relative displacements. If it is desired to use a vector as a command break (a vector indexed by a register the correct and incorrect methods of coding are:

```
        VECTOR:  COMMD1,COMMD2,.....

        JMP      @VECTOR(R0)      ;WRONG METHOD

        ADD      PC,R0            ;CALCULATE POSITION
HERE:   MOV      VECTOR-HERE(R0),R0 ;GET RELATIVE OFFSET TO LABEL
        ADD      PC,R0            ;MAKE IT ABSOLUTE
HERE2:  JMP      -HERE2(R0)
```

Explanation: R0 has a value (even) that is to be used to index into VECTOR. Since the module may be located anywhere in core using the label "VECTOR" as an absolute value will pick up a word from core that corresponds to where the assembler put the module (usually 0).

Instead, the PC is added to R0 so that R0 points to the label "HERE" offset by the amount that it formerly contained. The desired word is picked up by indexing with the displacement from"HERE" to "VECTOR". This is now the value of a label, as the assembler saw it. So the initial process is repeated with the final instruction a "JMP" if the process is for a command break or possibly a "MOV" or "CMP" if the vector contained data.

DESIGN

Thus far, the discussion has centered on how to write a module rather than what should go into it. From what has been said, it is evident that SL230 will actually support almost any piece of relocatable code that has a SIB on the front of it. This is due to the impossiblity of checking or protection on the PDP-11. Designing a module as if it was a stand-alone program is ignoring the resources of the software laboratory. The entire concept of SL230 rests on the general availability of small functional modules. Proper design of a module is of the utmost importance so as to maximize its usefulness.

The guiding philosophy should be to design modules that are globally useful. This means we want to design the modules small and functionally simple. Complex functions are generated by connecting many of these simple modules together. Unfortunately, there is a lower limit upon the size of a module. At some point the overhead involved in the system structures (DIB and stack, minimum = 48 words) is bigger than the module. In most instances this is undesirable. If modules this small are implemented, core is quickly lost through fragmentation and cluttering. A module in this range should be re-examined to see if it is really useful. If there are few uses for it, then it possibly should be included as a subroutine in the module that uses it. If there are many users the possibility of including the module as a system function should be considered. An example of a small module that can not reasonably do either of the alternatives is the TTYIN module. This module is 32 words in length (10 of which

are the SIB) and is an independent module solely because it does I/O. By having it do the I/O, other modules become more generally useful.

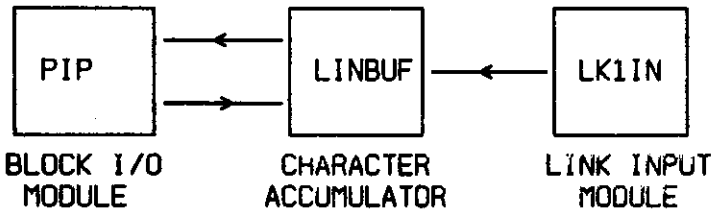
Modules should have a size on the order of 75 to 400 words. (The figure 400 results from writing many modules and evaluating what is contained in each. It is not an upper bound but rather a guide to be used when designing modules.) If a module is larger than 400 words it probably incorporates several functions that independent modules should do. It should be examined to see whether it can be broken down into smaller modules. An example of a "large" module is the Command Language Interpreter (CLI). It has a size of 512 words and consists mostly of special cases (the various commands). It would be difficult to divide the CLI into separate modules due to the common data base that the commands require and the fact that each individual command is too small to be an independent module.

The normal condition for the existence of large modules is the grouping together of several small sections of related code that are all accessed in the same manner. A possible way to eliminate this type of module is to provide a module that consists almost entirely of ports and the code merely sends the incoming messages out the various ports according to some well defined rule. If the resulting small modules are not generally useful it is not evident it is worth the effort (and overhead!) to do this.

Most modules occur in the context of a larger system or project and are originally designed as a part of that system. Dividing a system into modules can be done in many ways, not all of which are desirable. An example of modularizing a project is given in [9] in

what we consider to be one of the better ways to divide a system. A system should be divided along functional boundaries instead of the usual data flow boundaries. Functionally interdependent modules are easier to change then data interdependent ones. Since we wish to have the facility of easy changablity in the system, we must have the modules functionally interdependent, keeping data interdependency restricted to the messages that pass between two modules.

SL230 lends itself to functional interdependency. It is easy to see this in terms of an I/O module. SL230 has two classes of I/O devices, single character devices (teletype, link) and block devices (disk, DECtape). The I/O messages from different devices are not identical. If a module were designed requiring a block formatted input, it could not connect directly with a character oriented device. By keeping the I/O functions independent we seem to be losing access to some of the devices from a particular module. Obviously all that is necessary is to insert a conversion module between the two. If a module requires a particular type of I/O input this is the type of solution that should be considered. The link dedicated system provides us with an example of I/O type dependent modules. If it would be desirable to send an ASCII file from the PDP-10 to the PDP-11 the character would come into the 11 through the link input module(LK1IN). This module is single character oriented so that if we wished to use PIP11 to transfer the file to disk a direct connection could not be made. Instead, a character-to-buffer module would have to be inserted between the two. Schematically this looks like:



Doing input from the link this way allows the continued use of the single character capabilities of the link and also allows us to transfer files with a minimum of work. This solution would be superior to writing a new LK1IN module for it also generates the LINBUF module which should be useful elsewhere. (The link system is described in the following section on current systems)

As in most problems, the dividing of a project into modules involves the making of various trade-offs. In the software laboratory the desired end result is to have as many useful modules as possible. By checking on the kinds of existing modules it is (should be) possible to find most of the programming work done.

CURRENT SYSTEMS AND MODULES

At the present time there are two major systems that have been designed; a command language system and a link oriented system (see schematics in appendix B). The command language system is designed to provide the resources necessary to debug modules and construct systems. The human engineering aspect of the command language has been given considerable attention and the commands are designed to allow efficient use of the human resources available. The commands are given in the documentation of the Command Language Interpreter (CLI) module (see apendix D) and will not be given here.

The command language system can be easily extended if a user wants it to be. If a new command or facility is desired, a new module can be written to implement the command or an existing module can be modified. There is nothing permanent about the current version of the command language module other than the kinds of commands that it provides. The current version is actually the third one and represents a year of experimentation and use of other versions.

The link dedicated system is used to communicate with another computer by means of a link connecting the two machines (currently the link goes to a PDP-10). Since the PDP-11 is a small machine, the second machine is used to edit and assemble PDP-11 files and the binary output is sent to the PDP-11 over the link. Resources are available in the link system to transfer from the link to any other

block I/O device. This system allows the rapid debugging of modules (or systems) since the power of a bigger computer is available to the user. For a more exact description of the link system, see the schematic in appendix B and the description in appendix C.

In addition to the modules composing these two systems, there are several others that have been written. The documentation of all cuurrently existing modules is given in appendix D.

FUTURE PROJECTS AND SYSTEMS

There are only a few major projects left involving changes to existing systems. They are outlined in the following paragraphs to give examples of the kind of projects that could be considered. The particular ones given are those for which a solution is thought to be easily available.

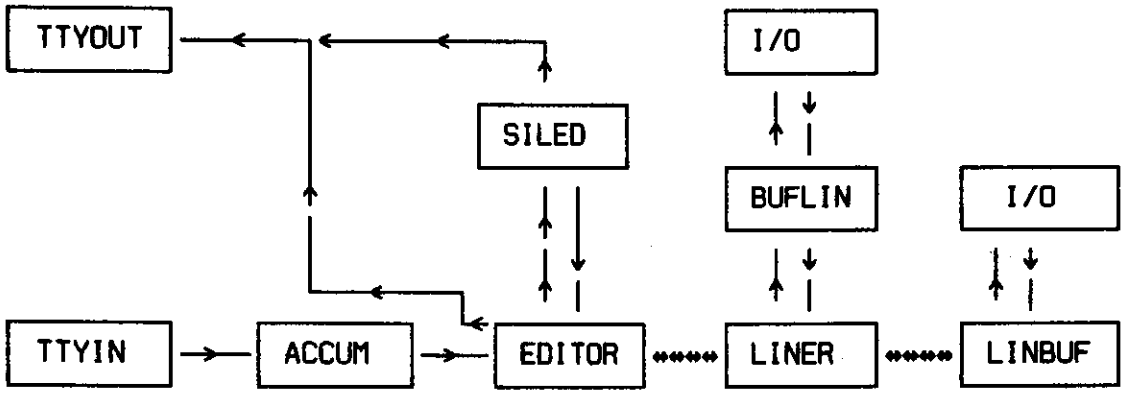
There are some changes that should be done to SL230 itself (as opposed to modules). One of these is the addition of a delete function. As was mentioned earlier in this report, this function is not in the current version because of the difficulty in deallocating the core assigned to a process.

