The Second Edition of the Magic Six Commands Manual

Why is this happen? With half the systems memory gone, a flaky CPU and no way to load diagnostics, try editting online. Lucky for you the rest of this already out of date mag is XGP'ed.

Information on "Fox" terminals on MagicSix.

Foxes are computer terminals with a video character display. They can display 24 lines of up to 80 characters per line. Each character position can hold exactly one character, hence "over strikes" (such as underlined words) are not possible. The default typing conventions are listed below, some are user-setable, for information on changing them type "help stty".

On typeout, at most one screenfull (24 lines - a "page") will be typed since your last input. If the command you typed causes more than one page of typed output, the last line typed on the screen will be:
--more--
You now have two choices:
(1) type a space ("blank"), to allow the terminal to type the next page.
(2) type any other character (such as the first character of the next command you intend to type) to "swallow" the rest of the typed output, rather then printing it. Note that the system "ready" message will also be "swallowed".

On input, most key pushes result in the corresponding character being added to the right hand end of the "current line". Some control characters are used to edit the current line and reformat the display but are not normally inserted into the current line:

RETURN terminates current line (it is then sent to the command processor, or whichever program is reading input).
DEL this deletes the rightmost character on the line, it "untypes" the previous character. (Note that the key on the Fox is labeled up side down, lower case is DEL, upper case is "underscore").
ctrl X excises the current line (deletes to the leftmost char on the line).
ctrl D
ctrl R redisplays the current line (handy if you were typing in while the system was typing out).
ctrl L clears the screen then redisplays. Note: this is much better then using the CLEAR ALL key, which also clears tab settings. The system sets tabs every 5 characters at newproc time.
ctrl C clears all type ahead.
ctrl W silences the typout (all text which would have been typed is swallowed until the next input).
ctrl Q quotes the following character (used to insert special control chars into the buffer).
ctrl Z signals break immediately.

To type a control character, "ctrl Z" for example, hold down the CTRL key and hit the "Z" key.

In uc (upper case) mode all letters are converted to lower case unless they are preceded by a *#*.

Memory Management

segment

most computers have a 1-dimensional address space, a single integer can be used to specify any given memory location. The Interdata 7/32 provides a 2-dimensional address space, each memory location is referred to by a segment number (from 0 to 15) and an offset (from 0 to 65535). At any given time any number of the segments may exist (a segment may exist even if it is actually out on the disk as long as the system knows that it has been initiated). Each segment may have its own size and access mode. This means that some may be programs which are read and execute only, while others may be data segments of varying sizes. If you read beyond the end of a segment (e.g. load from 5|5049 when segment 5 is only 4096 bytes long) an oobounds condition will be raised. If you write beyond the end of a segment to which you have write access the segment will automatically be grown to the appropriate size (in multiples of 2048 bytes) unless the segments maximum length is too small. The current default maximum length is 60KB. If you attempt to write to a segment to which you do not have write access a no_write condition will be raised. Branching to a segment to which you do not have execute access will cause a no_execu condition to be raised. Referencing a non-existent segment will cause a nonexist fault.

address space

is the name given to the set of up to 16 segments which may referenced at a given instant. Since a typical user needs many more segments the dynamic linker is able to perform an operation called address space switching which changes the set of accessible segments and attempts to make the arguments to the subroutine accessible in the new address space. (see argument mapping)

Each segment has a preferred address space or is expected to run in the address space of its caller. Thus whenever tv is called it will be run in the address space called tv while whenever ioa is called it will be run in the current address space.

access (or acl)

Each segment in an address space has an access mode. This consists of a set of bits which indicate the

segment is readable (r), writable (w), or executable(e) as well as a number of other bits which are used to protect the system from the user. When a segment is initiated its access is set to the one stored in the file system and will stay as such until it is changed by use of hcs$set_acl.

reference name (ref_name)

Each segment in each address space may have any number of reference names as long as they are unique in the address space. The reference names may be listed with the lrn (list_ref_names) command and are used by the dynamic linker when it tries to see if a segment is already initiated.

initiate

To initiate a segment the system looks up that segment in the file system (given a directory name and an entry name) and makes that segment appear to be in the current (or specified) address space by making one of the previously free segment slots contain that segment.

before:

referencing 7|0
would cause a nonexist fault

hcs$initiate(">u>common","uncommon",seg_ptr,error_code)

after:

seg_ptr = 7|0
error_code = 0
referencing 7|0 is the same operation as referencing the first byte in the segment/file >u>common>uncommon

terminate

To terminate a segment is to dissassociate the file system segment from the current slot in the address space.

before:

referencing 7|0 is the same operation as referencing the first byte in the segment/file >u>common>uncommon

hcs$terminate(7|0)

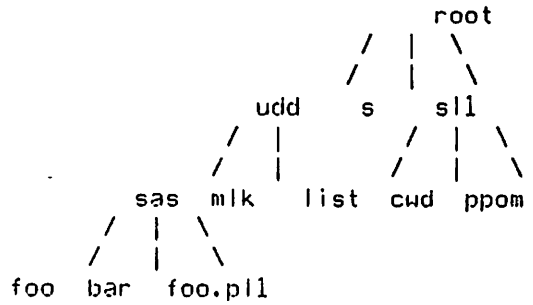
after:

referencing 7|0 gives a nonexist fault

File System

hierarchy

The file system can be thought of as a tree (an acyclic directed graph in which each node has a unique predecessor). At the top is a unique node known as the root. From this node it is possible to specify a path to any other node of the tree. Nodes which point at other nodes are called directories while those which do not are called segments (or files). Each branch leaving a given node may have any number of names.



pathname

A pathname is the name of a file given as a path to traverse through the file hierarchy. An absolute pathname is one which begins at the root. In the above diagram > refers to the root, >udd refers to the directory named udd immediately below the root. >udd>sas>foo is the name of the file foo in the directory sas in the directory udd under the root. Absolute pathnames begin with a > (pronounced "greater than" or "down") and start at the root. Relative pathnames start at the current working directory and may either indicate a list of branches to travel down (e.g. if the working directory is >udd>sas then foo.pl1 would refer to >udd>sas>foo.pl1) or may indicate that the specified node is reachable from the working directory only by first travelling up the tree. This is indicated by starting the pathname with a < or a series of <'s. (e.g. if the working directory is >udd>sas then <rk indicates the directory >udd>rk)

link

A link is a node in a directory which indicates that a branch is really located somewhere else in the file hierarchy. If a link in >sl1 name oink were created pointing to >udd>sas>bar then all references to >sl1>oink would be converted to references to >udd>sas>bar.

walking the hierarchy

Walking the hierarchy involves traversing the file system tree in such a manner as to visit every node below a given

one in the tree. Thus a complete tape dump must walk the entire file system from the root down.

starname or star convention

Sometimes it is useful to specify a set of file system entries which have certain sections of their names in common. For example, all programs ending with ".pl1" or all two component file names. To do this there is the star convention which allows commands which accept starnames to recognize the specified subset of all available files. A starname looks like any ordinary branch name except that it contains either a star (asterisk, *) or a question mark (?). In a star name a ? matches any letter. Thus, ?? would match all two letter names and mumble.???.frotz would match all names which start with "mumble.", end with "frotz." and have exactly three characters in the middle. Star matches any number of characters up to the next dot (period, .) so that *.pl1 would match all files which end with ".pl1" and contain no other dots. (e.g. Saturday_Night.pl1 would match while Sunday_Morning.incl.pl1 would not.) Star star (**) matches any sequence of letters, therefore ** matches everything, **.pl1 matches anything ending with ".pl1", *.**.pl1 matches everything with two or more dots ending with ".pl1".

I/O System

stream

An I/O stream is the basic unit of data transmission in Magic Six. It roughly corresponds to a logical unit on MAGIC 5, except that streams are given names rather than numbers and consist of a dim which does the I/O processing and a description which indicates to what device/stream the result is to be transmitted (or received from) and/or how such processing is to be performed. Streams may be created and destroyed within a process. The information about a given stream is stored in its iocb (I/O Control Block).

attach

A stream is attach by specifying the stream, a dim, and an attachment description which is interpreted by the dim. A stream must be attached before it can be used for I/O.

detach

The opposite of attaching a stream is detaching it which makes it unusable for I/O.

dim

This originally came from device interface module and is basically the program which actually does I/O to the given device.

A list of the most commonly appearing dims follows:

tty_io	interfaces to pasla devices and imlacs
syn_io	transmits data on to the target stream
mt_io	does magnetic tape I/O
file_io	does I/O to a segment (or set of segments)

Linking Subroutines

dynamic linking

In most systems the core locations of all subroutines must be known before a program may be allowed to run. This has many problems which should be obvious to anyone who has written a multiple overlay program on MAGIC 4. In a dynamically linked system subroutine references are resolved the first time the subroutine is called in a given process, meaning that the core location of a subroutine need not be known until it is actually called.

Calling is done by the means of a special instruction called a link. (Do not confuse this with the use of link in the file system sense, this usage is historical.) The first time a link is executed the dynamic linker finds the address of the subroutine and converts the link into a branch (or load immediate) instruction with the proper target address. This is called snapping the link. The call may then be repeated any number of times with no further intervention by the dynamic linker. (Since the link is obviously impure it resides in the so called linkage section or linkage segment along with the programs static storage.)

The dynamic linker accepts links of the form:

refname$epname

although the PL/I compiler and the assembler will convert external references of the form: refname to refname$refname. The work of the dynamic linker is done by a subroutine called scs$make_ptr. It first searches for a segment with the specified reference name and if it finds one decides if it needs to be moved into the current address space. Otherwise it searches through a list of directories called the search rules for a segment with the appropriate entry name.

If the link is of the form refname$ then it is assumed to refer to the base of the segment. If it is of the form rename$epname then it must be an object segment since it refers to the entry point named epname within it. Links of the form $epname are designed to prevent segment name clutter since the dynamic linker

first checks the calling segment for that entry point before trying to snap a link to epname$epname.

If the dynamic linker is unable to snap a link it will raise the condition linkage.

argument mapping

When a program calls a subprogram which runs in another address space the arguments which are perfectly accessible to the caller may be completely inaccessible to the subprogram since the corresponding segments may be initiated in one address space but not in another.

To get around this the system attempts to perform what is called argument mapping which makes all arguments accessible to the subprogram. Since pointers have values which are directly related to the address space they must be mapped specially. In particular, the pointer itself may be inaccessible but the data pointed to by it will be made accessible instead.

To allow a pointer to be passed verbatim, for example, when a pointer points to a data base which only the subprogram should look at then the subprogram should declare the pointer options(nomap)

ADDRESS SPACE DOCUMENTATION·

The commands which deal with address spaces are:

```
  lrn   Lists Reference Names of active segments (per address space).
  ls            List existing address Spaces.
  sn            sets the Space Name of a segment.
  in            Initiates a segment in the current address space.
  list  Lists address space for each segment (if it is not
                        null), its the next to the last thing on the line,
                        the last being the ring number.
  status        Gives stats on a segment, including address space.
  tmr   Terminates a segment from the current address space.
```

What an address space is:

Because of hardware limitations, at most 16 segments may be "mapped" or "in use". MagicSix provides a general facility for allowing the users to think they have as many active segments as they want. The operating system can manipulate entire sets of 16 segments called "address spaces", basically pushing and poping on a stack of pending address spaces, only the top one actually being active (hardware mapped). Each segment has a specified address space. When we link to a subroutine, a check of the address space of the segment (containing the subroutine) is made, if it is the same as the active space or it is null (indicating "I'll run anywhere") the segment is initiated in the current address space and linked to. If the space names do not match the new space is pushed and made active. Now when the segment is initiated it will be in the new address space and hence will not take up any segments in the caller's address space. When the subroutine returns the new address space is poped and the previous address space (the caller's) becomes active. Note that the address space is still "there" just not active.

sn:

This is the space_name command. It takes two args, a pathname and an address space name. The segment represented by the pathname is put in the address space named.

lrn:

This command has three forms. When called with no args, this command lists all the refnames on the segments in the current address space.

When called with one arg, and that is not a "*", then lrn lists the refnames on the segments in the address space named

by the argument. If the name is a "*", then all address spaces are listed and all the segments therein are listed with their refnames.

ls:

prints a list of all of the address spaces which have been entered since the last newproc (or login). "zs" removes spaces from this list.

