Multics Series
MS-99 Revision 1
August 1, 1981

MASSACHUSETTS INSTITUTE OF TECHNOLOGY INFORMATION PROCESSING SERVICES
ACADEMIC AND RESEARCH COMPUTING SERVICES · CAMBRIDGE MA 02139 (617) 253-1793

MULTICS BULLETIN ARTICLE REPRINTS

This memo contains diverse articles, which have appeared previously in the IPS bi-monthly newsletter, the Bulletin. Although we selected for reprinting only those articles which remain relevant and useful on the current Multics system, we cannot guarantee that all details are still accurate. The most up to date information can always be found in current Honeywell and IPS documentation and in the most recent issue of the Bulletin.

CONTENTS (8/1/81)

Active Functions
Learn about active functions and save yourself a lot of typing

Archives: Live and On Tape
A guide to the Multics "archive" command

Wanna Start Something?
Getting more out of Emacs by writing your own start_up routines

Excerpts from "Our Favorite Multics Questions"

How to copy someone else's segment

Likely causes of error messages: "storage condition", "record quota overflow", and "stack frame overflowed"

Hold everything; it's an interrupt!

How to join segments together

Help! I'm trapped in a question to the OLC!

Using "tape_archive"

Retrieving a lost segment

What happens if you just hang up

THINGS NOBODY TOLD YOU ABOUT YOUR START_UP.EC

by Richard Scott
reprinted from the March 1978 Bulletin

There are certain operations--such as adjusting printout to suit your particular terminal's characteristics, checking your Multics mail, and opening up the lines of communication to other Multics users--that you want to have performed just about every time you log in. To save you the effort of remembering to type (and typing correctly) the required commands each time, Multics permits you to put them into a segment in your home directory (the one where you find yourself immediately after logging in) and have the system automatically execute them for you. This segment must be named "start_up.ec".*

The start_up ec is usually created with a text editor (such as qedx), and contains two kinds of lines:

(1) Multics command lines, just as you would type them at the terminal; and

(2) special instructions, called control statements, that the exec_com command (which executes your start_up.ec) understands.

Let's consider some of the commands commonly found in start_ups. (For the purposes of discussion, we will assume a Multics login ID of "JQUser".)

KEEPING TUNED IN

One of the first things you see when you log in is the "message of the day". This message contains important information about the status of the system, system problems, schedule changes, etc. If you have no start_up, the message of the day is printed each time you log in. However, from the very existence of a start_up.ec segment in your home directory, the system assumes that you are taking explicit action to examine the message and ceases to print it automatically. Placing the print_motd command in your start_up.ec will handle this. The print_motd command keeps a copy of the last message of the day in a segment (called "JQUser.motd") in your home directory. Each time print_motd is invoked it compares the current message with the saved copy, and prints the message only if it has changed. This relieves you of having to read the same message over and over, when once is sufficient. Of course, it does cost $.02 a day to keep JQUser.motd around, so if you're pinching pennies, the command

```
print >doc>iis>motd.info 1 99
```

in your start_up will print the message of the day each time you log in, without incurring extra storage costs.

TAILORING TERMINAL OUTPUT

To take advantage of your terminal's capabilities and to make typing easier, you may want to use your terminal's set_tty command. This command has many options, documented in the MPM Commands and Active Functions and in the on-line info file (type "help set_tty"); we will consider only a few of the most general. The syntax of the command, for our purposes, is:

```
set_tty -modes OPTION1,OPTION2,...,OPTIONn
```

Note that there is a space before and after "-modes", and commas but no spaces between the options. For example, in the command:

```
set_tty -modes lfecho,polite,ll132
```

"lfecho", "polite", and "ll132" are options. Some particularly useful options are:

lfecho (line-feed echo) makes it possible to transmit typed lines by hitting only the RETURN key, instead of RETURN-LINEFEED. (This option is not necessary with Selectric-type terminals.)

ll132 (line length 132) tells Multics it can use the full carriage width of your terminal, not just 79 columns (the default). Of course, you can use a number other than 132, as appropriate. (For example, if your terminal beeps every time it passes column 72 and it's driving you crazy, use "ll71".)

---

*If you wish, you may also log in without executing these commands, by using the "-no_start_up" control argument to the login command.

tabs tells Multics that your terminal has settable tabs. Tab stops should be set at columns 11, 21, 31, etc., to correspond to Multics' view of the world.