Another major change to SL230 involves the manner in which a system is initially loaded. In the current version, each system must have its own system assembly since information about which modules are loaded exists as a vector in the system. A better way to initialize the system is to have the capability of using a load file that specifies the modules to be loaded and the connections to be made. This is easily implemented by using a subroutine, "DOEVER", from the command language interpreter module. By making this routine part of the system, all that is necessary to perform the proper connections (and loadings) for a system is to give the routine the correct data structure. The source of the data structure could be anywhere and thus could be a file on an I/O device. This would allow more efficient system loading and the system in core could be changed more easily. It would also ease the implementation of a command

language with each system (the link system does not have command language facilities).

Sometime, it might be desirable to change SL230 into a multiuser system. The PDP-11's available for this project were not big enough for more than one user so a multi-user system could not be implemented. Since SL230 is already designed as a multi-process system, it would be a simple matter to have each user have one process for his use. This would be the equivalent of the way most current operating systems are implemented, but it would provent the user from accessing most of the resources provided by SL230. of SL230 (multiple feasible lists, a recursive defination of the kernel, etc.) but they will not be discussed in this report.

There are many systems that could be designed for the software laboratory. Most of the first systems built should have the purpose of building up the library of modules in addition to building a useful system. Among these projects are a few that can be done with very little work, the modules that should be written are readily apparent. One of these would be a text editor. The editor need not be complex but should have a great deal of power. The actual design of the modules will depend upon the type of editor used (text mode, line mode, etc.) and the desired features of that type. It should, however, contain modules that are common to all types of editors. A possible design of an editor is:



The only modules that are not written are the three editoring modules, EDITOR, LINER, SILED. The first is the type dependent editor controler, it is the one that scans the input and decides what to do. Liner is a simple module that handles a list of strings (in messages). SILED is a more sophisticated part of the editor. It implements an "alter" command (a command which would allow the internal editing of a line of text with a line mode text editor) and would not be necessary for an initial version of the editor. All the other modules exist in some form. This design is neither the only possible design nor necessarily the best. It is one of the simpler ones and should be easy to implement.

Other systems that could be implemented include assemblers, compilers and text justification programs. Each of these should also have several modules implemented for each function, such as several symbol table modules, optimizers etc. When several projects such as these are completed there will be a useful library of modules available for users.

APPENDIX A

This appendix contains an example of a module.

```
;
;
;;      MODULE  LINBUF
;;
;;
;;      FUNCTION: ACCUMULATE SINGLE CHARACTERS INTO BUFFERS.  THIS
;;              MODULE LOOKS LIKE AN INPUT I/O MODULE (BUFFER SIDE)
;;              AND WILL CONVERT A SINGLE CHARACTER I/O MODULE INTO
;;              A BUFFER ONE.  TERMINATOR ON THE INPUT IS THE
;;              CHARACTER CONTROL Z (026 ASCII DECMAL).
;;
;;      PARAMETERS: PRIORITY=   20000
;;              STACK SIZE=     2↑5
;;              DIB SIZE=       2↑4
;;              NUMBER OF PORTS= 3
;;              MODULE SIZE=    170 WORDS
;;
;;      PORTS:  PORT    NAME    FUNCTION
;;               0              I/O COMMAND INPUT PORT. A BLOCK
;;                              ORIENTED I/O COMMAND IS ACCEPT
;;                              THROUGH THIS PORT. FOR FORMATS SEE
;;                              DSKCOM.
;;               1              I/O REPLY PORT
;;               2              CHARACTER INPUT PORT. CHARACTER IS
;;                              THE FIRST BYTE IN THE DATA PART OF
;;                              THE MESSAGE
;;
;;      ASSEMBLY: SYMHED, DSKCOM
;;
;;PAGE
```

```
;
;
                        ; ↑********************↑
BUFADR=0                ; ↑ BUFFER ADDRESS     ↑
                        ; ↑********************↑
BUFCT=1                 ; ↑ BUFFER COUNT       ↑
                        ; ↑********************↑
;
RECIVS=0
REPYS=1
INPUTS=2
;
LODSIB:
;
LODMDS=LODLST-LODSIB
LODEMPT=QSTART-LODSIB
;
        .WORD   LODMDS,20000
        .WORD   46166,7716      ;"LINBUF"
        .BYTE   5,4,LODEMPT,3
        .WORD   LODMDS,0
        .WORD   0,0     ;PORT 0
        .WORD   0,0     ;PORT 1
        .WORD   0,0     ;PORT 2
;
;PAGE
```

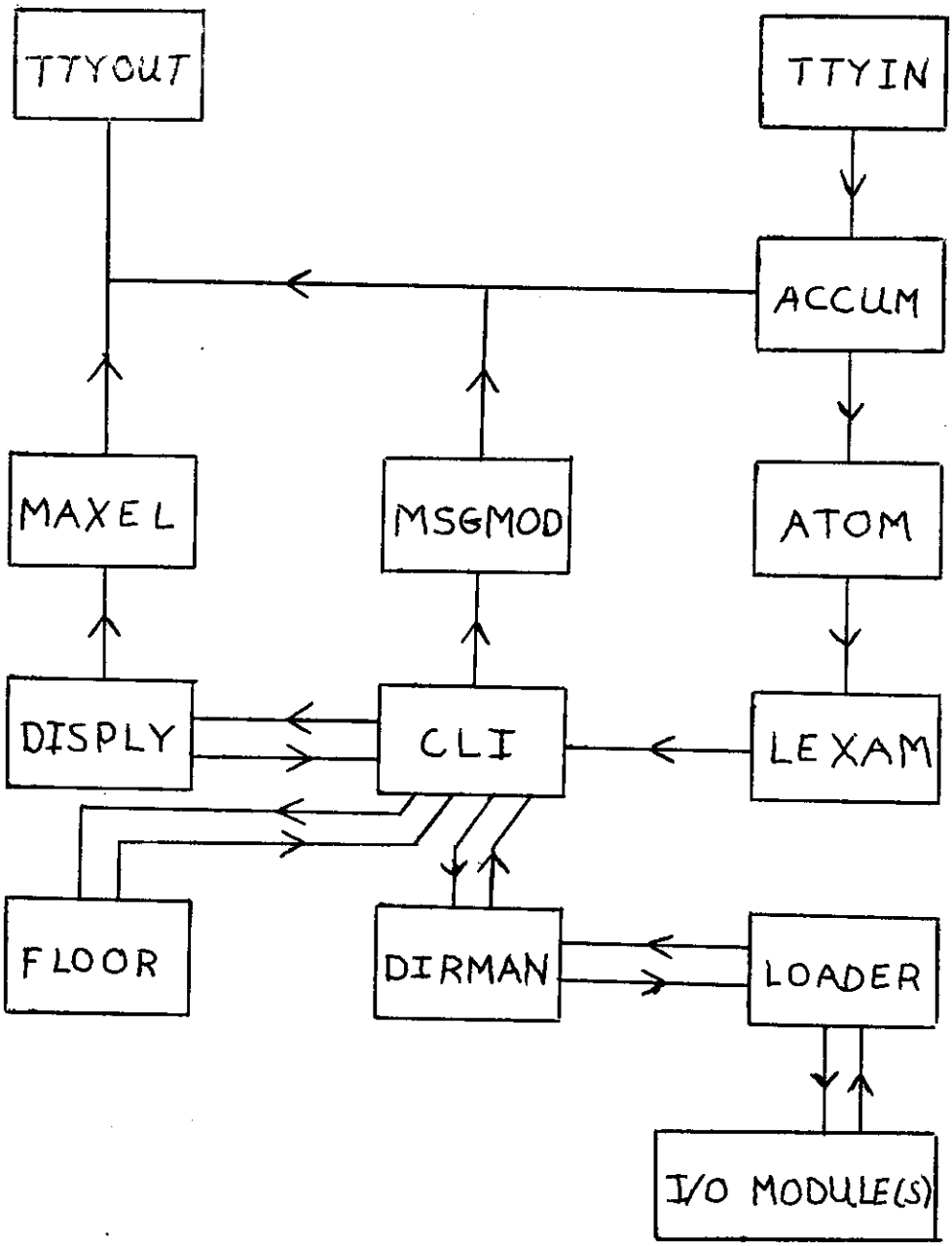
```
;
QSTART: CLR     -(SP)
        CLR     -(SP)
        MOV     SP,R2           ;POINT INTO THE STACK
        CLR     R3              ;ZERO R3
        BR      ADONE           ;START
;
;
RSTART: MOV     #-1,DEVCMD(R3) ;ERROR, AND TELL HIM SO
BDONE:  JSR     PC,CUTFIL
ADONE:  JSR     PC,INFIL
        CMPB    R0,#OPIN        ;OPEN FOR INPUT
        BNE     RSTART          ;IF NO, GIVE HIM A NEGATIVE NUMBER
        JSR     PC,COUMM        ;IF SO, OK
        CMPB    R0,#READF       ;HAVE THE NEXT, IS IT A READ?
        BNE     HUNTH           ;IF NOT, HE GOOFED!
;
;
;
LOP2:   JSR     PC,GETBYT       ;GET ONE BYTE
        CMPB    R0,#CTRZ        ;IS IT A CONTROL Z (THE END)
        BEQ     REND
        JSR     PC,PUTBYT
        BR      LOP2
;
REND:   JSR     PC,SENDBF       ;GO SEND THE CURRENT BUFFER
REND2:  CMPB    R0,#READF       ;WAS REPLY A READ? IF SO SEND EOF
        BNE     HUNTH
REND3:  MOVB    #EOF,DEVCMD(R3)
        JSR     PC,COUMM
        BR      REND2
;
;
;
HUNTH:  CMPB    R0,#CLOSZ       ;A CLOSE?
        BEQ     BDONE           ;IF SO, START OVER
        CMPB    R0,#RELESE      ;A RELEASE?
        BEQ     BDONE           ;IF SO , START OVER
        TST     R0              ;AN ASSIGN?
        BNE     REND3
        BR      BDONE
;
;PAGE
```

```
;
;       THE FOLLOWING IS THE GETBYTE ROUTINE
;
PUTBYT: INC     2(R2)           ;INCCREMENT THECHARACTER COUNT
        BGT     GETBUF          ;IF >0 THEN NONE LEFT, GET MORE
        MOVB    R0,@(R2)        ;GET THIS BYTE
        INC     (R2)            ;INC THE POINTER
        RTS     PC              ;RETURN
;
;
GETBYT: MOV     #INPUTS,R0
        TRAP    RECIV           ;GET THE BYTE
        MOVB    MSGDAT(R0),-(SP)        ;SAVE THE DATA
        MOVB    MSGSIZ(R0),R1
        TRAP    RELEAS          ;RELEAS THE MESSAGE
        MOV     (SP)+,R0
        BIC     #177600,R0
        RTS     PC
;
;PAGE
```