The archive manager:

usage is: ac <one letter command> <archive> [<archive entry>]

The ac command has at least four sub commands
t - generate a table of contents of the specified archive
d - delete an entry from the specified archive
a - add an entry to the specified archive
x - copy an entry from then specified archive into a file of the same name

The asr command takes two args. The first is a full pathname of a dir to be added to the search rules, and the second is a number between 1 and f (hex) which is the position in the rules to make this dir. The first dir after initiated segments is 1, and the others are numbered successively.
The name -wd will specify the working directory.

The dsr command takes one argument and removes it from the search rules. It warns you if the dir is not in the search rules.

The psr command prints the search rules.

The bind command concatenates several object segments into a single segment. The names of the segments to be bound are in a bind control segment.

The usage is: "bind foo"

The bind file is foo.bind and the first line in it is the name of the output file. All succeeding lines of the bind file are interpreted as segments to be bound, comments or options specifications. The last line that will be read by the binder is "end;", this terminates the binding. Any line beginning with a ";" will be treated as a comment line.

Any line containing:

options:option1,option2

allow the following options to be specified:

map	generate a binder map of pc1 and pc7 offsets
table	generate a map and copy all statement tables into the target segment
notable	suppresses the copying of statement tables
nomap	suppresses the copying of statement tables and the generation of binder maps

call entryname arg1 arg2 ...

calls a specified subprogram with the specified arguments

entryname is of the form:
segment[$entrypoint][/space]

segment is the name of a segment to find in the search rules
entrypoint is an entry in that segment
space is the space in which to run the program

The args are of the form:

[-r | -return] [-type] [value]

-r or -return indicates that the value is to be typed out after the routine returns

-type may be:

-code this causes a fixed bin(31) number to be passed in and interpreted as an error code on return

-c[#] passes the next argument as a character string, if a number is specified then a char(#) variable is passed (the value is padded with blanks if necessary)

-cv[#] is like -c except that a char(#) varying string is passed

-b32 treats the next argument as a bit(32) hexadecimal number
-b16 treats the next argument as a bit(16) hexadecimal number
-byte treats the next argument as a bit(8) hexadecimal number

-fb31 treats the next argument as a fixed bin(31) decimal number
-fb15 treats the next argument as a fixed bin(15) decimal number
-fb

-p treats the next argument as a hexadecimal pointer

If no type is specified the argument itself is passed off as a character string. Thus: "call x y z" is identical to "x y z".

To pass a series of values packed into a structure surround them by -[and -]. All arguments in between those two brackets will be packed together bytewise and passed as a single structure type argument. Note that -r -[may not print out things right on return.

Examples:

```
call ioa ^a^a foo bar

call scs$make_ptr -cv32 scs -cv32 ioa -p 0 -r -p 0 -code

call ioa ^p -( -fb 7 -fb 7 -)
```

The CLEAN command takes no arguments. It goes through the current working dir, and for each file that it finds, it prints the file name and ask you "what about it"? If you answer anything starting with a "d" or a "D", the file will be deleted, otherwise it will not. In either case, CLEAN will then go to the next file in the dir. When there are no more files in the dir, it will return. If you hit quit while in clean, you will be back at the level you started at.

There are various conditions that can be raised while running programs on MagicSix. Here is a list of them, what they mean and what to do to fix them.

bad_luck

This is a very unique condition, the fact that it is raised has nothing to do with the correctness of your program. It indicates the system's inability to recover from a certain combination of unlucky happenstances. To wit: a segment fault occured in the middle of the wrong instruction, destroying needed information. Just try again.

no_excut

This means that the location that this condition was raised at is not in an executable segment. This means that the e access (when you type list) is off. Most compiled segments have this access set correctly, so if you get this error, it most likely means that you are trying to execute a data segment. If you want to correct the access of a segment, use the sa command (type help sa).

no_write

This means that the program attempted to store into a read-only segment. The location the condition is raised at is the location of the instruction which attempted the invalid store instruction. The referencing address is the place that would have been modified.

gate_err

This error occurs when the system is called incorrectly. This means that either an entry point that should not be used by users has been called, a random branch into segment 0 has been made, or the wrong number of arguments have been supplied to a system routine. The location that this was raised at will tell you which routine was called. Register 14 will likely point to the caller.

badxstat

This is raised when an external static variable is referenced, and resides in a different address space than the caller.

bad_args

This condition is raised when a call to a system routine would cause the system to attempt to modify a read-only segment. It is raised at the point of entry into the system that detects the error.

area

This condition is raised when an allocate statement or call to allocn fails because of insufficient free space in the given area.

unimplop

This means that an un-implemented pl1 operator has been used. Notify MLK (Mike Kazar) and he will write it.

badnargs
You called a pl1 program with the wrong number of arguments.

oobounds
This condition is raisd when you attempt to either grown a segment past its maximum length, usually >60K, or try to reference past the end of a read-only segment. The latter is the most likely reason.

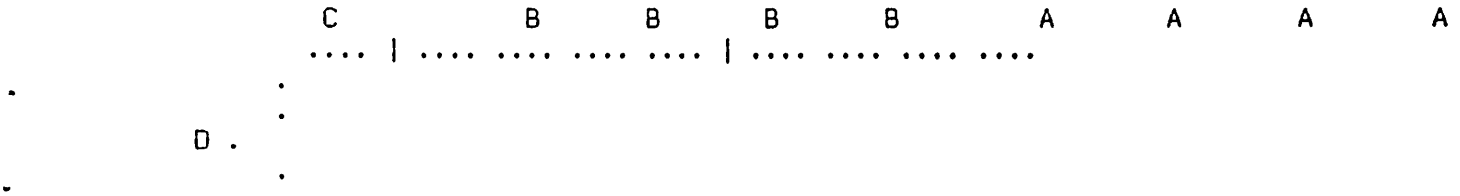
recurse
This condition is raised when you grow your stack too far, usually by recursing too deeply. You can type a few commands, but if you are not lucky, then you will get a fatal error.

nonexist
This condition is raised when you reference or branch into a segment that does not exist, i.e. that is not mapped over a file.

SVC0
This conditon means that the dynamic linker was unable to find a segment named in a link, or the entrypoint in the segment.

EVERYTHING YOU ALWAYS WANTED TO KNOW ABOUT THE LIGHTS BUT WERE AFRAID TO ASK

The information displayed in the front panel lights by MagicSix is extremely useful for debugging MagicSix crashes and other less serious problems. The four different sections are described below.



The Right Most Halfword (AAAA):

This displays the TCB address of the current user. There are two classes of TCB's. The first set are the TCB's for the two system processes. They both have zero as their upper digit (ie "0CE0" or "0908"). The other class of TCB's are those of users. These start at "D0C0" and can range into the "Fxxx" area.

All of these numbers are multiples of 8 hex which means that the low order three bits of this section should NEVER be lit. The usual cause of their being lit is some sort of irrecoverable disk error. To fix this problem hit SGL to stop the processor and turn the disk off then on. Restart the processor by hitting RUN.

Since all the system processes and only the system processes have zero as their first digit "thrash mode" is easy to detect. If very little time is spent with any of the first digit lit then no one is getting much work done: the system is thrashing. Warning: the system always spends a certain amount of time thrashing even with only one user and under normal circumstances. For instance the PL/1 compiler requires very little CPU time and lots and lots of memory. Therefore a PL/1 compilation with only one user LOOKS a lot like thrashing. A good test to see if you are hopelessly thrashing is for all users to hit break and try to go idle. if one or more user fails to respond with QUIT followed by a ready message within several minutes (give it a little time) then you are really thrashing. If one or more users gets to command level have them type "hcs$logout". If all idle users have typed this AND their consoles have stoped echoing then you have really had it. The system must be re-IPLed.

The Middle Halfword (BBBB):

This half word consists of a single bit which moves back and

forth. It advances one bit position every time MagicSix passes through the scheduler. That is when ever MagicSix attempts to find another user to run.
If no users are runable these lights do not move and the wait light above the key switch is on. This is "idle mode".
If the only load on the system is reading characters the lights are moving in fits and starts. This occurs when all the users are typing at command level or the user with the pinwheel it typeing and the other users are being held at bay so that the pinwheeler can get some works done.
Basically: one or more users are getting work done and the work is type in.
When a process is doing a fair amount of thrashing it will send many messages to the CORE_JOB (the system process with the smaller TCB address). The CORE_JOB often send messages to SYS_INIT (which is the disk server) when it needs something brought in from disk. This means that the processor will be switching processes quite often and the lights will be moving back and forth very quickly.
Simultaneously the TCB address in the AAAA slot will also be flashing vigorously. This is the normal mode for the PL/1 compiler.
Any program that loops or does prolonged crunching will be evident because the lights will be moving steadily and magesticly. One TCB will be in the AAAA slot. Assembling large programs with midi will make the lights act like this.
When this is happening the process is not volentarily giving up control of the processor but has it taken away when the clock interrupt happens. It happens 10 times per second. In this mode the dot should take about 3 seconds to complete one cycle.
The final state the lights can be in is characterized by the lights blazing. The dot is moving back and forth so fast that the single dot is blurred into a line. One TCB address is being displayed, usually a system process. This means that the process is waiting for some one else to do something and no one is. A very typical example is for a lock to get left locked somehow. If the process waiting on the lock is not a system process the a directory is probably locked and another user can try and unlock the offending dir (It is usually >pd that is locked so try it first followed by the users home dir). If there are no other users logged in hit ^Z on a free console and login in as foo to try and unlock things. Do NOT login as a regular user.

The Left Hand Corner (C):
This section is for hacks. It should be doing something.

The Left Most Edge (D):
These four bits are ALWAYS moving. If they aren't the system

is deader than a door nail. PUNT.

Documentation for copy (cp):

This is the copy command. It takes two arguments. The first is the pathname for the source file. The second is the pathname for the destination file. If only one file name is given, then it is the name of source file, and the destination file name is formed by taking the entry name of the source file and putting it in the current working directory. This command copies using the bit count field of the source segment, unless it is zero, in which case the number of characters to be copied is determined from the page count of the source file.

Here are a list of the codes flashed in the lights when the system crashes, and what they mean. They are due to svc7's being issued by the system.

1: Changed processes with the inhibit flag on.

2: Read_ipc got a spurious wakeup. Normally happens if you power up a disk with the machine running rather than halted. This is obsolete and should never occur.

3: Core job tried to grow a segment past sixty-four k.

4: Core job tried to wakeup deleted user.

5: Too many wired pages (No paddle).

6: Tried to read a disk record into segment 0.

7: Common heap free chain smashed.

8: Not enough free storage to satisfy request.

9: Smashed linkage offset table.

A: Checksum error on system loading.

B: Tried to swap out into vtoces 0, 1 or 2.

C: Tried to write out into pages 0 or 1.

D: Tried to load a file > memory hole size.

E: Power failure while executing critical code.

F: Fatal error in system process.

```
create path1 [path2 ...]
cr

creates the segments specified by the pathnames
e.g.
create foo      would create a segment named foo in the current
                        working directory

cr >u>eric>rotch>emergency_lossage
                would create a segment named emergency_lossage
                        in the directory >u>eric>rotch
```

(Note that the directory must exist for anything to be created in it.)

```
create_dir path1 [path2 ...]
cd

creates a directory with the specified pathname


link pathname | name1 target1 [name2 target2 ...]
lk

will create a link which will either:

        point to the specified pathname and have a name which
                is the entry name of the pathname specified

or

        have the name specified and point to the pathname
                indicated by the target pathname
```

pswː
cwd:
This command takes one argument, the pathname of a directory, and makes it the current working directory. It remembers the previous four working directories also, on a stack, for the ppw and tws commands.

wd:
pwd:
This command prints the name of the working directory.

tws:
This command prints out the entire working directory stack. Note that only the top directory of the working dir stack is the actual working dir, the others are completely ignored by the system.

ppw:
This is the pop_wdir command. It pops one dir off of the working dir stack and makes the new top dir the current working directory.

There are four commands for deleting file system entries (dirs, links, and segs).