^tabs tells Multics that your terminal does not have settable tab stops.

tabecho (used in combination with "^tabs") tells Multics to simulate tab stops on your terminal even though there are no hardware stops. When you depress the TAB key or CTRL-I, Multics prints the spaces required to move the print-head or cursor to where the next tab stop would be.

echoplex tells Multics that your terminal is permanently set in "full-duplex" mode and therefore cannot (without help) print what you type. If Multics will print on your terminal, but your typing causes no printing (and you have no switch labeled HALF-FULL or HDX-FDX), you need this option.

polite prevents lines you type from being interrupted by output from Multics (e.g., messages) unless you have taken more than 30 seconds to finish.

replay causes Multics to retype a line which does get interrupted, up to the point where you were stopped, so that you can continue it easily.

pl24 (page length 24) tells Multics that you have a soft-copy (CRT screen) terminal that can display only 24 lines at a time. After the 24th line, Multics prints "EOP" and refuses to display anything else until you hit FORMFEED or CTRL-L. This is to allow you time to read the screen before new lines drive the old ones off the top. The default is "pl0", which causes no interruption.

COMMUNICATING WITH OTHERS

The Multics mail facility is one feature of the system that you would do well to exploit fully, in spite of the slight overhead involved. It allows you to send and receive messages to and from other Multics users, communicate with the on-line consultant (type "help olc" to find out about this), and hear from the SysDaemons (those elusive, eternally-logged-in creatures that, among other things, print your files on the high-speed printer and back them up against future blunders and unforeseen disasters). To regularly check the messages in your mailbox (a segment in your home directory called "JQUser.mbx"), you can include the mail command in your start_up. If you don't already have a mailbox, the mail command will create one the first time it is invoked with no arguments. If you get lots of mail and don't always want to print it right away, you might include the line

```
&if [have_mail] &then &print You have mail.
```

instead of the mail command, in your start_up. This makes use of the exec_com control statements "&if", "&then", and "&print", and of the [have_mail] active function, and results in the printing of the message "You have mail." if there is anything in your mailbox. You may then examine the messages later at your leisure.

Experienced Multics users also communicate with one another via one- or two-line messages that are printed on the terminal immediately after being sent (via the send_message command) if the addressee is logged in and accepting messages. To accept messages, you may include the accept_messages command in your start_up. Two useful control arguments to this command are "-print", which prints any accumulated messages of this immediate type, and "-short", which replaces the sender's ID with "=:" if the same user sends several message lines in tandem. (After having accepted messages, you can use the defer_messages and immediate_messages commands to shut off and restore the printing of messages, if desired.) Methods of sending mail and messages, and further options available for these commands can be found in the MPM Commands and Active Functions and in the respective on-line info files.

MINDING YOUR PENNIES

You are probably concerned, to a greater or lesser extent, with how fast the computer is eating your funds. There are three commands you can include in your start_up to keep tabs on this:

```
resource_usage -total
```

prints out how much money you have used so far (exclusive of storage charges) in the current billing period prior to the current process, your limit (if any) for the period, and the amount you have used on your current project since you were registered.

```
estimate_bill -init
```

sets up the estimate_bill command to let you know how much you have used in the current login session (again, exclusive of storage charges) by shift and item. At any time later

in the session you can use estimate_bill to see how much has been spent so far. The estimate_bill command sometimes creates a segment, "JQUser.eb_data", for accounting purposes. Finally,

```
general_ready -set -time -inc_vcpu -inc_mem_units -inc_cost -level
```

changes your ready message (normally printed after the execution of each command line) to include the amount spent since the last ready message.

ALLEVIATING TYPIST'S CRAMP

The abbrev command allows you to define abbreviations for whole command lines or parts of command lines. You may use this facility by including the command in your start_up. Once this is done, you can issue special abbrev requests (documented in the MPM) to define abbreviations, which may then be used in your command lines. Properly used, the abbrev facility can save a lot of time and bother. It can also be used in combination with exec_com and the "do" command (a special command-line processor) to define, in effect, your own commands. It does create and use a segment called "JQUser.profile", which you should not delete unless you are prepared to lose all your abbreviations.