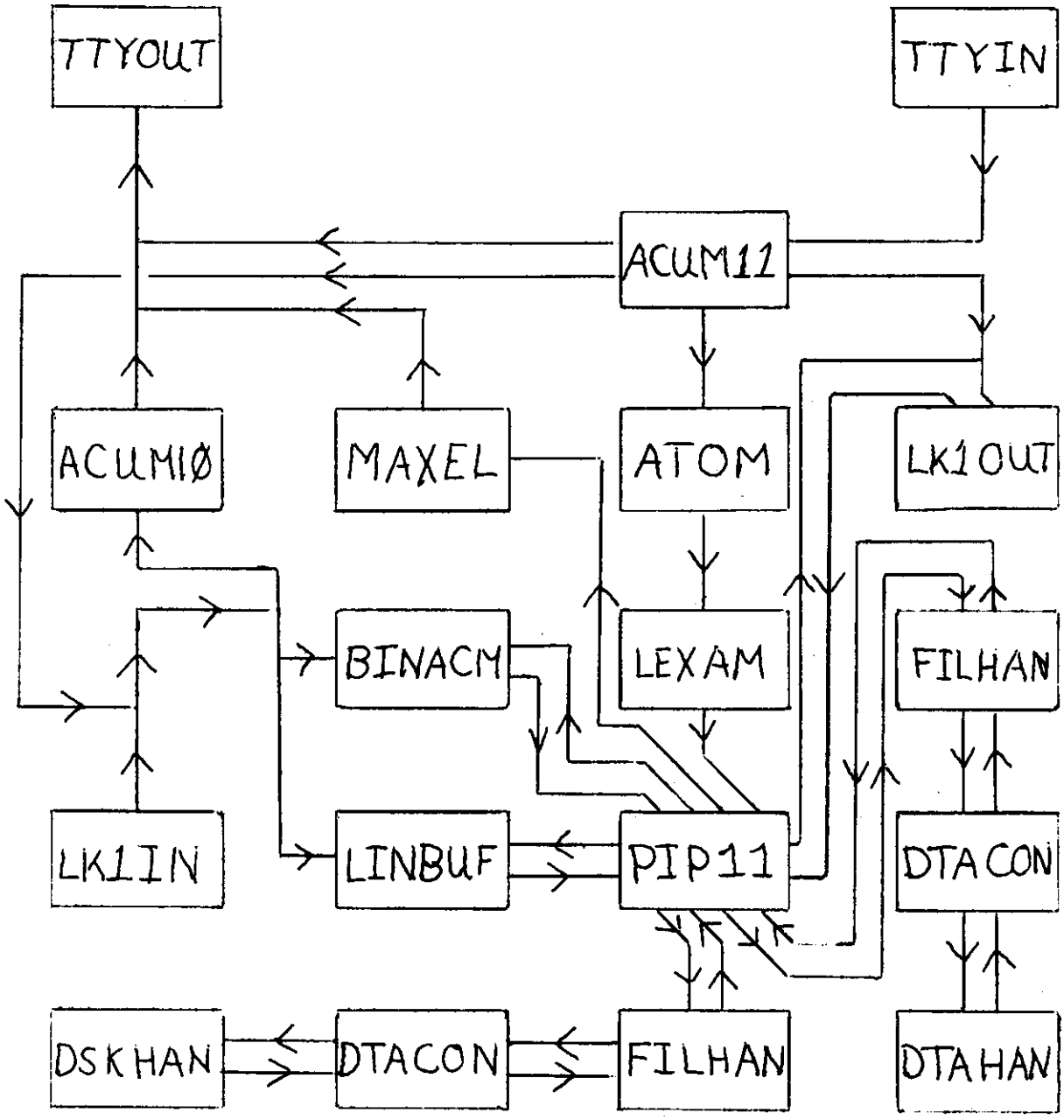
```
;
;
GETBUF: MOV     R0,-(SP)        ;SAVE THE DATA BYTE
        JSR     PC,SENDBF       ;SEND THE CURRENT BUFFER(IF ANY)
        CMPB    R0,#READF       ;READF NEXT?
        BNE     BOMB            ;IF NOT, BOMB
        MOV     DEVLNK(R3),R0   ;GET THE BUFFER PROVIDED
        BNE     OKSPS           ;MAYBE NO BUFFER?
        MOV     R3,R0           ;DITCH MESSAGE, IT MAY BE TOO SMALL
        MOVB    MSGSIZ(R3),R1   ;WE NEED A SIZE 4 AND IT MAY BE SIZE 3
        TRAP    RELEAS          ;FOUND OUT THE HARD WAY!
        MOV     #4,R0
        TRAP    GET             ;NOW WE HAVE THE RIGHT SIZE
        MOV     R0,R3           ;SAVE IT
        CLR     (R0)+           ;ZERO THIS MESS
        CLR     (R0)+
        MOV     #4,(R0)                 ;SIZE
        MOV     #10,R0
        MOVB    R0,DEVSIZ(R3)   ;PUT IN THE SIZE
        TRAP    GET             ;GET THE BUFFER
        MOV     R0,DEVLNK(R3)   ;PUT IT IN THE MESSAGE
OKSPS:  MOV     #-776,2(R2)
        TST     (R0)+
        MOV     R0,(R2)
        MOV     (SP)+,R0        ;RESTORE DATA
        BR      PUTBYT          ;PUT OUT THE CURRENT CHARACTER
;
BOMB:   MOV     R2,SP           ;RESET THE STACK
        BR      HUNTH
;
SENDBF: TST     (R2)            ;WE DO HAVE A BUFFER, DON'T WE?
        BEQ     NOSTUFF         ;IF NOT, FORGET IT!
        ADD     #776,2(R2)      ;SET COUNTER TO RIGHT VALUE
        BEQ     NOSTUFF         ;IF 0, THEN NOTHING TO OUTPUT
        MOV     2(R2),@DEVLNK(R3)       ;MAKE THE COUNT IN THE BUFFER
        CLR     2(R2)           ;ZERO THE WORLD
        CLR     (R2)
        BR      COUMM
;
;
NOSTUF: MOV     DEVLNK(R3),R0
        BEQ     SINK
        CLR     DEVLNK(R3)
        MOVB    DEVSIZ(R3),R1
        TRAP    RELEAS
SINK:   CLR     (R2)            ;ZERO POINTER
        CLR     2(R2)           ;ZERO COUNT
        MOV     #READF,R0
        RTS     PC
;PAGE
```

```
;
;
;
;
;
;
OUTFIL: MOV     R3,R0           ;SEND THE MESSAGE THAT WE HAMVE
        MOVB    #FILCMD,MSGTYP(R0)      ;PUT IN A GOOD TYPE
        MOV     #REPYS,@R0
        TRAP    SEND            ;SEND IT ON
        RTS     PC
;
;
COUMM:  JSR     PC,OUTFIL
;
;
INFIL:  MOV     R3,R0           ;NOW, RELEAS THE MSG WE HAVE
        BEQ     OKMSG
        MOVB    MSGSIZ(R0),R1
        TRAP    RELEAS
OKMSG:  MOV     #RECIVS,R0
        TRAP    RECIV           ;GET THE REPLY
        MOV     R0,R3
        MOV     DEVLNK(R0),R1   ;PUT THE LINK IN R1
        MOVB    DEVCMD(R0),R0
        RTS     PC
;
;
;
;
LOOLST:
```

APPENDIX B



Schematic for the command language system.



Schematic for the link dedicated system.