They are
1) delete or dl - deletes only segments
2) unlink or ul - deletes only links
3) delete_dir or dd - deletes only dirs
4) delete_entry or de - deletes any entry

The first takes one argument which is the star name of the entries to be deleted. The last three take any number of arguments which are the star names of the entries to be deleted. The first three only attempt to delete entries of the right type. Hence, if you have a dir named "foo" and say "delete foo", you will get an error messsage saying that no entries matched the star name as opposed to an error message saying that the entry was of the wrong type. In general the first three commands are meant for general use while the last is meant for masochists and really cleaning up a dir.

There is also the "clean" command to kill garbage segments. Type help clean for more info.

Comments / Complaints on these commands should be sent to RK.

do is a command line program which implements the general lambda form for command lines.

```
do pattern arg1 arg2 ...
```

The pattern is a string which is passed to the command processor after all parameters in it have been substituted for. A parameter is indicated by an ampersand, "&". The string "&&" is replaced by a single ampersand, otherwise when an ampersand is encountered the next few characters are scanned. These must be "r", "q" or a digit from "0" to "9". do scans up to the number and then performs the substitution.

&n	causes the n-th argument to be substituted.
&qn	causes the n-th argument to be substituted, but also requoted. This means that the string is surrounded by quotes and all quotes within it are doubled so that when do passes the line to the command processor, the result is one argument.
&rn	causes the n-th argument on to be substituted, each argument separated by a single space. This effectively scoops up the remaining arguments.
&qrn	acts like &rn except that it also requotes the result like &qn.

To see the result of a do expansion, use the entry point do$debug which acts just like do except prints the result as opposed to passing it to scs$cl. The expanded line must not exceed 256 characters in length.

Examples:

```
do "who"        -->>     who
do "pll &1 -nd" foo      -->>     pll foo -nd
do "send rk &r1" this is a message     -->>
        send rk this is a message
do "send &1 &qr2" sas this loses     -->>
        send sas "this loses"
```

exec_com: The command file executing routine.

Command Format:

exec_com filename [substitution string #1 [, sub. string #2 [,...]]]

< where 'ec' may be used instead of 'exec_com' >

The purpose of a command file is to allow a user to perform a defined set of tasks repeatedly with a minimum amount of effort. Exec_com is a routine designed to execute such a file.

Basically, exec_com allows string substitution, and flow control (by means of 'if' - 'then' - 'else', and 'goto' statements). The substitution string capability permits the user to substitute an argument in the command line for a string in the command file. With the use of the if - then - else constructs and the goto statement control can be shifted from one part of the command file to another.

All exec_com commands are preceded by an '&'.

The following is a list of the possible exec_com commands:

Argument Substitution:

'&&'	replaced by '&'.
'&ec_name'	replaced by the entry name of command file.
'&n'	replaced by the number of aruments to the exec_com command.
'&0"	replaced by the directory portion of the command file pathname.
'&i' (i is any integer)	replaced by the 'i'th substitution argument in the exec_com command.
'&fi' (i is any integer)	replaced by the 'i'th through the last substitution arguments

Flow Control Statements:

'&goto' <label_name>	transfers control to statements following the occurance of the label (see &label).
'&label' <label_name>	location to where control is transferred as result of a goto. <label_name> is a character string of 32 or less characters.
'&quit'	stops execution at this point and returns to user. this is an optional command; it is assumed at the end of the command file.
'&if' [ACTIVE FUNCTION]	the active function must be enclosed by square brackets, and is any function that is defined to

		the system or user. the square-bracketed expression is fed to the command-line-processor which returns either 'true' or 'false' (assuming the active functior is defined), control is then issued to the '&then', or '&else' statement depending on the returned value. All '&if' statements must have a matching '&endif' statements. there may be nothing else on the line following the square-bracket expression.
'&then'	control	is transferred to here if the square bracket expression of the corresponding '&if' is true.
'&else'	control	is transferred to this point if the square bracket expression of the corresponding '&if' is false.
'&endif'		closes a '&if' block. there must be a one-to-one correspondence between '&if' and '&endif' statements.

Miscellaneous Statements:

'&command_line_on'	causes subsequent command lines to be printed out to the terminal before they are executed.
'&command_line_off'	inhibits printing of command lines before execution.
'&ready_on'	invokes the calling of the user's ready procedure after each command is executed.
'&ready_off'	turns off the previous.
'&print'	prints rest of line to terminal. line is not executed.
'& '	rest of line is ignored. intended for commenting lines

If any of this seems unclear look in the Multics Programmers' Manual Reference Guide for 'exec_com'. The major difference between this version of 'exec_com' and the Multics version is that this version allows multiple line, and nested (to level 16), if-then-else constructs.

etc is the error table compiler.

```
etc error_table
```

converts a source error file error_table.et into an object error table in the current working directory.

The first line of the error table is the name to be given to the generated error table.

The format of a source error file is a series of lines which may be either comments (if the first character is an asterisk) or error designations, which are of the form:

error number, short name, long message

where:

error number is the internal error code number (negative if a serious error, positive if a minor note (e.g. segment already known).

short name is the short name of the error message. etc puts the error number at an entry point with the short name indicated.

long message is the message which will be printed out by com_error for the error in question.

For example:

-37,fuckyou,Fuck you, you mangey scumsucker.

would generate an error entry for error -37.
error_table$fuckyou would indicate a fixed bin(31) value of -37.
call com_error(-37,"eatdogs"); would print
eatdogs: Fuck you, you mangey scumsucker.

External data is normally stored in segment 14 by the system. This is the linkage segment and is known by the name "linkage". External data items may be set up by calling scs$make_static_external. The following is a list of the system defined external data items. They may be referenced as linkage$<name>.

variable	value
level	the current command processor level
ready_on	whether or not ready messages on on
accept_messages	whether or not messages are being accepted
long_error	whether or not long error messages are desired
print_com_error_messages	whether or not com_error should print the message this flag is reset by each call to com_error a is used by com_error$shut_up which on calls to suppress the printing of the message if you wa to do something else
user_io	iocb ptr for user_io stream
user_input	iocb ptr for user_input stream
user_output	iocb ptr for user_output stream
listing_output	iocb ptr for listing_output stream
error_output	iocb ptr for error_output stream

Fido is a general purpose watchdog program. It takes the following args:

-num	means watch for a change in the number of users.
-user foo	means watch for user foo to login.
-watch nnnn	means watch hexadecimal location nnnn for changes (1 word).
-off	turns fido off.
-on	turns fido back on.
-time n	tells fido to wakeup every n seconds.
-index n	tells fido to watch for vtoc index n to be used by anyone.
-entry n	tells fido to watch for the segment n to be initiated.
-sum	fido will summarize what it is waiting to see.
-clear n	clear the nth thing that fido is watching for.
-call foo	tells fido to call foo every time it wakes up.
-cl foo	like -call.
-free n	tells fido to watch for the sytem free storage to cross n.
-spy	tells fido to watch for spies.

Fido's are cleared across processes.

Document for FINDALL:

findall is used to search for a particular character string in a set of segments. Usage is:

```
findall <the-string-to-be-searched-for>
                                    [ <starname-of-files-to-be-searched> ]  [-s]
```

The starname defaults to "**", and can be either absolute or relative. The default is "long format", but any third argument will cause it to use "short format". In long format, each line containing the string is printed. In short format only the number of occurrences is printed.

Examples of use:
To find all labels in foo.pl1:

```
findall  :  foo.pl1
```

To find all of the calls to subroutine "zap" in all of the pl1 programs in your working directory:

```
findall  zap  *.pl1
```

To find all (or at least most) of the setq's to the atom quack in all of your lisp files:

```
findall  "(setq quack"  *.lisp
```

To count the number of statements in each of the pl1 programs in the system source directory:

```
findall  ";"  >s>*.pl1  -s
```

5/30/77

forall is a program which allows one to perform an operation on a set of segments, links, or directories. The general format of an invocation of forall is as follows:

```
forall starname options command
```

where the starname (optional if the first argument begins with a "-" in which case it becomes **) is a standard star/pathname such as <***.PL1

options are keywords which begin with a "-" and are as follows:

-BF	brief - don't print the command string
-S	segments - only perform the operation on segments
-L	links - only perform the operation on links
-D	dirs - only perform the operation on dirs
-WK	after you finish a given directory recursively do everything to all of the sub-directories
-P	causes substitution to use the full path name vs. a relative pathname

If no options are specified -S is assumed.

The commands is an argument or a series of arguments which are catenated together (separated by spaces) to form a string. If the string contains a ^c it will be replaced by the entry name or full pathname (see -P) of the entry being operated on. "^c" may appear up to 4 times in the command string. If "^c" does not appear in the command string it is catenated on at the end.

Examples:

```
FORALL **.PL1 ST
does a status on all PL1 programs in the current dir

FORALL * -BF COPY ^c >SL1>^c
copies all single component segments in the wdir into >SL1
without printing out messages

FORALL >***.PL1 -P -WK ASCIIP
checks all PL1 programs on the system (note the -WK)
and sees if they are ASCII files
```

help.pl1

** Novices to Magicsix should try "help intro1" and "help intro2".

The help program looks up and prints the documentation on a program or subroutine. "help tv" will print out the documentation on the tv command, for instance. Help * will print out all the topics on which help is available.

Two other programs also make use of the help_file: "whatis" which gives a short explaination of the program, and "whoseis" which looks up the maintainer of the program.

The data base in which the help (and whatis and whoseis) program looks is kept in >doc>help_file. The format of each entry is as follows.

[<name of program> <pathname of documentation> <maintainer(s)> / <oneliner>]

The pathname can be one of three things.

1) a real pathname.
2) the word "none" meaning that the oneliner is the only available documentation and that it should be printed out.
3) the word "doc" meaning use ">doc>name_of_program.doc" as the pathname.

Since many programs have alternate names the number of help_file entries can be minimized by the use of the assoc_table. This is a file containing a list of associations to be made between what topic help is needed on and where the help can be found. The format of the assoc_table, which is in >doc>assoc_table, is very simple:

[<name> <name of help entry>]

```
ieh scan
```

or

```
ieh load dname[:ename] ...
```

These programs provide a facility for scanning and loading Magic 4 or 5 standard IEH DUMP tapes. To set up the tape stream type:

```
io attach tape si mt_io tape
```

ieh load takes arguments like those used on Magic 4 except that the Magic Six star convention is used (thus **.pl1 will work). All letters in requests are upper cased but the file names are lower cased when they are loaded. This is probably the most convenient transformation. The Magic 4/5 directories are turned into subdirectories of the current working directory.

Example:

```
ieh load zing_*:**.pl1 zing_*:* zing_*:*.sysin
```

might create the subdirectories zing_a and zing_b in the current working directory and load the appropriate files into them.

The IEHLIST command does a cheap version of the Magic 4 IEHLIST. It prints out the name of every file in the file system, by recursing from the root. Files are indented if they are contained in a subdir.

Inquire prints out useful information about people known by the system.
It optionally takes one arg which is the user id to lookup (default is you).
If additional arguments are present these are interpreted as special fields
which are individually looked up.

This information is kept in >doc>people>foo.doc where foo is a login
id. There are only a few requirements as to the format of the file.
1) Each topic is delimited both before and after with a double CR.
2) Each topic is headed by its name followed by ": ".

For an example see >doc>people>example.doc. Currently inquire knows
about the fields name,nick_name, phone, address, last_logout and hacking.
Good luck.

The io command allows a number of io system requests to be performed at command level. These include attaching, detaching, and many I/O and order calls. The basic syntax is:

```
io request stream options
```

The various forms are as follows:

```
io attach stream mode attach_string

io detach stream

io order stream request

io get stream nbytes

io put stream nbytes
```

get requests input from the terminal while put sends ascii to the terminal.

In addition to segments and directories, there are file system objects called links that indicate that a file or directory in a directory is not there, but can be found in another directory. Thus if there was a link from ">a>b>c" to ">e>f", then a reference to the file ">a>b>c" would result in the file ">e>f" being referenced instead.

link:
lk:
This command creates file system links. It has two completely different formats. The first is
link <pathname1> <pathname2>
which creates a link at the first pathname that when referenced yields the segment, directory or link at the second pathname. Thus
link >u>mlk>a >u>rk>o>plums
would create a link named "a" in the dir ">u>mlk" that refered to the segment plums in the dir >u>rk>o. The second format is
link <full path name of target>
e.g.
link >u>rk>o>plums
which would create a link, in the current working dir, named plums that refers to >u>rk>o>plums. Thus if the current working dir were >u>mlk, a this would have been equivalent to the longer command
link >u>mlk>plums >u>rk>o>plums.

ul:
unlink:
This command deletes a link. The object of the link is not touched. The command format is the same as for the dl and dd commands.