YOUR PRIVATE SECRETARY

The memo command may be included to keep a kind of calendar or checklist for tasks to be done. This facility (again, documented in the MPM) allows you to set up memos for yourself which are printed at a prespecified time, or repeated periodically. The memo command creates a segment called "JQUser.memo". This facility can prove valuable if you log in often and are prone to absent-mindedness.

PARAMETERS AND CONTROL STATEMENTS

In the start_up.ec, the special parameters "&1" and "&2" represent (and are replaced by) arguments passed to the exec_com by Multics when the start_up is executed. These arguments may be (depending on the situation, as noted):

&1	&2	
login	interactive	when you log in;
new_proc	interactive	when you issue the new_proc command, or after a fatal process error; or
login	absentee	when an absentee job you have submitted is logged in by Multics.

Then there are the special exec_com control statements:

&command_line off	tells exec_com not to print each command before it is executed. It is helpful to omit this instruction when debugging a start_up (or any exec_com);
&goto XXX	tells exec_com to continue executing, starting with the line labeled XXX;
&label XXX	where XXX is some character string to be used as a label; if XXX is one of the special parameters mentioned above, it is replaced by the respective argument as a label; and finally,
&quit	tells exec_com to stop executing, bringing the start_up to a halt.

Any line that begins with an ampersand (&) followed by one or more blanks is treated as a comment and ignored during execution.

PUTTING IT ALL TOGETHER

Let us now consider an example of how some of these commands can be combined into a start_up. (The numbers at the left are for reference and do not actually appear in the start_up.ec segment.)

```
 [1] & <<<<<<<<<<< A SAMPLE start_up.ec SEGMENT >>>>>>>>>>>
 [2] &
 [3] & ALWAYS EXECUTED:
 [4] &command_line off
 [5] abbrev
 [6] &goto &2
 [7] &
 [8] & FOR INTERACTIVE PROCESSES ONLY:
 [9] &label interactive
[10] accept_messages -print -short
[11] memo -brief
[12] estimate_bill -init
[13] &goto &1
[14] &
[15] & FOR INITIAL LOGIN ONLY:
[16] &label login
[17] set_tty -modes lfecho,ll132,^tabs,tabecho,polite,replay
[18] print_motd
[19] mail
[20] resource_usage -total
[21] &quit
[22] &
[23] & FOR A NEW PROCESS ONLY (NOT LOGIN):
[24] &label new_proc
[25] &quit
[26] &
[27] & FOR AN ABSENTEE LOGIN ONLY:
[28] &label absentee
[29] &quit
```

Lines 1, 2, 3, 7, 8, 14, 15, 22, 23, 26, and 27 are comment lines, used only to make the start_up more readable and understandable. The "&2" in line 6 is replaced by either "interactive" or "absentee", meaning that the next line executed will be line 10 or line 29, respectively. The "&1" in line 13 is replaced by either "login" or "new_proc"; the next line executed will therefore be line 17 or line 25, respectively.

This, of course, is only one example of what you can do in a start_up.ec. Many variations of the commands discussed, as well as other commands and exec_com controls, can be utilized to precisely tailor the Multics environment to your individual needs.

MULTICS TIME-SAVERS: A BEGINNER'S GUIDE TO (PERSONAL) ENERGY CONSERVATION

Adapted with permission from an article by Alicia Towster in Two Bits Worth, newsletter of the University of Southwestern Louisiana Computing Center (February-March 1978). Appeared in the May 1978 Bulletin.

Are you worn out from typing lengthy pathnames, repetitious command sequences, or tedious lists of control arguments? You are? Well, if you're willing to use your memory, your ingenuity, and possibly some of your computer funds, you can minimize keyboard cramp.

First, there are addnames. Addnames are additional names which can be given to segments or directories. This permits you to keep, on each entry, a long descriptive name (up to 32 characters) to help you to remember what it is, and a short convenient name which is what you actually type. You add names to entries by using the add_name command. For example, user HTudor has a segment named catherine_of_aragon. Once he has executed the command:

```
an  catherine_of_aragon  kate
```

he can refer to the segment as simply "kate". (He will see both names when he lists the directory--addnames appear right under the "primary" name--but he can tell they are addnames rather than other segments, because quota and access information is given only with the primary name.)