APPENDIX C
LINK DEDICATED VERSION

The link dedicated system is designed to provide facilities to make the PDP-11 appear like a TTY to the PDP-10. It also provides the mechanism to transport files both ways across the link. Input is from the TTY and is accumulated in an accumulation module (ACUM11). This module does all the echoing and handles control U, control O, rubout and line overflow. For more information on exactly what happens in each of these cases see the module itself (it should suffice to say that the result is approximately the same thing as would happen on the PDP-10). ACUM11 also provides another service, it has several output ports for the string, one to the link and thus the PDP-10 and one to the PDP-11's port interchange module (PIP11). It also has a port connected up to the link input accumulation module (the link input is accumulated into lines to provide more efficient buffering) and will send an altmode to this port if required. The purpose of this is to free any message that is stuck in the accumulation module because it wasn't terminated by a break character (ex. 'CONFIRM:' from a LOGOUT). The ports are changed by control characters:

↑A - set port out so the string goes to the PDP-10
↑B - set port out so the string goes to PIP11
↑D - send an altmode to link accumulation module

The PIP module, when initially loaded has the following symbolic port assignments (see PIP documentation for explaination):

0 - system initial load device. In SYSLDT => DECtape, SYSLDK => disk
1 - alternate device (one not used as system load. SYSLDT => disk, SYSLDK => DECtape
2 - binary accumulation module. Input device for shipping binary files over the link
3 - ASC11 accumulation module. Input device for shipping ASCII files over the link
4 - link output module. output device for shipping any file to the PDP-10
5 - unassigned

The current binacm module allows only absolute binary files to be shipped to the PDP-11. It should be noted that the commands for each machine must be typed individually.

EXAMPLES:

shipping a binary file to the PDP-11

```
     (1 => disk, <character> => control <character>)
↑A                    !switch ACUM11 to send to the link
↑C                    !stop anything running on the 10
.R PIP                !start PIP
*↑B1;0:PIP11-BIN<2;0:  !return to 11, give PIP11 command
```

```
↑ATTY:/I←DSK:PIP11.BIN !back to 10, tell PIP to output file
                        to TTY
*                       !PIP done
*                       !PIP11 done
```

shipping an ASCII file to the PDP-11

```
↑a
↑c
.R PIP
*↑B1;0:PIP11-P11←3;0:
↑ATTY:/I←DSK:PIP11.P11
*
*
```

shipping a file to the PDP-10

```
↑A
↑C
.R PIP
*DSK:PIP11.P11←TTY:/A
↑B4;0:←1;0:PIP11-P11
*
*
```

If an error occurs during the transfer of a file, one of two things will happen. If the binary accumulation module should stop too soon (caused by a premature start block) then the rest of the file will be fumped on the TTY. The best thing to do is type control O on the TTY and when the file is really finished try again. The other thing that can happen is that the binary module won't see the start block and thus continue waiting for more input. This is characterized by the fact that even after a long wait nothing happens. Of course the problem may be that the PDP10 has gone down, but for most purposes this is unlikely. In this case you either have to reload the system or transfer another file over the link an hope the module becomes unstuck.

Once a file is on the PDP-11's disk it is very easy to transfer it to a DECtape so that it may be loaded using 11UP.

EXAMPLES:

transfering files from disk to DECtape

```
      (1=> disk, 0 => DECtape)
*0;0:PIP11-BIN←1;0:PIP11-BIN

*/X←TTYIN-BIN,TTYOUT,ACUM11,ATOM
```

```
TTYIN   BIN
TTYOUT  BIN
ACUM11  BIN
ATOM    BIN

*
```

All the underscored parts are the print out of the PDP-11. The second command is an example of the advanced form of PIP11 and makes use of the fact that nothing is lost between commands.

APPENDIX D

```
MODULE  LOADER

FUNCTION: pdp-11 absolute binary loader. loads modules into core from
        a block oriented i/o input. for the format of an
        absolute binary file see the paper tape software
        loader manuel.

PARAMETERS: priority=   700
        stack size=     215
        dib size=       214
        number of ports= 4
        module size=    261 words

PORTS:  port    name    function
         0              command input port. the second and third data
                        words are assumed to contain the file name
                        that the module is in.
         1              reply port. when the module is loaded, the address
                        is returned in the first data word of the message.
                        if a error occured, the reply address is in the
                        i/o page or else 0. valid errors are:
                                value returned          error
                                0       checksum error occured
                                -1      no room in core for module
                                -2      module size troble(too many blocks)
                                -2      module size trouble(too few blocks)
                        all other error are i/o errors and the i/o
                        error number is returned. see dskcom for
                        these values.
         2              i/o output port. commands to the i/o module are
                        sent out this port.
         3              i/o reply port. for message format see dskcom.

CONNECTIONS:
        port    module
         0      dirman:2
         1      dirman:3
         2      i/o module :0
         3      i/o module :1

ASSEMBLY: symhed, dskcom
```

MODULE PTP11

FUNCTION: does character, string and block mode i/o with the paper tape punch. mode is determined by the message type in accordence with atommg and dskcom. if block mode i/o then any output oriented command are accepted and a reply is generated. an error occurrs if a read or read oriented command is given (such as read a block, read a directory) or if a directory oriented command is given (delete a file).

PARAMETERS: priority= 77771
stack size= 215
dib size= 214
number of ports= 2
module size= 74 words

EXTERNAL: paper tape punch data register (ppb)
paper tape punch status register (pps)
paper tape punch semaphore (hsp)

PORTS:

port	name	function
0		message input port. messages are received through this port. if the type is 0, it is assumed to be a single character type message (the data in the low byte of the message data area. if the type is positive, a string is assumed with the first byte of data being the character (byte) count. if the type is negative, a block oriented i/o is assumed and a reply is sent out port 1
1		reply output port for block oriented i/o

ASSEMBLY: symhed, dskcom

```
MODULE  PTREAD


FUNCTION: handle the input from the paper tape reader.  initiates the
        ptr for input and waits on the ptr input semaphore. output is
        single byte mode i/o.

PARAMETERS: priority=   17700
        stack size=     2↑5
        dib size=       2↑4
        number of ports=  1
        module size=    32 words

EXTERNAL: ptr status register (prs)
        ptr buffer register (prb)
        ptr input semaphore (hsr)

PORTS:  port    name    function
         0      inp     character output port.  character is in the low
                        byte of the message

ASSEMBLY: symhed
```

MODULE PTR11

FUNCTION: do block mode i/o on the paper tape reader. will accept any block mode command as given in "dskcom" but will return an error if an output is tried, of any sort i.e. a write, open for output, delete, etc. or if a directory is requested.

PARAMETERS: priority= 77771
stack size= 215
dib size= 214
number of ports= 2
module size= 84 words

EXTERNAL: paper tape status register (prs)
paper tape reader data register (prb)
paper tape input semaphore (hsr)

PORTS:

port	name	function
0		command input port. messages of the command format (as given in dskcom) are input through this port
1		data and command reply output port.

ASSEMBLY: symhed, dskcom

```
MODULE  SINK

FUNCTION: the message bit bucket.  all messages received are deleted,
        never to be seem again.

PARAMETERS: priority=   777
        stack size=     2↑5
        dib size=       2↑4
        number of ports= 1
        module size=    16 words

PORTS:  port    name    function
         0              message input port.  any format of message
                        is allowed.

ASSEMBLY: symhed
```

MODULE FLOOR

FUNCTION: to separate two groups of modules on the feasible list. this module does a busy wait loop. it looks for a message on the input port by repeated tests of empty. when a message is received a 'p' is done on the floor semaphore. when released from the semaphore, the input message is sent out and a wait for a new message is started.

PARAMETERS: priority= 400
stack size= 2↑5
dib size= 2↑4
number of ports= 2
module size= 32 words

EXTERNAL: floor semaphore (floor)

PORTS:	port	name	function
	0		message input port. the message may be any format.
	1		message output port. the output message is the same as the input message.

ASSEMBLY: symhed

MODULE LK1IN

FUNCTION: to do i/o with the link, input only. this module is used for a "fast response" link, one that requires a response within 200 to 800 micro-seconds. it requires a special interrupt routine that fetches the data byte into core before doing the "v" on the semaphore. the semaphore is used as the data counter, (i.e. a "v" is done for every data byte).