(cwr 1

MagicSix L I S P

(type "help lisp-int" for internal documentation)

LISP is a programming language for symbolic manipulation, it is a very versatile and elegant language, having its roots in the Lambda Calculus of Alfonso Church. The name LISP is a contraction of LISt Processing, a "list" is the most common data structure in lisp. LISP programs (functions) are usually interpreted, at least until the functions are completely debugged. But lisp functions can also be compiled for increased execution speed and efficent use of memory.

This online document is not intended to be an instruction manual on lisp programming (several of these exist off line) but this will serve to sumerize the available features and compare MagicSix lisp with its ancestors, MacLisp and Lisp Machine Lisp.

Symbols for lisp constructs used in this document:

<expression> a lisp symbolic expression,
  either an <atom>
  or a <list>.

<atom> a terminal in the tree structure,
  either a <symbol>,
  an <number>,
  or a <array>.

<list> a tree structure,
  either an empty list (called "nil"): ()
  or a list of expressions: (<x1> <x2> ...)

<number> a numeric quantity,
  either a <fixnum>,
  or a <flonum>.

<n> a fixnum, an integer

<f> a flonum, a floating point number

<array> an array,
  a special type of atom allowing indexed access to data,
  can have any number of dimensions,
  each element of the array can have 8, 16, or 32 bits.

<x>, <x1>, <x2> ... <y>, ...
  any <expression>

==========
Alphabetical listing of supplied functions:
==========

addl (addl <n>.) => n+1
1+

addr

and

apply

ar (ar <array> <dim1> <dim2> ...) => <element-value>

arrayp

as (as <value> <array> <dim1> <dim2> ...) => <value>
Array Store function, sets the indicated element
of the array <array> to <value>.

atom (atom <x>) => t if <x> is an atom
=> nil otherwise

append (append <x1> <x2> <x3> ...)

assq (assq <x> <y>)

autoload (autoload <symbol> <char-str>)
Declares <symbol> to be an AUTOmatically LOADed function.
The first time the function <symbol> is evaluated the
MagicSix segment named <char-str> is read in by "load",
then the expression is re-evaluated.

baktrace (baktrace) traces all pending functions on stack after error,
printing their names, starting with most recen

(baktrace t) trace includes calling form in addition to name.

boundp (boundp <symbol>) => t if <symbol> has a value bound to it
nil otherwise

catch

catenate

car

cdr

close

cond

cons

cons-string (cons-string <addr> <length>)
cons_string

cos (cos <number>) => trigonometric cosine of an angle of <number> radians

Ctol

declare

defun

delq (delq <x> <y>) two arg version only, deletes all elements of <y> which are "eq" to <x>

difference

do

eq
=

equal

eval

fboundp

fixp

floatp

fmakunbound

fset

fsymeval

gc

gc_cleanup

gc_setup

gensym

get

getpname

get_ascii
   get-ascii

go

grind

greaterp

intern

ItoC

lessp

list

load

load2

make_array (make-array <bit-size-of-elements> <dim1> <dim2> ...)
   make-array

make_array_header (make_array_header <bit-size> <address> <dim1> ...)

makoblist

makunbound

mapcar

max

memq (memq <x> <y>)

min

minus
   -

nconc (nconc <x> <y> ...)

not

nreverse

null

open

or

plist

plus
+

print

prog

progn

putprop

quit (quit) Does an "(release all)", and exits to MagicSix.

quotient
/

read (read) => one <expression> read from console.
(read <file>) => one <expression> read from <file>.

release (release) Used after an error to unwind stack to where you were before evaling the function that got the error, the previous top-level.

remainder
\

remprop

rep (rep) does a read-eval-print operation

rep_loop (rep_loop) repeatedly does (rep)

return

rplaca

rplacd

set

setq

set_gc_flag

sin (sin <number>) => trigonometric sine of an angle of <number> radians

substr

stringlength
   string-length
   string_length

stringp

subl
   1-

symbolp

symeval

terpri

throw

top-level (top-level) By default this is the same as the function "rep":

```
(defun top-level
       ()
       (print (eval (read))))
```

But it may be redefined to do whatever one wants done every top level evaluation.

times
   *

trace

tyi

tyo

untrace

zerop

==========
Initially bound atoms:
==========

nil => nil

obarray => an array of 128 lists (hash buckets) of atoms, all interned atoms are on one of these lists.

pi => 3.1415926

t => t

2pi => 6.2831852

... and a baby's arm holding an apple.

list:
l:

This command lists a directory. It takes various control arguments and also an optional star name to match in the listing operation. The default star name is "**", i.e. the wild-card star name. The optional control arguments are "-d" to only list directories, "-s" to only list segments, "-l" to only list links (the default is to list all three and any combination may be given on the command line) and "-p <dname>" to list in the directory named. Note that if a directory name starts with a ">" character or a "<" character, then the "-p" is not necessary before the name, as the list command knows that this must be a dirname and not a star name. Here are some examples of various list commands:

list **
lists all files in the current working dir.

l
does the same as the above.

list >sll motd**
lists all files in the dir >sll that start with the first four chars "motd".

l -p source foo*.* -d
if the working dir is >u>ota, for example, this lists all dirs in the directory >u>ota>source that have two component names where the first three characters of the first component are "foo".

The listing format first gives the access you have on the segment (any combination of "r", "e" and "w" for read permission, execute permission and write permission), followed by the number of 2K pages allocated to the segment, followed by the byte count (calculated from the bit count) in parenthesis, followed by a "*" if the segment has been modified since it has been backed up, followed by the name of the segment, followed by the address space name if not "", followed by the ring number in brackets.

The -since mm/dd/yy.hh:mm:ss option lists only those segments that have been modified since the specified date.

midi is a middle scale assembler for the 7/32

midi filename options

causes midi to assemble filename.sysin into an object file named filename.

The options are:

-nc supresses code generation
-ls generates a listing on the console
-ni stops the listing of include files
-nx stops the listing of macro expansions

midi knows about most of the machine opcodes and many pseudo-operations:

entry	defines an entry point
extrn	defines an external symbol
extrd	defines an external data reference
equ	sets a symbol
set	sets a symbol weakly (no error checking)
=	assigns the remainder of the line to a variable
end	ends the source program
.ent #	generates an entry sequence with # bytes of automatic storage
.rtn	generates the return sequence
pc	sets the program counter to: 0 absolute/no code 1 pure code 7 impure code
dsect	same as pc 0
pure	same as pc 1
static	same as pc 7
macro	defines a macro, macros may not define macros
mend	ends a macro definition
@	indicates that the next symbol is to be replaced by its value, either a number or a string
'	quotes an @ or quoted string
incl	includes the specified file.incl.sysin in the source
copy	like incl, except only for the first pass
dc	defines constants (halfwords)
dac	defines constants (fullwords)
ds	define storage NB: this rounds up to the nearest halfword
align	aligns storage on the specified boundary

```
.type           types out the remainder of the line
.push           pushes the numerical or string value of a symbol onto the stack
.pop                    pops the numerical or string value of a symbol off of the stack
a .len b                sets a to the length of the string variable b

.if                     conditional assembly, if arg>0 then true
.ifeq           compares its two string arguments e.g. .ifeq a,b
                        the comma separates the arguments
                        the one argument (no comma) form translates as .if exists
.ifne           like .ifeq except that its truth value is reversed
.else           flip state of truth flag
.fi                     balances a .if
```

(see also: midi.macros)

Some Thoughts On Macro Assemblers

Magic Six has had a small macro assembler named midi since this summer. It has proven to be a valuable test bed for a number of macro invocation ideas and will probably evolve into a full fledged macro assembler over the next several months. Since its inception it has supported a weak macro expansion capability which has slowly been changing into a more advanced one.

Originally, macro arguments could only be tokens. Thus mumble and x'f8010000' could be passed to a macro as an argument while mumble+1 or 246+x'f8010000' could not be. This allowed for the simple use of macros which did not do anything complicated or require complicated arguments.

More recently it was decided to try out a newer scheme. This one involved the use of string variables. In a typical assembler symbols may have numerical values and a number of other properties such as relocatability and entry-ness. When an expression is evaluated these values are substituted for the symbols. A string variable may have an indeterminately long character string value. An assignment statement would exist to allow variables to be given string values. In addition, when a macro is invoked each of its parameters is bound to its appropriate string value as an argument after the old binding is pushed on a stack. This is similar to the "shallow binding" mechanism used used in MACLISP (as well as Arch Lisp).

As with most such schemes the issue of exactly how and when string variables (symbols bound to strings) would be replaced with their values. After a false start it was decided to implement the substitution process as a series of passes over the source line each of which substituted each of the string variables encountered with its value. To allow for arbitrary construction of tokens constant strings of the form " string " were allowed and were defined to be replaced by their contents whenever they were encountered. Thus, if foo was bound to "jeckle", then:

```
        dc      "c'"foo"'"
```

would become:

```
        dc c'jeckle'
```

This had a number of problems. One involved the casual substitution of items in string assigments, for example, the a in:

```
        test=This is a test.
```

might very well be replaced by the value of some macro parameter. Other problems included the same problem as requires the existence of at least one fexpr in a lisp implementation. That is, how does one reassign to a symbol? If a is bound to "b" then:

```
a=c
```

would become:

```
b=c
```

and the value of a would be unchanged. This caused a number of capricious and arbitrary rules to be generated for when string values were substituted and when they were not. Finally, if a were bound to "a" substitution would be continued indefinitely unless a level limit were placed on substitution, ... , and so on. It seemed like land war in Asia. [credit for that goes to Bernard Greenberg at Honeywell CISL]

After discussing the problem with Daniel Weinreb at the AI Lab we reached the conclusion that the problem was basically the same as that in the original "lunar macro" package, which is a set of macro defining macros for use with MACLISP. [They were called the lunar macros since they were written by David Moon also at AI.] The "semi-lunar" and "cis-lunar" macro packages solved thus problem by requiring that all substitutions be explicitly indicated.

The character "@" was chosen as the string "indirection" operator during the substitution pass. No substitutions would occur unless the symbol was preceded by an at-sign. An at-sign followed by an at-sign would be ignored so that multiple level of "indirection" could be supported. Quoted strings would be substituted as before. Thus:

```
a = This is a test.
```

would work correctly. This means that all references to the values of arguments would have to be preceded by an at-sign, but in exchange eliminated a number of possible "accidental" substitutions.

As a final feature a quoting character is needed to allow an at-sign to be assigned to a string (for example), as in:

```
xyz='@abc
```

which would set xyz to "@abc". Note that:

```
xyz="@abc"
```

would set xyz to whatever abc was bound due to the way quoted strings are substituted for. In addition it is now possible to stuff quoted strings into strings by using accent ague to quote them. It has been suggested (but not yet implemented) that the meanings of @ and ' be more or less reversed during macro definition as in the semi-lunar macros.

One reason for the recent interest in macros is the desire to improve the readability of assembler code. One common method of doing this is by means of a set of structured programming macros. This suggests that one method of eliminating the short/long branch problem is by not allowing the user to directly code the branch instructions so that a table of label/branch interdependencies can be built and optimized. In a higher level language this is not as important implying that goto's (in a very limited sense) are "harmful" only in lower level languages but not higher level ones.

[The long/short branch problem is common on machines which have two flavors of branch instructions, which require different amounts of memory so that it is desirable to use the shortest ones possible. These machines include the IBM 1130, all recent (model 3 and up) Interdata Machines, many microprocessors and a host of others.]

As a final note, the strangest macro expander I have run into is Multic's mexp which is a two pass macro expander, the first expands all macros and their arguments and the second evaluates all conditionals.

The MINI command runs the miniature assembler written by SAS. It is primarily used to assemble compiler output, but can be used by users also. The command format is:

MINI <filename>

where the filename will have ".sysin" concatenated to its end by MINI. The automatic assembly feature of PL1 makes use of this command for PL1 output obsolete, as most programs are automatically assembled rather than manually.

The output is to the segment <filename> without any modifiers, such as .text.