The reason HTudor can type "an" instead of "add_name" is because "an" is an addname of "add_name". (You may have to read that sentence twice.) Addnames are used throughout the Multics system; many are selected and added by systems designers (who don't much care for extra typing either). Clearly you can save a lot of time and nerves by learning the choices they have made. The surest way to find the addnames of a particular entry is to move to the directory which contains it and do a "list" (or rather, "ls"). For standard Multics commands, you can check the Table of Contents of the MPM Commands and Active Functions (AG92); and on-line info files give addnames in the first section of each writeup.

Addnaming your programs is just slightly more complicated. HTudor has a source program called "ann_boleyn.pl1" which he decides to addname "ann.pl1". (The compiler insists on the ".pl1" suffix for addnames, too.) He then compiles the program (referring to it by the addname), and the object program is stored in a segment called "ann". But alack! When he tries to execute "ann", Multics refuses, choking on the entry point defined by the label which begins the source program. We often ignore the entry point name, since, unless told otherwise, Multics assumes it is the same as the segment name. HTudor can manage by typing "ann$ann_boleyn", but this is not very graceful. A far better solution is to plan a head (!) and put two labels at the beginning of the source program, thus:

```
ann_boleyn: ann: procedure;
```

System designers have also defined standard abbreviations for control arguments. (Why type "-brief" when "-bf" works just as well?) You can see a list of many of these by typing "help control_arguments". Invest some time and dip more deeply into Honeywell documentation, scanning for words like "default" and "optional" (maybe you can avoid typing some control arguments altogether), and the note "Star convention allowed." The star convention enables you to reference several entries at once, and is described in Section III of the MPM Reference Guide (AG91), and in an on-line info file. (Type "help star_names".)

Even taking advantage of every available addname, you may still find yourself typing some rather long pathnames. If you expect to be using the same pathname repeatedly, you should probably create a link.* User HTudor might type:

```
link  >udd>Britain>HTudor>TskTsk>wives>wives.archive
```

Thereafter, when he is in the directory in which he created the link, he need only type "wives.archive" to reference that segment. Links are handy "pointers" to other segments, and do not consume your quota nearly as fast as real live segments do. However, you can't take them with you, (e.g., when you change working directories or log into a different project); links stay in the directory in which they were created. You may, however, link to a link.

If you have at least one record to spare, you can use the portable energy saver--abbrevs. Abbrevs are your own abbreviations for the kinds of processing you do; they can stand for anything you want, from a few characters to a deluge of sequential commands. If HTudor

---

*See also the article "A Beginner's Guide to Links" in the May 1977 Bulletin.

wants to use abbrevs, he types:

```
abbrev
```

The very first time he does this, a segment called "HTudor.profile" is created for him in his home directory to contain any abbreviations he wishes to define. Most users will want to invoke the "abbrev" command (addnamed "ab"--getting confused?) as part of their start_up.ec.*

Giving the "abbrev" command tells Multics to invoke the abbreviation processor; every command typed subsequently is examined to see if it begins with a period (meaning it is a special request to manipulate abbreviations) or contains an abbrev. Command lines containing abbrevs are first "expanded"--the abbrev is replaced by what it stands for--and then executed. This process may, on occasion, produce unexpected results. For example, "ro" might seem like a nice abbreviation for "runout"; however, the command "ro" (for "revert_output") would then be expanded to "runout", which Multics would find incomprehensible as a command. To guard against this kind of mishap, some users decide to define abbrevs that include some special character (e.g., a colon), or use capital letters (since standard Multics commands are always lower-case).

HTudor prefers capitals, and is now ready to define his first abbrev:

```
.ab  Behead  delete ann
```

The ".ab" means add to the profile an abbrev to be expanded only when encountered at the beginning of a command line (or immediately following a semicolon).