PARAMETERS: priority= 10887
stack size= 2↑5
dib size= 2↑4
number of ports= 1
module size=

EXTERNAL: link 1 input status register (lnkaos)
link 1 input semaphore (l1isem)
link 1 temporary data buffer (lnkbf)
link 1 tmp data buffer size (lnksz)

PORTS:

port	name	function
1	inp	data output port. data is in the low byte of the message

ASSEMBLY: symhed

```
MODULE  LK2IN


FUNCTION: to do I/O with the link, input only.  single
        character mode.  this module gets the data from the
        link data register and thus requires only the
        semaphore (see lk1in for a different kind of
        link input.

PARAMETERS: priority=   18087
        stack size=     2↑5
        dib size=       2↑4
        number of ports=  1
        module size=

EXTERNAL: link  2 input status register (lnkbos)
        link  2 input data register (lnkbob)
        link  2 input semaphore (l2isem)

PORTS:  port    name    function
         1      inp     data output port. data is in the low
                        byte of the message

ASSEMBLY: symhed
```

```
MODULE  LK1OUT


FUNCTION: module to do io with the link - output only.

PARAMETERS: priority=   18888
        stack size=     215
        dib size=       214
        number of ports=  2
        module size=    88 words

EXTERNAL: link  1 output command register (lnkaos)
        link  1 output data register (lnkaob)
        link  1 output semaphore (llosem)

PORTS:  port    name    function
         0              data input port.  if the type is zero then the
                        first and only the first byte of data is outputed
                        to the link.  if the type is positive then
                        the first byte of data is assumed to be a count with
                        the data following.  if the type is negative then it
                        the is assumed that the message is a file type
                        command (see dskcom for what these commands are).
                        if the command involves a directory or
                        a real file then an error condition is returned.  an
                        error condition is returned if the command has to do with
                         initiating a file.  otherwise the only command anything is
                        done with is the writf command which causes a buffer
                        to be outputed (the buffer address is the second data word
                        and the first word of the buffer is a byte count)
         1              file command reply port

ASSEMBLY: symhed, dskcom
```

```
MODULE  LK2OUT


FUNCTION: module to do io with the link - output only.

PARAMETERS: priority=   10000
        stack size=     275
        dib size=       274
        number of ports=  2
        module size=    88 words

EXTERNAL: link  2 output command register (lnkbos)
        link  2 output data regster (lnkbob)
        link  2 output semaphore (l2osem)

PORTS:  port    name    function
         0              data input port.  if the type is zero then the
                        first and only the first byte of data is outputed
                        to the link.  if the type is positive then
                        the first byte of data is assumed to be a count with
                        the data following.  if the type is negative then it
                        the is assumed that the message is a file type
                        command (see dskcom for what these commands are).
                        if the command involves a directory or
                        a real file then an error condition is returned.  an
                        error condition is returned if the command has to do with
                         initiating a file.  otherwise the only command anything is
                        done with is the writf command which causes a buffer
                        to be outputed (the buffer address is the second data word
                        and the first word of the buffer is a byte count)
         1              file command reply port

ASSEMBLY: symhed, dskcom
```

```
MODULE  MAXEL


FUNCTION: module to convert internal format messages into external format
        messages.  the two types of messages affected are binary numeric and
        radix 50 alphanumeric.  it converts these into ascii characters.
        (numeric is converted into octal)  all
        other messages are unaffected.  the types are given in the interface
        language file "atomsg".  leading zeros (on numeric)
        are deleted, as are trailing spaces on alphanumeric.

PARAMETERS: priority=   1001
        stack size=     2↑5
        dib size=       2↑4
        number of ports= 2
        module size=    165 words

PORTS:  port    name    function
         0              data input port.  the input has the relavent
                        data in the first word for numeric and first two
                        words for alphanumeric.
         1              data output port.  the first data byte is a character
                        count followed immediately by the characters

ASSEMBLY: symhed, atomsg
```

```
MODULE  CLI


FUNCTION: to implement a command language for the software lab.  the
        teletype format of the commands is:
                load command
        l [<sib name>][ '<' [<unit number> ':'][<filename>]'>']
                file name defaults to sib name, unit defaults to 0
                multiple loads are separated by a '='
                connect command
        c [ <dib name>][ ',' <sib name>][ '<' <unit number>
                ':' ][<file name> ) '>' ] [':' <port number !
                port name >] < '=' ! '←' ! cr>
                if the terminator is '=' then a new mailbox is gotten for the
                connection and a command string of the same format is
                expected after the '=' (note: this command could contain
                a new '=' or '←' so that the command can go on indefinately).
                if the terminator was a '←' the existing connection of the
                left side (if any) is used. again the same format of string
                is gotten for the connection.
                the connect command saves almost all data between
                commands. this allows a very strong default
                structure (i.e. only the port need usually
                be changed).  also, the internal defaults
                are:
                all names default to the dib name, if is typed
                (except the port name)
                the file name defaults to the sib
                name.
        dd [< dib list>]
                disply items about the dib(s) if no name
                is given, disply the list of possible dibs
                i.e. the names of all the dibs on the system
                dib list
        ds [<sib list>]
                disply characteristics of the sibs named. if no
                name is given, the names of all the sib
                on the system sib list are displied.
        r
                cause the floor to lift and the cli to hang itself up.
        these commands will have the described result only if all the
        connections shown are made.
        examples:
        c lexam:1←pip,pip<1:pip11>:0
        c maxel:0←:1
                these two commans connect the 0 th port of the pip
                incarnation of the pip module to the 1st port of lexam
                and the 1st port of pip to the 0th port of maxel.
                note the use of the defaults.
```

```
PARAMETERS: priority=   7000
        stack size=     216
        dib size=       216
        number of ports= 6
        module size=    512 words

EXTERNAL: system dib list (udibhd)
        floor semaphore (floor)

PORTS:  port    name    function
         0       go     command input port. lexemes of the commands
                        are accepted in this port. types are given in
                        atommg.  all messages are assumed to be in
                        internal format, i.e. binary numbers, radix 50
                        character names.
         1      ptr     message output port. a number corresponding
                        to a particular condition is outputed tthrough
                        this port.
         2      ok      directory manager request port.  requests
                        for the directory manager are output through this
                        port.
         3      dir     directory manager reply port.
         4              floor output message port. when the
                        floor is to be lifted, a message is sent out
                        this port.
         5              floor response port. when itis desired to release
                        the cli a message is sent to this port. i.e. after
                        the cli sends a message o the floor out port 4
                        it waits for a reply on this port.
         6              disply communication port. a message for displying
                        is sent out this port. the first data word contains
                        the second letter (in radix 50) of the command
                        and the third word contains the address of the
                        dib or sib (if any)
         7              disply reply port. after sending a message out
                        port 6 a reply is waited for on port 7

CONNECTIONS:
        port    module
         0      lexam:1, lexam:0=atom-1, atom:0=accum:2
         1      msgmod:0
         2      dirman:0
         3      dirman:1
         4      floor:0
         5      floor:1
         6      disply:0
         7      disply:1

ASSEMBLY: symhed, atommg
```

MODULE BUFLIN

FUNCTION: change buffers into lines. looks like a i/o device and connects to a block i/o device. accepts commands and returns either with a reply or a series of strings followed by a reply. terminators of strings are carrage return (possibly followed by a line feed, which is includedin the string), a line feed (possibly followed by a carrage return, which is included in the string) or an altmode, if the buffer becomes full, the string is sent on and a new string started.
********undebugged********

PARAMETERS: priority= 20000
stack size= 2↑7
dib size= 2↑4
number of ports= 4
module size= 220 words

PORTS:

port	name	function
0		command input port. the commands are those given for block i/o. only read-type, non directory command generate non-error returns. if the command is read, a buffer is read and then turned into strings (type field is positive) where a string has a byte count in the first data word ad the charactrs in the following bytes.
1		command reply port. aso the string output port
2		i/o module command port. commands for the i/o module are set out this port.
3		i/o reply port.

CONNECTIONS:

port	module
2	i/o module :0
3	i/o module :1

ASSEMBLY: symhed, dskcom

```
MODULE  PIP11


FUNCTION: tranfer files between two block oriented i/o devices. accepts
        commands from the tty (through several modules) and
        executes those commands. all commands shown are those that
        would be typed on the tty. the command format is:
        i<item>←<item list>
        <item list> ::= tem ! item ',' item list>
        <item> ::= < specific ! specific  item>
        mspecific> ::= <  symportnumber ! unitnumber ! filenaeme ! fileextension !
                switch >
        symportnumber ::=  symbolic port number ';'
        unitnumber ::= number of device unit ':'
        <filename> ::= file name in directory
        <fileextension> ::= '-' extension
        <swtch> ::= '/' < 'l' ! 'x' ! 'z' ! 'd' >
          the switchs have the following result (all switches must be
        on the left side to have any effect)
                l - list the directory (ies) of the input devices (right side)
                x - use the same file name as given for input
                z - zero the directory of the output device
                d - delete the files given as input
        a typical command looks like:
                0;1:/x←1;0:ttyin-bin,ttyout,accum,atom,iexam;cli
          (the defaults are the previous object used in that
           position - possibly from the previous command line)
                /l←0;1:,2:,1;0:,1:
                /d←0;0:tmp.bak,tmp1,tmp2
                1;1:input.pll←0;1:symhed.pll,atommg,cli,tail

PARAMETERS: priority=
        stack size=     216
        dib size=       215
        number of ports= 12
        module size=    688 word
```

```
PORTS:  port    name    function
         0              command input port. this port is used
                        to receive all commands. the format of the messages
                        is internal atom as given in atommg.
         1              command response port. the response of the commands
                        is output to this port. normaally just a '*' is
                        output but when a directory is listed, it goes through
                        this  port.
         2,3            symbolic port 0. commands for the 0th device are sent
                        through these 2 ports, (see dskcom for formats)
         4,5            symbolic port 1
         6,7            symbolic port 2
         10,11          symbolic port 3
         12,13          symbolic port 4
         14,15          symbolic port 5

CONNECTIONS:
        port    module
        0        lexam:1, lexam:0 to atom:1, atom:0 to accum:2
         1       maxel:0
        2,3     any i/o module
        4-15    any i/o modules

ASSEMBLY: symhed, atommg, dskcom
```