For your convenience and safety, MagicSix keeps track of the phase of the earth's moon, Luna. The three commands are:

moon Prints out a diagram of the current lunar phase.

ppom Prints phase of moon, short format.

plpom Prints long version of phase of moon.

Related entry points:

```
moon$pom   returns char string for ppom
   dcl moon$pom entry (bit(64), char (20) varying);
   call moon$pom (time, short_pom_str);
   where: time is a standard system time code, (zero means current time)
          short_pom_str is the corresponding quarter and offset

moon$lpom  same as moon$pom, but returns long verbose string
   dcl moon$lpom entry (bit(64), char (80) varying);
   call moon$lpom (time, long_pom_str);

moon$fpom  returns phase as a floating point number between 0.0 & 1.0
   dcl moon$fpom (bit(64), float);
   call moon$fpom (time, ratio);
   where: time is as above
          ratio is a floating point number between 0.0 and 1.0,
             0.0  new moon
             0.25 first quarter
             0.5  full moon
             0.75 last quarter
```

NITE 04/15/78

Nite is a text justifier. It is supposed to be extensible and all that but only time will tell. At any rate it should be good enough for UROP reports and the like. By default the input text comes from foo.nite and the output goes to your terminal. The output destination can be changed with the tty, file, detach, and attach commands. Here are a list of available commands. As usual with these things all commands are the first and only things on a line and begin with a dot. Any other lines beginning with ". " are treated as nite comments, they are ignored by nite, and do not effect the finished document.

In general numeric arguments which refer to white space can be given in terms of character positions or inches or centimeters. (e.g. .linel 6.5i sets the line length to 6.5 inches).

linel - (ll) this takes a number and makes that the line length.
pagel - (pl) This sets the page length
page - (pa) Causes a page break (starts a new page).
break - (br) This causes a break in the filling of text.
lines - (ls) this sets the line spacing to its argument. .double -> .ls 2
tty - send output to the tty pauses at every page. type any non printing character to continue
file - if no args returns to previous file else send output to specified file.
attach - reattach user_output and send output thru user_output. This takes an argument which is the attach description that is passed to iocs$attach. For example .attach tty_io carousel will make your output go to the carousel with formfeeds and .attach file_io filename -msf -max 8192 will make your output go to a multisegment file named filename001, filename002, etc. Using dprint -msf this can be a very useful feature.
detach - reverts user_output to your terminal and nite output to a file or the tty depending on .files and .ttys.
fill - (fi) make line as long as possible.
nofill - (nf) take each line as it comes. The above two commands take a modifier (l, left, r, right, b, both, c, center) which specifies the alignment of the line. l means justify is left, this is the normal mode. r means make the right margin even. b means make both margins even. and c centers the line.
indent - (in) Takes a number and indents all suceeding lines that far. If the number is preceeded by a sign then the new indenting is being specified relative to the current indenting.
undent - (un) This takes an amount to temporarily un-indent. The change last only for one line.
read - This takes the name of a file which is read and processed by nite as though it had appeared in the main file rather then the ".read".
begin - This pushes a new environment. All old parameters are carried

across but changes will go away when the environment is poped. This includes the definition of variables in these frames.

end - This pops the current frame.

setvbl - This sets the variables named by the first arg to the string consisting of the rest of the line. If a previous variable of the same name existed it value will be bashed.

setlvbl - The sets a local variable. It is the same as the above except that the value of the same variable in previous frames is saved and restored when the current frame is poped.

macro - This defines a macro with the name of the first argument. The definition is ended with a .mend <macro_name>. Only one space is alloued between the mend and the name and there may be no trailing spaces. The macro is invoked whenever the command .<macro_name> is encountered durring normal parsing. No breaks are generated.

skip - The takes a number and generates that many blank lines. This is exactly equivalent to a like number of carraige returns. If the line spacing is set to three (3) then three times as many physical lines will be output.

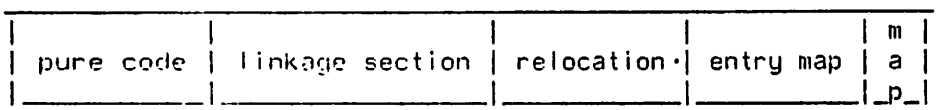
block - This insures that <arg> blank lines will be left blank contiguously. The number of lines is interpreted like in the skip command.

Special variables are built into nite. The four variables "long_date", "date", "time", and "page" are set to the current date, time and page number when nite is started and can be used any time. Also the macros "header" and "footer" are called at the beginning and end of every page (except the beginning of the first page), if they are defined.

For an example of a nite input file, see the file used to create this documentation ">doc>info>nite.nite".

The Magic 6 Object Segment Format

An object segment looks more or less as follows:



where:

pure code — is the executable code and other pure portions of the program

linkage section — is the impure section of the program

relocation — is the relocation bits for the program which are a string of structures as follows:

```
dcl 1 relocation_info based,
      2 type fixed bin(7), /* 1,2,7, or 8 */
      2 number_of_hws bit(8), /* usually "a0"b4 */
      2 offset fixed bin, /* offset into section */
      2 bits bit(80); /* one per halfword */
```

for relocating pure code:
- 1 indicates a reference to the linkage section
- 2 indicates a reference to pure code

for relocating the linkage section:
- 7 indicates a reference to the linkage section
- 8 indicates a reference to pure code

for all other types of blocks the offset is the total size of the block (including 4 bytes for the header)

for object information
- 65 indicates a statement map follows which consists of an array of 16 bit entries
  - bits 0...5 are the increment in PL/I line number
  - bits 6...15 are the increment in code offset
- 66 indicates that this block contains printable information on who and what compiled this program when
- 67 indicates that the next two halfwords are the start of a pc1 and pc7 block respectively

and are used to indicate the bounds figured out by the binder when creating the segment

The blocks appear in the relocation section serially and are terminated by a halfword containing zero.

entry map

The entry map consists of a series of names each preceded by a halfword length and consisting of an ascii string. At the first halfword after the string there is an offset into the pure code.

To use this:

```
dcl 1 map_entry based(mp),
      2 nchars fixed bin,
      2 name char(1);
dcl offset fixed bin based;

do mp=map_start
   repeat addrel(map_start,elen)
   while(map_entry.nchars^=0);

   elen=bit(nchars+1,16)&"fffe"b4;
   if substr(name,1,nchars)=epname
      then ep=ptr(baseno(map_start),
                  addrel(mp,elen-2)->offset);

   end;
```

To get a pointer to the start of the map given a pointer to the segment and its bit count look up scs$get_obj_map.

map

is a structure giving the offsets within the segment of the various sections described above. The structure is declared as follows:

```
dcl 1 object_map based,
      2 code_section fixed bin,
      2 entry_map_offset fixed bin,
      2 linkage_section fixed bin,
      2 relocation_bits fixed bin;
```

If relocation_bits is zero then their are none.

The online salvager, ols, is a program that cleans up a directory. It basically reallocates everything in the dir in a wired tempseg that is not part of the file system, and then truncates the dir being salvaged and copies the new contents from the tempseg back to the dir. This program is useful when a hash bucket is somewhat damaged or something. Be careful using it on directories that are badly smashed, it may loop. If you do not know what this does clearly, then do not use it.

options(variable)
pass descriptors for interpretation at run-time

options(fixed)
put link in object code (hcs only), or perhaps scs, since the links get snapped in place.

options(float)
use old style link so no lot entry need be mainfest. These are very useful for I/O modules since the link may not be snapped from the same address space as when they were set.

options(read)
set read only bit for an argument, very useful for ring 0 code only, remember to set validation on

options(nomap)
indicates that a pointer argument is to be received from the calling address space verbatim, as opposed to having been mapped so that its target is accessible to the called routine

Peek is a program which gives all sorts of useful and interesting information about the current users of the system. The current columns are as fiollous.

name
tcb address
pinwheel pointee
psw
console device address
idle time
pinwheel quantum
CPU time
current address space

The pl1 command runs the PL/I compiler.

Its usage is: pl1 pathname options

where pathname is the name of the file to compile.

The options may be:

-nc suppresses code generation
-na suppresses the assembly only
-se short error message mode
-sz takes the next argument as a hex number of 4K blocks that the compiler is to use for scratch storage
-ls print listing
-nd suppresses the generation of debugging information
-tb causes the compiler to generate a symbol file

(see also: pl1.language and pl1.changes)

General Notes on The Architecture Machine PL/I Language Subset

Numbers:

There are numbers: fixed and float

fixed are like Fortran INTEGER
float are like Fortran REAL

fixed constants include: 1, 247, -3121
float constants include: 1.0, 247.0, 0.0064, 12352.34, 1e23

operations on fixed and float constants include: +, -, *, /, **

If both operands are fixed the result is fixed, otherwise it is float.

```
2/3              yields 0 (fixed)
2./3             yields 0.666667 (float)
66.2+32.1 yields 98.3 (float)
12+7             yields 19 (fixed)
```

To convert from fixed to float and back there are two "builtin" functions: fixed and float.

```
float(1)                yields 1.0
fixed(2.3)       yields 2
```

In addition fixed and float numbers may be compared with the basic 6 relational operators: >, <, >=, <=, =, ^=.

Bits:

Bits are binary digits. PL/I supports bits and bit strings.

bit constants: "101010"b, "1"b, ""b, "000111"b

operations on bits include: &, |, ^

```
"1010"b&"1100"b yields "1000"b
"1"b|"0"b                       yields "1"b
^"0011"b                        yields "1100"b
```

To extract a substring from a bit string:
substr(bit string,first bit,number of bits)

```
substr("1010"b,3,1)     yields "1"b
substr("110011"b,3,3)   yields "001"b
```

The relational operators apply to bit strings. "1"b is greater than "0"b and the shorter string is assumed to be filled out with blanks. Thus, "1"b is greater than "01"b.

NOTE:
The result of all relational operations are bits. "1"b if the comparison was true, "0"b if it had been false. Thus: i>j & j>k is a bit expression.

Characters:

character constants: "abcd", "", "the quux ate luux"

|| is the concatenation operator

"abc"||" "||"def" yields "abc def"

substr works on character strings more or less as it does on bit strings.

substr("abcdef",2,3) yields "bcd"
substr("foo",2,1) yields "o"

Variables:

Variables are places to put values. Variable names may be very (128 or more) letters long and may contain all the letters (upper or lower case), $, _ and the numbers. They may begin with any of these characters except the numbers.

abc, i, quux, foo_bar, age_of_time and so on are variables

A variable may contain:

a fixed number
a float number
any given number of bits
any given number of characters
up to any given number of characters, plus how many there are

Declarations:

To declare a variable:

```
 dcl (or declare) variable the type;
```

or

```
 declare variable type, another type2, yet_another type3;
```

Semicolon ends the statement!

Instead of a single variable you can use a list of them such as: (var1,quux,var2). So:

```
 dcl (i,j,k) fixed, name char(32) variable;

 dcl (x,y) float;

 dcl able_to_read bit(1);

 dcl letters_seen_today char(26) varying;
```

Arrays:

If there is more than one of a given thing:

```
 dcl thing float;
 dcl things(10) float;
```

causes: thing(1), thing(2), ...

To specify the range of allowable subscripts:

```
 dcl things(19:47) char(32) varying;
```

causes: thing(19),thing(20), ... ,thing(47)

Structures:

To group things together for many reasons:

```
 dcl 1 line,
         2 x1 float,
         2 y1 float,
         2 x2 float,
         2 y2 float;
```

builds a box called line thusly,

```
 _____
|x1__|  line.x1
|y1__|  line.y1
|x2__|  line.x2
|y2__|  line.y2
```

Structures can be used to just organize things or for more powerful purposes.

Both:

A structure may contain arrays and an array may contain structures.

```
dcl 1 line(10),
        2 x1 float,
        2 y1 float,
        2 x2 float,
        2 y2 float;
```

causes: line(1).x1,line(1).y1,line(1).x2, ...
.line(10).y1,line(10).x2,line(10).y2

or perhaps,

```
dcl 1 shape,
        2 nsides fixed,
        2 side(12),
          3 x1 float,
          3 y1 float,
          3 x2 float,
          3 y2 float;
```

which could be abbreviated:

```
dcl 1 shape,
        2 nsides fixed,
        2 side(12) like line;
```

and causes: shape.nsides, shape.side(1).x1, shape.side(1).y1
and so on.

Brief Note on Syntax:

PL/I accepts input in free format, no columns are reserved. Each statement must be ended by a semicolon and any number of statements may appear on a line. It is good practice not

to goonch all the code together since it gets hard to read and hard to trace in the debugger which isn't too bright.

Comments may appear anywhere and have about as much impact as a blank space. They begin with the sequence /* and may contain anything except a */ which ends them.

Assignment:

It has already been implied that there is some way to set a variable to a given value. One of the most common ways is via the assignment statement:

thing to assign to = thing to assign;

Examples:

```
i=1;
name="John Doe";
pudding_proved=pudding_weight>10.4 & pudding_density<=0.9 & ^fruit_cake;
dist=x**x+y*y;
```

The substr builtin function mentioned above can be used on the left hand side of an equal side to indicate that a substring of the string variable specified is to be modified. For example:

```
dcl c char(8);
c="";
substr(c,2,4)="quux";
```

would set c to " quux "

Procedures:

The atomic unit of code is the procedure.

```
nuke:procedure;

end;
```

is the procedure nuke. Anything between the procedure statement and the end statement is in the body of the procedure.

A procedure has associated with it all variables which are declared in it. So:

```
nuke:proc;
```

```
dcl abomb fixed;
dcl hbomb float;

        luke:proc;

         dcl abomb char(32);
         dcl qbomb bit(1);

        end;
end;
```

has abomb and hbomb declared in it. There may be other variables named abomb and hbomb but the abomb and hbomb declared in nuke are only known inside of nuke.

nuke also has luke declared in it. luke has abomb and qbomb declared in it. Inside of luke, abomb and qbomb refer to the variables declared therein. hbomb refers to the hbomb declared in nuke, the "containing block". Inside of nuke qbomb is unknown and abomb refers to the one declared in nuke.

This nesting and masking of variable names is known as "lexical scoping".

Flow of Control:

Normally the flow of control in a program is sequential. Each statement is executed in turn. This can be altered in several ways:

- branching with a goto
- executing an if statement
- executing a do statement
- calling a procedure
- invoking a function

Branching with a goto:

The syntax of this is: go to place;

where place must appear somewhere in the program at the start of a statement and followed by a colon.

here: go to here; will loop indefinitely

here is declared to be a label at that location.

Label values can be put into label variables.
For example:

```
 dcl mabel label;

quux: mabel=quux;
         go to mabel; will loop indefinitely
```

Sometimes it is useful to branch to a given location
given an integer value (for example the Fortran computed
goto ...). To do this declare a label array:

```
 dcl actions(7) label constant;
```

and then you can define labels such as:

```
label(1):
```

which would be the target of a branch to label(i)
when i was equal to 1.

Executing an if statement:

```
if condition then action1;
                  else action2;
```

causes the condition to be evaluated to a bit(1) value
and if it is "1"b executes action1, otherwise it executes
action2. So:

```
 if a>b then max=a;
           else max=b;
```

Indentation is not required by the language but if you
don't bother don't expect to debug your code. Ask people
how to set the tabs on your IMLAC, it is not much worse
than on a Selectric.

An action may either be a statement or a statement group.
The former we have been describing all along but the latter
is a special construct:

```
 if a>b then do;
                  max=a;
                  min=b;
                  end;
           else do;
                  max=b;
```

```
                    min=a;
                    end;
```

Note that end is used to end statement groups and to close procedures. PL/I is full of this kind of thing, notice: a=b=c; is not unreasonable in PL/I.

if-then-else clauses may be nested and otherwise weirdly grouped when the program logic gets hirsute.

```
if a>b then if c>d then flag="11"b;
                   else flag="10"b;
       else if e>f then flag="01"b;
                        else flag="00"b;
```

In this example the indentation should indicate which if, then and else go together.

One of the most common constructs is:

```
if a=1 then do;
        end;
else if b=44.3 then do;
        end;
else if command="GO" then do;
        end;
```

and so on.

This "else if" construct is used so frequently that some languages have an elseif statement.

Executing a do statement:

The do statement is perhaps one of the most powerful and versatile statements in the language. The simplest case was recently mentioned and simply groups the contained statements into a group for use with if clauses.

do also allows the body of a do group to be executed iteratively (like the Fortran do statement):

```
 do variable=start to end [by increment];
```

which means that to add up all the numbers from 1 to 10...

```
        sum=0;
        do i=1 to 10;
                sum=sum+i;
```

```
        end;
```

while to add the numbers from 1 to 10 backwards...

```
sum=0;
do i=10 to 1 by -1;
        sum=sum+i;
        end;
```

The loop figures out which direction to go if it isn't obvious at compile time.

NOTE: The values in the to and by clause are calculated when the loop is entered, not each time through! So,

```
 do i=i to n;
        end;
```

cannot be exited by setting n to 1 or less than i, since its initial value has been stored away somewhere.

A more general form of the do allows the user to completely specify the test and the iteration step:

```
 do variable=start repeat iteration [while(condition)];
```

This sets the variable to the start value and as long as the condition is true (if you leave out the condition it will loop forever) executes the body of the do group. Each time it reaches the end it sets the value of the variable to the recalculated value of the iteration expression.

So to add the numbers from 1 to 10...

```
sum=0;
do i=1 repeat i+1 while(i<=10);
        sum=sum+i;
        end;
```

Another form of the do simply indicates that the body is to be iterated until some condition is no longer true. This involves the while clause alone, as in:

```
do while(error_margin>0.001);
        end;
```

The while clause may also be combined with the to and by constructs to provide another escape from the loop in addition to the to clause, as in:

```
do i=1 to n_people while(person(i)^="Jimmy");
        end;
```

Calling a procedure:

Calling is done via the call statement. Thus to call the procedure nuke (from above):

```
call nuke;
```

This causes a branch to the program which nuke which immediately grabs some storage from somewhere and saves all the information it will need to get back to right after the place it was called. nuke also uses this storage for other things, but as soon as nuke is done executing that storage is released to whomever wants it and the flow of control resumes where it left off. (This ability to save the machine state is at the heart of a good deal of modern recursive function theory ...)

Executing a return statement will cause the procedure executing to return to its caller as will "falling through" to the end statement.

Arguments and Parameters:

When a procedure is called it is possible to pass it a number of values or variables (references) to work with. These values or variables are passed as arguments and are received as parameters.

```
zoot:proc(soot,boot);
        dcl (soot,boot) float;
        soot=boot;
        end;
```

gives an example of procedure with two floating point parameters.

If x is declared float and i is declared fixed.

call zoot(x,4); would set x to 4.0 since zoot requires two float arguments, 4 would be converted to 4.0 and passed as an argument. x would be passed by reference so that soot in zoot would refer to x.

call zoot(i,x); would not appear to do anything since

i would have its value converted to floating point and passed as an argument by value. Thus, soot would refer to this temporary value as opposed to i. x would be passed by reference.

External Procedures:

In order for PL/I to generate code for a given call statement it must know the attributes expected for each argument for the procedure in question. Toward this end there is the data type entry and the entry declaration:

```
dcl things to declare entry[([attribute1[,attribute2 ...]);
```

to declare nuke, dcl nuke entry;
to declare zoot, dcl zoot entry(float,float);

to declare a procedure which accepts a char(32) values, two floating point numbers and a bit,

```
dcl moby entry(char(32),float,float,bit(1));
```

Some external entries are very versatile so they can figure out how they have been called at runtime by means of the options(variable) clause. Something you may see a lot is:

```
dcl (ioa,ask) entry options(variable);
```

These procedures may be called with any number or type of arguments and if they can figure out what to do with them they will. PL/I has no language feature to write that kind of program but there are subroutines that can be called to do a lot of it.

Functions:

Any procedure may return a value so that it may be used in an expression. This is done by means of the returns clause in the procedure statement. A routine which returns a floating point number could be written:

```
won:proc(x) returns(float);
 dcl x float;
 return(sin(x)*sin(x)+cos(x)*cos(x));
end;
```

Note that the return statement indicates the value to return. This value will be forced to the appropriate

type before it is returned, thus the above program could have been written with a "return(1);".

External entries may be declared with a returns clause as well. For example:

```
dcl mumble entry(float,float) returns(char(64)varying);
dcl ages entry returns(fixed);
```

These routines may be used in functions:

```
i=ages()+1;
if mumber(uon(),i/2)="" then ...
```

The print (pr, type, ty) command prints a file on your console. The usage is:

```
print <pathname> [-no_header | -noheader | -nhe]
```

Unless the -no_header option is given, the file will be preceded by a header giving the full path name and the date-time modified. The arguments can, of course, be given in any order.

The dprint (dp) command can be used to print a file on the line printer.

This is the documentation for the symbolic debugge, pb.
Here is what the commands do. Note that numbers are
in hex unless preceded by a control d character.

Command ^j (line feed):
This command opens the next location, by adding four to the current location and redisplaying it.

Command ^m (carriage return):
This command opens the previous location from the current, by substracting four from the current open location.

Command .ch:
This command prints the current location out as a char n string, where n defaults to four and can be specified by preceding the command with a number.

Command .fi:
This command prints the current location in decimal, treating it as a fixed(31) variable.

Command .hi:
This command does the same as .fi, only for a fixed(15) variable instead.

Command .r:
This command prints in the current stack frame the value of register n, where n is a standard arg to the command, i.e. is placed immediately preceding the "." character. For example,
3.r
would display register 3.

Command +:
This command adds its argument to the current location, and the argument defaults to 2 if not specified.

Command -:
This command does the same (with the same defaults) as the "+" command only it subtracts the argument from the current location instead of adding it in.

Command ::
This command modifies the open fullword. The argument is the value that is put in the location. If there is no argument, then the value of zero is used.

Command .in:
The .in command is used to initiate segments. The name of the segment to be initiated is put after the .in, and exactly one space is assumed, so that files with weird names can be easily handled. The command prints out what segment the segment is initiated into, and the code returned from initiate. If .in is given a numerical argument then it attempts to initiate the pathname in a segment with that number.

Command ..:
The .. command is used to send commands to the command processor via the scs$cl entry. The .. is immediately followed by the command line as it is to be sent to the cl entry.

Command .ins:
The .ins provides the macro facility. It is preceded by the buffer number, and is followed by the command name, which should start with a dot. You then input the macro, ending with a line that contains only a backslash. You can then run the macro by typing the command name that you have given it.

Command .ps:
The .ps command is used to set plod break points. It takes an argument, which is the line number at which to set the break point. As usual, the break is before the line. The program name is taken from the last .use command.

Command .pk:
This command kills all pll break points that have been set in this process.

Command .pr:
This command is just like .ps only it kills the break point rather than setting it. It warns you if there is no such break.

Command .qs:

This command sets assembly break points. It is preceded by the location at which to set the break, or if none is provided, then it sets it at the current open location.

Command .qk:
This kills all assembly break points in the current process.

Command .qr:
This command works just like .qs only it resets the break point instead of setting it.

Command .pl:
This command lists all pl1 breaks.

Command .ql:
This command lists all assembly break points.

Command .fy:
This command turns on flow tracing of pl1 debug programs.

Command: .fn:
This command turns off flow tracing.

Command .w:
This is the watch command. It takes a location, which defaults to the current location, and watches it for changes. It is checked at every pl1 debug line entry, and if it changes, a warning is printed and you get a break point encountered message.

Command .s:
This is the symbol loader command. It has two forms.

```
.s
.s filename
```

In the first, the file sym will be used for the symbols file, while in the second, the file filename.sym will be used. Expand path is called.

Command .tr:
The .tr command does stack traces. If preceded by a number, only the last n frames will be traced. If followed by an "l", then the long

form stack frame trace will be done, which prints all registers.

Command q:
The q command causes the debugger to return to its caller. This will work on pl1 breaks and assembly breaks as a break return, see ^g's documentation for more information. This command simply returns.

Command t:
The t command chases a pointer. The pointer at the current open location is used as the address of the new current location.

Command o:
The o command does the same as t, only it only takes a halfword and adds in the offset base, set by the .so command.

Command .so:
This command sets the offset base. All o commands have this value added to them to get the new current location. If no argument is provided, then it is assumed that the offset base is the current location, otherwise the argument is the new offset base.

Command .sb:
This command works like .so, only it sets the "based base". This is the value that all based variables are based at when using the symbol table hackery.