MODULE LINBUF

FUNCTION: accumulate single characters into buffers. this module
looks like an input i/o module (buffer side) and will
convert a single character i/o module into a buffer
one. the terminator on the input is the control z character,
(026 ascii decmal)

PARAMETERS: priority= 20000
stack size= 215
dib size= 214
number of ports= 3
module size= 178 words

PORTS:	port	name	function
	0		i/o command input port. a block oriented i/o command is accept through this port. for formats see dskcom.
	1		i/o reply port
	2		character input port. character is the first byte in the data part of the message

ASSEMBLY: symhed, dskcom

```
MODULE  MSGMOD

FUNCTION: take a numeric input and convert it into a message.
        the mmessages are (currently) a "." and a "?" (in single
        character format messages.

PARAMETERS: priority=   477
        stack size=     2↑5
        dib size=       2↑4
        number of ports= 2
        module size=    64 words

PORTS:  port    name    function
         0              command  input port. the first word of the message
                        is used to determine which of the messages to
                        output.
         1              message output port. the messages are in the
                        single character format ( chracter in the first data
                        byte).

CONNECTIONS
        port            modules
         0      cli:1
         1      ttyout:0

ASSEMBLY: symhed
```

MODULE ACUM10

FUNCTION: accumulate characters until a full line is in the buffer and then output a string. a line is delineated by 1 of 6 things, (1) a carrage return is the input character (possibly followed by a line feed, which would also be included in the line), (2) a line feed is the input character (possibly followed by a carrage return, which would also be included in the line), (3) an altmode, (4) an asterisk ("*"), (5) a period (".") or (6) a full buffer.

PARAMETERS: priority= 77770
stack size= 217
dib size= 214
number of ports= 2
module size= 188 words

PORTS:

port	name	function
0	inp	character input port. messages with characters in low data byte are received through this port
1	out	string output port. the accumulated strings are sent out this port. a byte count is in the firstdata byte with the characters following.

ASSEMBLY: symhed

```
MODULE  ACUM11


FUNCTION: accumulate characters into strings, echoing the individual
        characters. a string is comprised of characters followed
        by a terminator (line feed, carrage return or altmode).  if the
        terminator is a line feed, a carrage return is also echoed
        if a altmode, then a dollar sign ("$") is also echoed
        or if a carrage return then a line feed is also echoed
        if an attempt is made to input more than 80 characters
        in a line, all characters which would make the count
        exceed 80 are ignored and the bell on the tty is rung.
        in addition, accum inplements the special characters
        control u, rubout, control c and control o.  control u
        causes the entire current line to be ignored and accum
        to effectively restart with an empty buffer.  rubout causes
        the previous character to be lost, after echoing it between slashes
        (the first slash is typed  when the first rubout
        is detected, and the last slash isn't typed until something
        other than a rubout is typed. ex. "acdeedcbcde"
        the buffer now contains "abcde").  control
        c causes a "v" to be done on the floor semaphore.  this
        is used to release the floor when it is desired to stop
        the user modules from running.
        conrol o causes a bit to be set so that the ttyout module
        stops printing.  this bit is cleared by accum
        whenever a character is received.
        in addition, there are control characters which determine
        the output port.  control a sends the accumlated lines
        out the first output port, control b the second, and a
        control d causes an altmode to be sent out the third
        output port

PARAMETERS: priority=   18887
        stack size=     2↑7
        dlb size=       2↑4
        number of ports=  3
        module size=    280 words

EXTERNAL: floor semaphore (floor)
        tty status word (ttysts)
```

PORTS:

port	name	function
0		character input port, data is low byte of message.
1		character echo port, format is the same as port 0.
2		first string output port, first byte of message is character count, characters are in the consecutive bytes
3		second string output port, format same as port 2
4		third output port. only an altmode can be sent out this port. the format of the message is the same as those of ports 2 and 3

CONNECTIONS:

port		modules
0	ttyin	
1	ttyout	

ASSEMBLY: symhed

MODULE ACCUM

FUNCTION: accumulate characters into strings, echoing the individual characters. a string is comprised of characters followed by a terminator (line feed, carrage return or altmode). if the terminator is a line feed, a carrage return is also echoed if a altmode, then a dollar sign ("$") is also echoed or if a carrage return then a line feed is also echoed if an attempt is made to input more than 88 characters in a line, all characters which would make the count exceed 88 are ignored and the bell on the tty is rung. in addition, accum inplements the special characters control u, rubout, control c and control o. control u causes the entire current line to be ignored and accum to effectively restart with an empty buffer. rubout causes the previous character to be lost, after echoing it between slashes (the first slash is typed when the first rubout is detected, and the last slash isn't typed until something other than a rubout is typed. ex. "acdeedcbcde" the buffer now contains "abcde"). control c causes a "v" to be done on the floor semaphore. this is used to release the floor when it is desired to stop the user modules from running.
conrol o causes a bit to be set so that the ttyout module stops printing. this bit is cleared by accum whenever a character is received.

PARAMETERS: priority= 10007
stack size= 2↑7
dib size= 2↑4
number of ports= 3
module size= 288 words

EXTERNAL: floor semaphore (floor)
tty status register (ttysts)

PORTS:

port	name	function
0		character input port, data is low byte of message
1		character echo port, format is same as port 0
2		string output port, first byte of message is character count, characters are in the consecutive bytes

CONNECTIONS:

port		modules
0	ttyin	
1	ttyout	

ASSEMBLY: symhed

MODULE BINACM

FUNCTION: accumulate individual binary characters into an
absolute binary file (block oriented i/o). checks the
checksum of the file and eliminates unecessary characters. this
module acts just like a byte oriented loader that
puts the bytes into
buffers instead of core (retaining the control bytes).

PARAMETERS: priority= 28000
stack size= 215
dib size= 214
number of ports= 3
module size= words

PORTS:	port	name	function
	0		command input port. the input from this port is a block oriented i/o command message. the command generates a return (always) which is a error return if the command is either directory oriented or output oriented. (commands are gven in dskcom).
	1		command reply port. replies to the block oriented commands are sent out this port.
	2		binary byte input port. messages received through this port are assumed to contain one binary byte in the first byte of the data area.

ASSEMBLY: symhed, dskcom

MODULE ACUME

FUNCTION: accumulate characters until a full line is in the buffer and then output a string. a line is delineated by 1 of 4 things, (1) a carrage return is the input character (possibly followed by a line feed, which would also be included in the line), (2) a line feed is the input character (possibly followed by a carrage return, which would also be included in the line), (3) an altmode or (4) a full buffer.

PARAMETERS: priority= 77770
stack size= 2↑7
dib size= 2↑4
number of ports= 2
module size= 188 words

PORTS:

port	name	function
0	inp	character input port. messages with characters in low data byte are received through this port
1	out	string output port. the accumulated strings are sent out this port. a byte count is in the firstdata byte with the characters following.

ASSEMBLY: symhed

MODULE SPLIT

FUNCTION: take an input message and produce two copies of it.

PARAMETERS: priority= 777
stack size= 2↑5
dib size= 2↑4
number of ports= 3
module size= 28 words

PORTS:

port	name	function
0		message input port. input is accepted through this port.
1		first message output port. an exact copy of the input message is sent out this port
2		second message output port. and exact copy of the input message is sent out this port

ASSEMBLY: symhed

```
MODULE  DIRMAN


FUNCTION: to manage the sib list.  this module handles all requests
        to find, delete, or add modules to the running system.

PARAMETERS: priority=   700
        stack size=     215
        dib size=       214
        number of ports=  4
        module size=    600 words

PORTS:  port    name    function
         0      req     directory request port.  a message is received on
                        this port that tells the directory manager what to do
                        valid request codes are
                        0 - lodnxt / load next sib on tape into core and link it onto
                                siblist, and return its adress.  if the sib
                                is alreaady on the siblist the adr of the old
                                version will be returned
                        1 - fndsib / search list for specified sib and return its adress
                        2 - fndlod / search siblist for specified sibb and return its
                                address, if not found search tape for specified sib,
                                load it, link it on the siblist and return
                                its adress.
                        3 - delsib / delete specified sib from siblist
                        4 - delall / delete all unused sibs from siblist
        1       rep     reply port. the reply message normally just contains
                        the address of the required sib, or is positive.
                        if a error occurs the return code corresponds
                        to an address in the i/o page.
                        valid  error codes are:
                                  0  checksum error
                                 -1  no more core
                                 -2  too many  blocks in tape sib
                                 -3  not enough blocks in tape sib
                                 -4  sib not found
                                 -5  sib in use
                                 -6  invalid request
         2      lod     loader communication port.  messsae just contains
                        the filename of the module required.
         3      cnt     loader reply port.  first word is 0 if an error, otherwise
                        it contains the address of the loaded module.

CONNECTIONS:
        port        module
         2      loader:0
         3      loader:1

ASSEMBLY: symhed
```

```
MODULE  FILHAN


FUNCTION: this module handles references to files.  it
        works in terms of directories and forms theinterface
        between the user and the directory device controler.
        an atempt has been made to keep it device independent
        and along this line it makes no assumptions about
        the sizes of the directories or the positioning
        of the entries in the directory.  rather this information
        comes from the individual device handler.

PARAMETERS: priority=   777
        stack size=     215
        dib size=       214
        number of ports=  4
        module size=    378
PORTS:  port    name    function
         0              the format of the user input  is (specifying only
                        the data)

                        ********************
                        ↑device  ! command ↑  the command for the file handler
                        ↑********************↑  and the device number
                        ↑  data words      ↑  any required data
                        ↑******************↑

                        the required data varies with the command.
                        for the directory command a block number is needed
                        that tells which directory block is wanted
                        for the open and enter command a file name is
                        needed,and for the write command a buffer
                        address and size is needed
         1              the user gets back the address of the filled buffer
                        even if its his, if the transfer was sucessful
                        and a 0 if an error occured with the next word
                        telling the source of error.
         2              device controler output port. the format of the output
                        varies with the command. for more information see
                        a device controler (dtacon)
         3              device controler reply port.

CONNECTIONS:
        port    module : port number
         0      user module
         1      user module
         2      devcon:0 (device controler)
         3      devcon:1 (device controler)

ASSEMBLY: symhed, dskcom
```

MODULE DTACON

FUNCTION: dectape device controler. it takes input from the file handler
in terms of a directory entry. the various commands cause
this module to read the directory, write it , update
it, read or write a block in a file, automatically
keeping track of space on the dectape.

PARAMETERS: priority= 777
stack size= 2↑5
dib size= 2↑4
number of ports= 2
module size= 568 words

PORTS:

port	name	function
0		command input port. the format of the command various with the command. the simplest is the request for a directory block. this command message has the format of: ↑**********↑ ↑ unit ↑ comd ↑ ↑**********↑ ↑ block number ↑ ↑**********↑ the other formats and the replys are given in the interface language file - "dskcom"
1		reply
2		dectape handler command ouput port. the message that is sent out this port contains commands for the device handler. see "dtahan" for the format of the messages.
3		device handler reply port

CONNECTIONS:

port	module : port number
0	filhan:2
1	filhan:3
2	dtahan:0
3	dtahan:1

ASSEMBLY: symhed, dskcom

MODULE DTAHAN

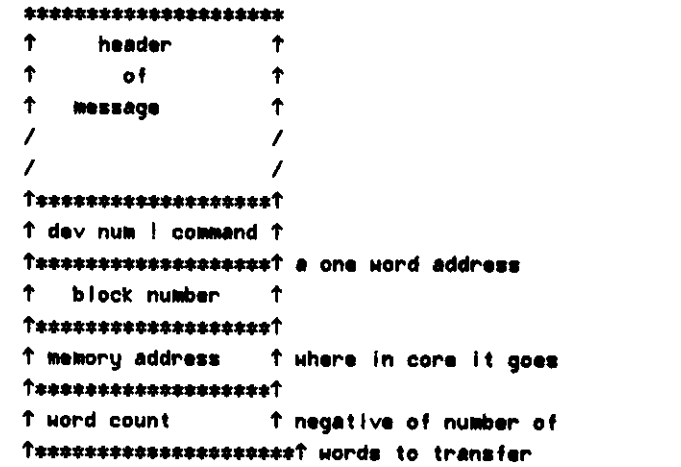
FUNCTION: module to handle the direct i/o with a dectape.
doesn't do any work on the data received, justs put it in
the dectape registers and then waits on the dectape i/o
semaphore. if an error occurs the operation will be repeated
before giving up.

PARAMETERS: priority= 10007
stack size= 2↑5
dib size= 2↑4
number of ports= 2
module size= 188 words

EXTERNAL: dectape command register (dtacmd)
dectape semaphore (decsem)

PORTS: port name function

0 command input port.the format of the
message in is:



the commands are: 1=> read, 2=> write
all others are errors

1 reply port. when finished, a mssage is replyed
that indicates the status of the requested operation
if the operation suceeded then the command byte is
set to zero, otherwise if an error the byte is
negative and the second data word has the following
meaning:

bit	meaning
15	error
14	parity error
13	mark track error
12	device is write locked
11	select error
10	block miss (a soft error)
9	data miss(bus was busy, soft error)
8	non-existant memory

CONNECTIONS:

port	module
0	dtacon:2
1	dtacon:3

ASSEMBLY: symhed, dskcom

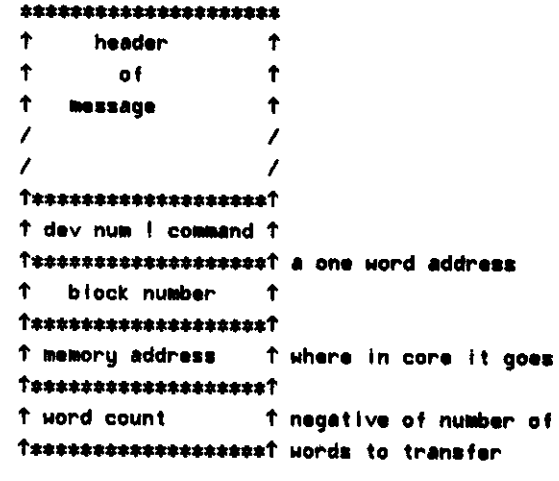
MODULE DSKHAN

FUNCTION: module to handle the direct i/o with a disk
doesn't do any work on the data received, justs put it in
the disk registers and then waits on the disk i/o
semaphore. if an error occurs the operation will be repeated
before giving up.

PARAMETERS: priority= 1887
stack size= 2↑5
dib size= 2↑4
number of ports= 2
module size= 185 words

EXTERNAL: disk semaphore (dsksem)

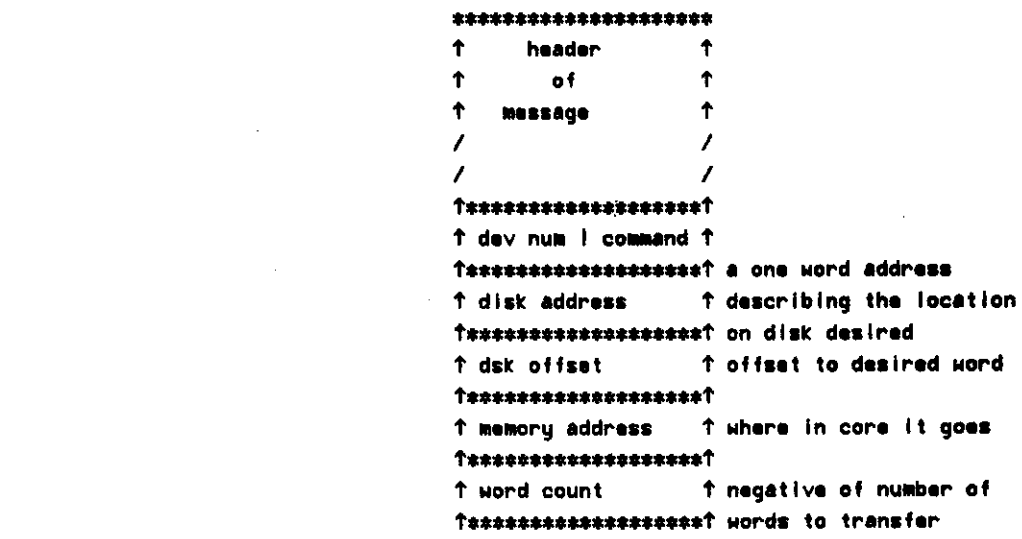
PORTS: port name function
0 command input port.the format of the
message in is:



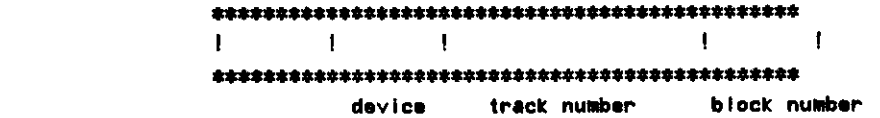
a one word address
where in core it goes
negative of number of
words to transfer

the commands are: 1=> read, 2=> write
all others are errors

to allow word accessing of the disk an alternate form of the
message is allowed. the difference is that
the commands are negative 1-3 with 3 bing the
write check.the alternate form of the message
is:



the format of the internal disk address is



1 reply port. when finished a message is replyed that indicates the status of the requested operation if the operation suceeded then the command byte is set to zero, otherwise if an error the byte is negative and the second data word has the following meaning:

bit	meaning
15	error
14	parity error
13	mark track error
12	device is write locked
11	select error
10	block miss (a soft error)
9	data miss(bus was busy, soft error)
8	non-existant memory

CONNECTIONS:

port	modules
0	dskcon:2 (or dtacon:2)
1	dskcon:3 (or dtacon:3)

ASSEMBLY: symhed, dskcom

MODULE DISPLY

FUNCTION: display relavent information about sibs and
dibs. for sibs this includes the names of all
incarnations and the port names. for dibs the
information is the parent sib name and the number
and name of all other dibs connected to each port

PARAMETERS: priority= 1000
stack size= 2↑5
dib size= 2↑4
number of ports= 3
module size= 300 words

PORTS:

port	name	function
0		command input port. the first data word is assumed to be a radix 50 command word. if the second character of the command is a "d" then the displaying is done for a dib, otherwise for a sib. the second word of the message is assumed to be the addrress of the object to be displayed. if the address is zero then the names of all the possible objects is displaed. (i.e. if the command was for dibs, then all the dib names are displayed)
1		reply port. when finished processing the command a reply is returned via this port. the format has no meaning.
2		display output port. all the information is oututed to this port for later processing. the format is internal, i.e. radix 50 names, etc.