Command g:
This command returns from assembly break points in a clever way, namely that the current break points is cleared so that you can procede, and then it is replaced after the single instruction that was break pointed has run. This is not the standard return since some segments are read-only and this would cause a segfault in some cases trying to catch execution after the break point.

Command p:
This is the pl1 next line instruction. It executes one line of pl1 and then hits a breakpoint.

Command .use:
This command takes two forms, <n>.use and .use <name>. In the first form, it tells the debugger to use frame n for

examination of such things as registers. The second is more powerful. This form tells it to look up the stack for a pl1 entry with that name. If it can find it, it uses that frame, otherwise it prints a warning. It then looks in the current symbol table and searches for the symbol table for the proc. If it can not find one, then it again prints a warning. If the symbol table entry is found, then all further symbol table requests will be done relative to that procedure.

Command .ti:
This command does text insertion. The rest of the line after the space after the ".ti" is inserted from the current location until the <cr> is found. The <cr> is not inserted.

Command v:
This command prints a char varying string. It prints the one at the current location. Giving it a location of course opens the new location, i.e.
3000v
prints the char varying string at location x'3000'.

Command n:
This is the "next" command to execute one assembly instruction after an assembly type break has been encountered.

Command .g:
This command returns from a macro to reading commands from the terminal.

Command .tmr:
This command takes one argument, which is the number of the segment to terminate.

Command .map:
This command takes a segment number like .tmr only instead lists all of the entry points in the particular segment.

Command .st:
This command takes one argument, and controls

subroutine tracing with the .n command. If you do a
1.st, then subroutines will be traced, and 0.n resets
this. The default mode is not to trace subroutines.

Here are some examples of how to do some of the more commonly performed operations in probe.

To use a symbol table foo.sym, and look at the variable bar in proedure mars:

```
.s foo                  ;need only be done once per process.
.use mars                       ;prints spurious warnings sometimes.
,bar                            ;prints the value of bar.
```

To chase a pointer, you use the
t command.

If you have an area at location 50330 for example, and have lots of offsets relative to this area that you want the "o" command to work with, then do

```
50330.so                        ;sets the offset base to 50330
```

and then from then on, the "o" command will take the halfword at the currently open location, add it to the base value of 50330 and move to this location. This is the equivalent of the "t" command for offsets instead of pointers.

Note that probe currently does not work for debugging programs that run in weird address spaces.

To run a program, watching to see if a static variable foo in the procedure mars, with symbol tabl mars.sym, has changed, we do the following.

```
pb
.fy                             ;turn on flow tracing.
.s mars                 ;tell pb which symbol table to use.
.use mars                       ;tell pb which procedure to use.
,foo                            ;open the location foo
.w                              ;tell probe to watch this location.
q                               ;tell probe to quit.
```

Then type at command level:

```
mars                            ;tell command processor to run the program mars.
                                ;pb will print every line number, and will stop
                                ;if the value of foo changes.
```

Commands are:

t	Print message.
n	Move to next message.
d	Mark message for deletion.
u	Undelete message.
p	Move to previous message.
l	Log current message (creates logbox if necessary)
s	Summarize undeleted messages. (* indicates current message)
a	Summarize all messages. (D indicates message is marked for deletion)
q	Quit rmail, deleting marked messages.
x	Same as q.
j	Jump to message. (No argument - jump to 1st message)
f	Forward message, prompting for recipients.
r	Reply to message.
?	This information. (obviously)
" "	Print the next <argument> lines of this message. (default = all)

<n> Becomes the argument for the following command. When no argument is supplied, the current message is used (except where specified above).

Command line options:

-summarize	Summarize the mailbox, then quit.
-log	Use the logbox.

The standard default ready procedure.

It is automatically invoked by the command processor after each command. By default it gives the time of day followed by the CPU time used by the last command followed by the pathname of the current working directory. It also reports the current address space if it isn't the default one, gives you the level if the level is not one and warns you when the number of segments in the current address space is less than 3.

rdy or ready optionally can be called with an argument which is the ioa control string to use when printing the ready message. This ioa string will only be used for that call to ready. If the -set option is specified the next argument is retained for the life of the process as the ioa string to be used by ready. Your profile segment can also contain a "ready_string" field which is made the default when ever you login.

The ioa string shuffles the arguments by using the ^g command and prints out any information that is of interest. Following is a list of the argument that ioa is called with.

time_string	This is the current time in hh:mm:ss format.
inc_cpu	This is the cpu time used since the last ready message as a floating point number. (use ^f)
working_dir	The pathname of the current working directory.
level_number	The number of the current command processor level.
cardinality	This is a 2 character string giving the proper cardinality for the level number. (eg if level_number = 3, cardinality = "rd")
space_name	This is a char(4) or fixed(31) variable which is the current address space.
warn_message	This is a character string which if included will warn of few segments.
number_users	The number of users.
inc_pf	The number of segment swapped in for you since the last rdy.
percent	The percentage of incremental cpu-time to incremental real-time.
free_segs	This is the number of free segments in the current space.
initials	These are the initials of all the users logged in.
last_command	This is the last command line typed at the command processor.

Examples of use:

OTA's ready proc:

```
^4g^v[^;^*^4g^i - ^]^12g^a (^2g^f ^9g^i) ^3g^a^r
```

The above string is represents my ready proc format. ^4g means select the fourth argument: the level number. the square brackets form a conditional since I only print out the level number if it is not one.

RK's ready proc:

```
^4g^v(>^)^8g^v(-^) ^i ^2g^f ^10g^f% ^i ^8g^v(-^)^4g^v(<^)^r
```

This string makes use of iteration (the ^(and ^) delimit the iteration).

The default ready proc:
"ready ^a (^f) ^c^v[^;^*, ^4g^i^c level^]^6g^v[^;^*, addr space: ^6g^c^]^7g^a\m";
This is a really hairy one since it tests the address space to see it is
a one (the default) and only if it isn't doe it print any thing out.

For more info on this stuff type help ioa.

If you have a special ready proc format and are not into hacking ioa
programs then send me a message and I'll be glad to help you work something
out.

rn
rename

RENAME COMMAND: rename <old-name> <new-name>

The rename command takes two arguments, the first is an absolute or relative pathname of the segment to be renamed, the second is the new segment name (which must be a relative pathname (no ">"s or "<"s in name)).

Examples:

rename foo bar	The segment now named "foo" in the current working directory is renamed "bar", if found.
rn >ding>dong>dang dank	The segment ">ding>dong>dang" becomes: ">ding>dong>dank"

document for sa:

This is the set_access command. It takes two arguments. The first is a pathname of a segment (or a directory), star names are allowed. The second is any combination of the three letters "r","e" and "w" for read, execute and write permission.

```
sa data r
                    allows read access only to segment "data"

sa ** rew
                    removes all access restrictions in the working dir
```

In general, object code segments should have the access "re" and source segments "rw" or "r". Dirs should have the access "r". If the segment is already initiated in any process, the access change will not take place until the segment is terminated and re-initiated, or until a new_proc takes place.

The set bit count command, sbc, sets the bit count of a file.
The usage is:
sbc <pathname> <bytecount>

The bit count is provided in bytes for convenience.

stty: a program to set terminal descriptions

stty accepts four types of arguments:
1) arguments which describe a bit
2) arguments which describe a numerical parameter
3) arguments which describe a terminal or terminal type
4) arguments which print out the current settings
5) arguments which specify a device

Bit values are modified by giving their name optionally preceeded by a not sign indicating that the bit is to be reset not set. Numerical parameters are followed by their new value expressed as a decimal number. Types are specified by their name. describe, des, show, and ? will cause the current modes to be printed. save say to change the flags used when the device is first attached to the current flags. Devices are specified by device, device_force, dv, df, stream, or st which all take one argument (either a device name or a stream name).

Examples:

```
stty pad ^more  ;this will set the pad bit and reset the more bit
stty page1 53   ;this sets the page length to 53
stty show       ;this will display all parameters and their current
                ;setting
stty device printer printer save ;this will first say to use the
                ;printer device, then reset its modes to the
                ;standard printer modes, and finally to make them
                ;the default modes
```

Settable Parameters:

bits:

```
echo                ;do echoing on input
raui                ;do no special processing on characters on input
rauo                ;output characters without translation
speed           ;initialize this device in the high speed mode
display         ;this terminal can do cursor positioning (is a Fox)
nobreaks            ;no breaks will be noticed
more                ;do more processing
reverse         ;reverse the _ and del keys
parity          ;do parity checking
wrap                ;assume that the scroll enable key is up
notabs          ;output tabs as spaces (for devices without hardware tabs)
formfeed            ;output control-l without translation
uc                  ;assume the terminal is uppercase only
ringbell            ;send a real control-g's to the terminal
visbell         ;send a strange looking sequence of characters to the
                 terminal for control-g's (if both ringbell and visbell
                 are off then control-g's come thru as "^g")
lf_before_cr    ;<cr> will be sent as <lf><cr> as opposed to <cr><lf>
```

```
numbers:
        pagel           ;number of usable lines on a page
        linel           ;number of usable columns on a line
        padcrlf         ;number of pad characters to send after a crlf

types:
        fox                     ;Perkin-Elmer Data Systems Fox - 1100
        la36                    ;DEC LA36 Decwriter II
        decwriter               ;       (suitable for most 30 cps terminals)
        carousel                ;Interdata Carousel P.O.S.
        printer         ;Bright BI 1215 Bar Printer
        host                    ;Semi-Intelligent Device
        network         ;       (actually meant for use with the Multics
                                ;               call out unit)
        glass           ;suitable for a CRT type terminal without
                                ;       using cursor positioning and tabs
        imlac           ;Imlac PDS 1
```

Swl is a program that allows you to swap your linkage section. It viciously increases the likelyhood of bad_luck faults, but gets you past those "tight" times when you thrash your ass off.

Welcome to the wonderful world of send_mail.

^C sends the mail.
Altmode (escape) is the command prefix. Sub-commands of alt are:

s	Subject: Prompts for subject line.
t	To: Prompts for additional users and adds them to the recipient list.
u	Un-to: Prompts for a user name and removes it from the list.
f	From: Prompts for a user name and changes the From field.
i	Insert file: Prompts for a file name and reads it into the buffer.
e	Edit escape: Calls tv on the buffer. Remember to write it back out!
n	Notify: Toggles notification of mail sent. Default: off.
q	Quit: Exits send_mail without sending the mail.
?	This information. (Obviously)

^L	redisplays the entire message.
^R	redisplays the current line.
^X	kills the current line.
^U	same as ^X.
del	rubs out the last character.

M A G N E T I C T A P E U T I L I T Y
tape, tp

The tape command dumps, loads and scans MagicSix format tapes. Every call to tape requires a sub-command, the (l)oad and (d)ump sub-commands also take a pathname and optional arguments. Arguments always force options which are not the defaults.

sub-commands:

r rewinds tape

f skips to just after the next eof mark

s scans the tape printing the files: dtb and size of each

d takes a pathname and dumps it to tape
-nw (-nowalk) Don't dump any sub directories.
-inc dump only those segments which have been modified since last dumped.
-time takes a date/time string and dumps all files modified since that date. Date is of standard form. See also date_to_binary.

l takes a pathname, locates it on the tape and reloads it
-rp (-replace) replace files found in the file system by those on tape.
-any reload and entry found with the given name regardless of the dir it was dumped from.

The tint command runs the editor TINT. This is the version TINT 8. It takes an optional pathname, which is read in to the buffer.

When you eq from tint, and then reenter, the buffer is still intact, and the current position is unchanged.

See tint.order for information about commands and other features.

REFERENCE MANUAL for TINT

Introduction

TINT (an acronym for Tint Is Not Teco) is a character oriented text editor. In addition to having a multitude of commands for editing text, TINT also supports macros, multiple buffers and q-registers.

The basic unit of text storage is the buffer. All characters manipulable by TINT must be in one. At any given time one buffer is always the "current" buffer and all text editing commands act on that buffer. Every buffer is identified by a name and may contain text and/or a number. Commands are available to create buffers, delete buffers, move numbers or characters to or from a buffer, and to make a particular buffer current.