CONNECTIONS:

port	module
0	cli
1	cli
2	maxel

ASSEMBLY: symhed, atomng

```
MODULE  ATOM

FUNCTION: produce atoms from strings.  an atom is defined as:
          <atom>::= <id>!<numeric>!<specialcharacter>!<breakcharacter>
          <id>::= <letter> ! <id> (<letter>!<number>)
          <letter>::=abcdefghijklmnopqrstuvwxyz
          <number>::=1234567890
          <numeric>::=<number>!<numeric><number>
          <breakcharacter>::= <cr>!<lf>!<altmode>!<last character>
          <specialcharacter>::=<otherwise>

PARAMETERS: priority=   777
        stack size=     2↑5
        dib size=       2↑4
        number of ports=  2
        module size=    288

PORTS:  port    name    function
         0              string input port.  low data byte is a character
                        count with the characters in the consecutinve bytes
         1              atom output port.  for numeric, id or breakcharacter
                        the low byte is a character count with characters in
                        consecutive bytes.  for special characters the low
                        data byte is the character.

CONNECTIONS:
        port            modules
         0      accumulation modules (accum,acum10,acum11,acume,!inbuf)

ASSEMBLY: symhed, atommg
```

```
MODULE  LEXAM


FUNCTION: convert atoms into internal format.  internal format
        depends on the type of the atom.  for special characters and
        break characters nothing is done.  alphanumeric are converted
        into radix 50 and numeric atoms (assumed to be in octal)
        are converted into binary numbers

PARAMETERS: priority=
        stack size=     2↑5
        dib size=       2↑4
        number of ports=  2
        module size=

PORTS:  port    name    function
         0              atom input port.
         1              lexeme output port

CONNECTIONS:
        port            module
         0      atom

ASSEMBLY: symhed, atommg
```

```
MODULE  TTYIN

FUNCTION: handle the input from the teletype.  initiates the tty for input
        and waits on the tty input semaphore.

PARAMETERS: priority=   17700
        stack size=     2↑5
        dib size=       2↑4
        number of ports=  1
        module size=    32 words

EXTERNAL: tty status register (t8ks)
        tty buffer register (t8kb)
        tty input semaphore (ttyrd)

PORTS:  port    name    function
          8     inp     character output port.  character is in the low
                        byte of the message

ASSEMBLY: symhed
```

MODULE TTYOUT

FUNCTION: handle output to the teletype. simulates the
necessary functions for form feed (8 line feeds), vertical
tab (4 line feeds), horizontal tab (tab stops every 8
spaces) and control o (stop printing until control o bit is
cleared.

PARAMETERS: priority= 10000
stack size= 2↑5
dib size= 2↑4
number of ports= 2
module size= 150 words

EXTERNAL: tty status register (t0ps)
tty output buffer register (t0pb)
tty output semaphore (ttywrt)
tty status word (ttysts)

PORTS:

port	name	function
0		input port. format is either a single character, string, of buffer. if single character, then the character is the low data byte. if a string, then the low data byte is a character count with the characters in the consecutinve bytes or if a buffer then the tty output looks like any other output only i/o device (see the file dskcom for particulars)
1		reply port if input was a buffer mode message

ASSEMBLY: symhed, dskcom

```
MODULE  OUTBIN


FUNCTION: to help calculate the radix 50 values of names by changing
        tthe type of radix50 messages to numeric (binary) and
        sending on two binary numbers (one for each three characters.

PARAMETERS: priority=
        stack size=     2↑5
        dib size=       2↑4
        number of ports= 2
        module size=    words

PORTS:  port    name    function
         0              message input port. only type of
                        ainb50 are affected by passing through
                        this module. (as given in atommg)
         1              message output port

CONNECTIONS:
        port    module
         1      maxel:0

ASSEMBLY: symhed, atommg
```

Bibliography

[1] Clark, W., "Macromodular Computer Systems," SJCC 67.

[2] Bell, G., et al., "The Design, Description and Use of DEC Register Transfer Modules (RTM)," Computer Science Department Report, Carnegie-Mellon University, Oct. 1971.

[3] Krutar, R., private communication related to his Ph.d. thesis, Carnegie-Mellon University, 1971.

[4] Jones, A. and Habermann, A. N., "Interprocess Communication Mechanism," Internal Memo, Computer Science Department, Carnegie-Mellon University, 1970

[5] Wulf, et al., "Bliss Reference Manual," Computer Science Department, Carnegie-Mellon University, revised April, 1971.

[6] Dijkstra, E., "Cooperating Sequential Processes," Technological University, Eindhoven, 1965.

[7] Wirth, N., "Program Development by Stepwise Refinement," CACM, Vol. 14, No. 4, (April, 1971).

[8] Bell, et. al., "C.mmp: The CMU Multiminiprocessor Computer," Department of Computer Science, Carnegie-Mellon University, August 1971.

[9] Denis, J. B., and Van Horn, E.C., "Programming Sematics for Multiprogrammed Computations," CACM 9, 3 (March 1966), 143-155.

[10] Dijkstra, E.W., "Cooperating Sequential Processes," Programming Languages, (F. Genuys, ed.), Academic Press (1968), 43-112.

[11] Dijkstra, E.W., "The Structure of THE MultiprogramminG System," CACM 11, 5 (May 1968), 341-346.

[12] Hansen, P.B., (ed.), RC4000 Software Multiprogramming System, A/S Regnecentralen, April 1969, Falkoner Alle 1, Copenhagen F. Denmark.

[13] Lampson, B.W., "Dynamic Protection Structures," Proc. AFIPS Conf. 35 (1969) FJCC.

[14] Jones, A.K., Private Communication, Carnegie-Mellon University, 1971.

[15] Parnas, D.L., "Information Distribution Aspects of Design Methodology," Special Report, Department of Computer Science, Carnegie-Mellon University)February 1971)

[16] Parnas, D.L., "A Technique for Software Module Specification with Examples," Special Report, Department of Computer Science, Carnegie-Mellon University (March 1971)

[17] Parnas, D.L., "On the Criteria to be Used in Decomposing Systemsinto Modules," Special Report, CMU-CS-71-101, Department of Computer Science, Carnegie-Mellon University (August 1971).

[18] Dijkstra, E., "A Constructive Approach to the Problem of Program Correctness," BIT 8 (1968).

[19] Wulf, et.al., "Bliss/11 Reference Manual," Department of Computer Science, Carnegie-Mellon University, 1971.

[20] Hansen, P.B., "Short-term Scheduling in Multiprogramming Systems," Third Symposium on Operating Systems Principles, October 1971.

Security Classification

DOCUMENT CONTROL DATA - R & D

(Security classification of title, body of abstract and indexing annotation must be entered when the overall report is classified)

1. ORIGINATING ACTIVITY (Corporate author) Computer Science Department Carnegie-Mellon University Pittsburgh, Pa. 15213	2a. REPORT SECURITY CLASSIFICATION UNCLASSIFIED 2b. GROUP

3. REPORT TITLE

SL230 - A SOFTWARE LABORATORY: INTERMEDIATE REPORT

4. DESCRIPTIVE NOTES (Type of report and inclusive dates)

Scientific Interim

5. AUTHOR(S) (First name, middle initial, last name)

W. Corwin, W. Wulf

6. REPORT DATE May, 1972	7a. TOTAL NO. OF PAGES 89	7b. NO. OF REFS 20
8a. CONTRACT OR GRANT NO. F44620-70-C-0107 b. PROJECT NO. 9769 c. 61102F d. 681304	9a. ORIGINATOR'S REPORT NUMBER(S) 9b. OTHER REPORT NO(S) (Any other numbers that may be assigned this report)	

10. DISTRIBUTION STATEMENT

Approved for public release; distribution unlimited.

11. SUPPLEMENTARY NOTES TECH OTHER	12. SPONSORING MILITARY ACTIVITY Air Force Office of Scientific Rsch (NM) 1400 Wilson Blvd. Arlington, Va. 22209

13. ABSTRACT

This report describes the resources and data structures of SL230 (Software Laboratory 230) and the designing of SL230 modules and systems. SL230 is a simple, multiprocess, operating system used to create an environment suitable for the construction of experimental programming systems for educational and research uses.

DD FORM 1 NOV 65 1473

Security Classification