A buffer may have the following properties:

a) a numerical value
b) a text string
c) a current position within that text string

The current position in a buffer is initially zero and determines a reference point for many commands. The position is always between two characters, thus, if the position is 5 then there are five characters in the buffer before the current position. The current position is referred to as the cursor or as "." in numerical expressions.

All TINT commands one character long and in general take two types of arguments:

1) numbers - are kept on an argument stack are picked off by each command as needed
2) character strings - follow the command and are terminated by a backslash

Numbers and numerical expressions are pushed onto the stack as

they are encountered. For example: 0, 255, 3+5, and -987 are numberical arguments. A command may pop no, one or two arguments off of the stack.

TINT is capable of the full range of arithmetic operations. The precedence is strictly left to right, no other order is imposed. Thus: -z+.*4 will evaluate as: 4*(.-z).

Parentheses may be used to control the order of evaluation of expressions.

The symbol "." (period) is treated as a number which is the current position within the current buffer. This is also known as the cursor.

The symbol "z" is treated as a number which is the number of characters in the current buffer.

The symbol "h" pushes zero and then z on the stack for specifying the whole buffer for commands which take arguments.

The symbol ":" (colon) sets the "colon flag" which is simply a flag which modifies the behavior of the next command which encounters it.

The symbol "^" (caret) sets the "caret flag" which is simply a behavior modifying flag such as the colon flag.

A number of control characters have special meanings in string arguments. Control-g quotes the next character so that any special meaning it may have is ignored. For example, to include a backslash in a string, use control-g,backslash.

Control-c allows the contents of a buffer to be included in a string argument as if it had been typed in there. Control-c is followed by the name of the buffer to include and then a backslash to terminate the buffer name. A control-d is similar to control-c except that the specified buffer is inserted verbatim. A buffer control-c'ed into a string argument may in turn contain control-c's.

Control-b inserts the name of the current buffer thus allowing TINT code to remember the original buffer if it switches buffers.

Control-x uses the next digit in the argument and and tells the character reader to look back that number of levels when reading characters. It continues to read at this level until a backslash is encountered at which time the reader "pops up" and continues reading from the top level. This allows a macro (a saved sequence of TINT commands) to tell with which arguments it was invoked since macro invocation simply "pushes" the reader level so that subsequent commands are taken from that buffer.

An example of the use of control-x follows:

```
ifoo^x1bar\
```

this would insert a "foo" followed by anything typed at the caller's level until a backslash terminated input at that level, followed by "bar". Thus if this string were in a macro called from command level, the stuff between "foo" and "bar" would be the characters following the macro name on the command line.

Control-x0 is a special case which indicates that input is to come from the keyboard at which the user is typing.

Basic Commands

i - insert

The string is inserted into the current text buffer right before the current position.

```
imumble\                    inserts "mumble" into the current
                            buffer

ithis is what is in buffer foo: ^cfoo\ see?\
                            would insert:
                                "this is what is in buffer foo:"
                                the contents of foo
                                "see?"
```

:i - insert until specified character encountered

The character immediately following the i is the delimiting character. NO special processing is done on the string.

```
:i@here are \^c^b^g some random characters@
                            would insert:
                                "here are \^c\b\g some random characters"
```

d - delete characters relative to the cursor

Deletes characters starting at the current cursor position. This works forward if the numerical argument is positive and backwards if the argument is negative.

```
5d                          deletes 5 characters between positions
                            . and .+5

-d                          deletes the character immediately before
                            the cursor
```

k - kills characters or lines

When called with no argument k will delete characters up to the next carriage return.

When called with one argument k will delete characters up to and including the next n carriage returns. If the argument is zero k kills back to the start of the current line. If the argument is negative k deletes back over the specified number of lines.

When called with two arguments k will delete the characters between the two specified buffer positions.

j - jump to position
This command sets the current position of the current buffer to its argument if it had one, otherwise to zero.

c - move forward characters
This command increments the current position of the current buffer by its argument, otherwise by one.

r - move backward characters
This command decrements the current position of the current buffer by its argument, otherwise by one.

l - move by lines
This command moves the cursor to just after the n-th carriage return from the current position. If no argument is given it moves the cursor to the start of the next line.

0l	sets the cursor at the start of the current line
l	moves to the start of the next line
-3l	moves back 3 lines

s - search for a string
The s command searches for the n-th occurence of its string argument starting at the current position. The cursor is left at the end of the string searched for in the direction of the search. If the argument is negative then a backwards search is done and the cursor is left at the start of the found string.

If no argument is given then s searches for the next occurrence.

If the specified string is not found either an error message is printed or the current macro or iteration level is exitted. In any event the cursor is never moved if the search fails.

t - type out text
If given no argument t types out the current line.

If given one argument t scans forward (or backward if the argument

is negative) passing over the specified number of carriage returns.
-5t5t prints the five lines before and the five after and including the current line

If given two arguments t types out all of the characters between those two positions in the current buffer.

e - external command
This is a command which takes several sub-commands:

ei - Input: reads the file specified by the string argument into the current buffer at the current position.
eo - Output: writes the current buffer out to the file specified by the string argument.
The current position and buffer remain the same.
ed - Delete: deletes the file specified.
em - Macro: reads the file into the buffer "external macro" and calls it that macro.
The buffer is automatically deleted after the macro is finished.
eq - Causes TINT to return to its caller.

! - pause to command level

= - print out the current numerical argument

Advanced Commands

b - create/select/delete buffers
If no argument or a zero argument is given the buffer specified by the string argument is selected as current after having been created if it had not already existed.

If a positive argument is given the specifed buffer is pushed before it is selected. This means that a new "instance" of the buffer is always created so that any buffer already existing by that name is inaccessible until this new buffer is deleted.

If a negative argument is given the specified buffer is deleted.

x - move text out of the current buffer (export)
This command takes arguments like the k command, but rather than destroying the text indicated it copies into the buffer specified by the string argument replacing the old contents of the target buffer.

hxfoobar\ copies all of the current buffer into buffer named "foobar"
1xline\ copies the rest of the current line into

the buffer named "line"

:x - append text to another buffer
This command is like x except that it appends the specified text to the buffer as opposed to bashing it.

^x - insert text into another buffer
This command is like x except that it inserts the specified text into the specified buffer at the current position within that buffer.

p - put a numerical value
This command puts its numerical argument into the numerical value of the buffer specified by the string argument.

q - get the value of a buffer
This command returns (pushes onto the numerical stack) the numerical value of the buffer specified by the string argument.

Assorted Hacks

^g - screw around with the gap (this one "rings the bell")
This undeletes then last <arg> characters that were deleted. It's great for error recovery.

^b - lists the names and useful information for all the buffers

^s - forces the redisplay of the line at the top of the screen

^c - erases the imlac screen

Macro Facility

The macro facility in TINT works by allowing the input stream to be redirected from the keyboard to the contents of a buffer. This buffer is then referred to as a macro since this redirection or invocation results in the execution of many TINT commands.

When the reader reaches the end of the buffer it returns control to wherever the reader was previously reading. Thus, control may return to the keyboard, but if a macro invokes a macro then control returns to the invoking macro.

If the macro is invoked with numerical arguments on the stack they are available for the macro to use as if they had been typed at the start of the macro.

String arguments to macros are accessible via the control-x feature described above.

m - invoke a macro
Redirects the TINT reader to the buffer specified by the string argument. Execution always begins at the first character in that buffer.

:m - invoke a macro with an errset
Behaves like m except that an error handler is established. If an error occurs inside of a handler then control is returned as if the end of the macro had been reached and an error code is placed in the numerical value of the buffer "error".

< - iteration command
Takes as a string argument all characters up to the next non-quoted ">" and places them in a new instance of a buffer named "iteration source" and then invokes that buffer as a macro either:
1) the number of times specified by the numerical argument or indefinitely if no argument was given
2) until an error occurs with no errset (see m above)

Whenever the iteration terminates the buffer is automatically deleted.

```
10<sfoobar\-6diquux\>   replaces the next 10
                        occurrences of "foobar" with "quux"
```

^x - make executable
If given an argument of 1 (or no argument) this command makes the character immediately following it be associated with the buffer of that name. Thus, ^xa would turn "a" into a regular TINT command which runs the TINT code in the buffer named "a" whenever it is encountered.

If the buffer named "]" contains: "^ci@\-8t8t-d" then ^x] would turn] into a command which clears the screen and displays the local 16 lines designating the cursor with an "@". To undo this type: 0^x]. The ^x command allows the user to turn TINT into a personally tailored edittor.

; - conditional termination
If given a positive or zero argument this command will terminate the most recent macro as if the end of the buffer had been reached. This makes it useful for ending iterations and so on. If it is given a negative argument this command does nothing.

The TC command takes a file name and truncates the file to zero length. It leaves the acls and names intact.

Quick Guide to tv on Magic Six

Type "tv", "tb" or "to" to get into tv.
If these commands are given arguments they perform the
following upon entering tv.

tv - a ^X^R is done on the argument.
tb - a ^X^F is done on the argument.
to - a ^X^O is done on the argument.

(See below for description of these commands.)

If no argument is given the old contents of the buffer
are undisturbed.

If you type any printing character it will get inserted.

The control key causes hitting a letter to do something.
The esc key (meta) is used to prefix a character and make it
also do something.

The delete key deletes to the left of the cursor.

The four basic directions are:

^P	previous line
^N	next line
^F	forward a character
^B	backwards a character (backspace (^H) also works)

To get around on a given line:

^A	go to the beginning of the line
^E	go the end of the line

To get rid of stuff:

^D	deletes the current character
^K	kills to the end of the line (or the new line itself if nothing is left)

To get to either end of the file:

meta-<	go to the top (or ^T)
meta->	go to the bottom (or ^Z)

To move up or down half a screen:

^V	move down half a screen

meta-V move up half a screen

To search forward for a string type:
^S
the string to find (which gets echoed near the bottom
of the screen (delete works here))
a backslash
If the string is null (ie ^S\) then the the previous search string is used.

To search backward the same syntax is used but the command is ^R.

To interface to the outside world two character combinations starting with ^X are required. All strings (filenames buffer names etc) are terminated by a CR.

^X^R Reads from a file (prompts for a file name)
^X^W Writes to a file (prompts for a file name)
^X^S Writes the buffer to the current filename but doesn't ask.
^X^C quit tv
^X^E Executes an external command (prompts for a command line).
^X^Y Prints out the entire buffer (like an "ht" in tint).
^X^X switches point and mark.
^XB Asks for a buffer name and makes that the current buffer. Each buffer has a mark and a filename of its own. There is one search string and kill area per invocation of TV.
^X^F Asks for a filename and reads that file into a buffer whose name is derived from the filename.
^X^O Asks for a buffername from a previous invocation of TV. It is assumed that the file "<username>.tvbuf.<buffer_name>" can be found in the process_dir. That buffer is made the current buffer; it should be identical to the old buffer.
^X^B List all the buffers currently known about.

To kill a large region of text use ^@ to set the "mark" to the current position then go to the end of the region to be deleted and use ^W. Both this command and the ^K command push the deleted text onto a kill stack. The top item on this stack may be copied into the buffer at the current position with the ^Y (Yank) command. The last item copied into the current buffer can be replaced by the previous item on the kill stack with a meta-Y. (this does a ^W a pop-last-kill-item and a ^Y).

For handling words and Lisp s-expressions:

meta-F move forward over word.
meta-B back up over word.
meta-D delete next word (pushed killed text like ^K).
meta-rub delete previous word (pushed killed text).

meta-^F move forward over s-expression.
meta-^B move backward over s-expression.

meta-^D delete next s-expression (pushed killed text).
meta-^rub delete previous s-expression (pushed killed text).

tvc is the MagicSix version of the transfer vector compiler, which is a method of creating command abbreviations by use of the do mechanism.

```
tvc filename
```

converts the file, filename.tvc into an object file called filename. filename.tvc contains lines of the form:

```
label:pattern
```

or

```
label:
line1;
line2;
line3;
end;
```

where label is the name of the entry point to be defined. This name is also added to the object segment unless there is some name conflict.

Typing the command:

```
label arg1 arg2 ...
```

then acts as if you had typed:

```
do pattern arg1 arg2 ...
```

See the documentation on do for more information.

Examples:

```
nd:pll &1 -nd
w:who
reattach:
io detach user_output;
io attach user_output syn_io user_io;
end;
note:mail sas -s &qr1
```