HTudor suddenly realizes that he wants his abbrev to work from any working directory, so he corrects it by typing:

```
.abf  Behead  dl >udd>Britain>HTudor>ann
```

using the absolute pathname of the segment "ann". The "f" means force the abbrev to replace any existing abbrev of the same name. HTudor also defines:

```
.ab  Annul  cwd >udd>Britain>HTudor>TskTsk>wives;da ann_of_cleves HTudor.*.*
.a   GIRLS  (MTudor ETudor)
```

Since GIRLS is not something that occurs at the beginning of a command, he defines it using ".a" instead of ".ab". Now he can type such commands as:

```
Annul; sm GIRLS.Britain  See me ASAP!
```

Some other useful abbrev requests are ".l" (to list some or all of one's abbrevs) and ".d" (to delete abbrevs). A ".q" tells abbrev to quit using abbrevs. As you define progressively more elaborate abbrevs, you will find the "do" command very helpful. (See the description of "do" in the MPM Commands and Active Functions, or type "help do". If you know a hardened Multician, you might ask him/her if you may go browsing through his/her abbrevs. (Most will be flattered to oblige, and you can learn a lot just by looking.) To do this, have your friend set you "read" access to his/her ".profile" segment. Then type:

```
.u  >udd>FriendProjectID>FriendPersonID>FriendPersonID
```

The ".u" tells Multics to use the profile segment whose pathname you have given. You may then use the ".l" (list) request to peruse the abbrevs contained in this profile segment. To return to your own abbrevs, type ".u" again, followed by a carriage return.

* * * * *

None of these energy-saving techniques are directly useful when inputting text with an editor. However, a set of recently-installed Multics commands permits another kind of personal abbreviations to be used in this manner. The "Speedtype" commands, as they are known, provide a facility for expanding input text automatically. They should be used cautiously--experienced typists have, on occasion, been known to produce errors of a somewhat bizarre nature ("Speedtypos"?). Use of Speedtype requires at least three extra records and quite a bit of patience, but can greatly ease the tedium of large typing jobs, and in many ways is far safer than defining--and remembering--one's own "abbreviations" for input text (for later expansion by an editor's global-change request). For more information on Speedtype, consult the WORDPRO Reference Guide (AZ98), now available for purchase in the IPS Publications Office (Room 39-484), or for reference in the Reading Room (39-430).

---

*See also the article "Things Nobody Told You About Your start_up.ec" in the March 1978 Bulletin.

YOU DON'T HAFTA ANSWER RIGHT AWAY....

by W. Olin Sibert
reprinted from the January-February 1979 Bulletin

Many Multics commands will ask you questions when they require some extra information in order to execute. These questions (and the answers you supply) are all handled by the same subroutine (command_query_), and follow certain system-wide conventions. A new feature now allows to you "escape" command_query_ if you need to know more information yourself before you reply. Just begin your "answer" with two periods (".."). The rest of the line will be passed directly to the Multics command processor rather than being returned to the program that called command_query_. After your command has executed, command_query_ will prompt you again, printing only "Answer:" rather than repeating the question, (since the question can be quite long, and you have already seen it once). At this point you may type in your answer, or "escape" the query again if you wish (procrastinating as many times as you see fit). This allows you quite a bit of added flexibility in answering commands' questions: if the question is unexpected and/or makes little or no sense to you at first, you can issue a command or two to try to find out what prompted it. The following example illustrates. User typing is preceded by an arrow (=>).

```
=>  delete **.pl1

    delete: calliope.pl1 is protected.  Do you want to delete it?  =>  ..list

    Segments = 3, Lengths = 7

    re     2  calliope
    r w    4  calliope.pl1
    r w    1  calliope.list

    Answer:  =>  ..print_wdir
    >udd>CIRCUS>Elephant>work
    Answer:  =>  ..repeat_query

    delete:  calliope.pl1 is protected.  Do you wish to delete it?  =>  no
    r 1409.2 0.480  17.426
```

As shown above, the "repeat_query" command can be useful in this type of situation, if you want to see the question again--e.g., if it has scrolled off the top of a CRT-type terminal screen.

Another useful capability provided by this general-purpose question routine is the "answer" command, which allows you to supply preset answers for questions that commands will (or might) ask you. This is primarily useful in abbrevs, although you may also find it helpful at command level. For example, suppose you want to write an abbrev that moves a segment from one directory to another, and then creates a link in the old directory pointing to the segment's new location. If you were sure it would always be okay to delete any segment that already existed in the new location, you could define an abbrev like this:

```
.ab  mvlk  do "answer yes -brief  move &1 &2; link &2 &1"
```

The "answer" command in the abbrev will "catch" any question asked by the move command (e.g., an "are-you-sure-you-want-to-delete" query brought about by a turned-on safety switch or a name duplication), and answer it "yes". The "-brief" control argument says that no such question will even appear on the terminal. (If "-brief" is omitted, the question and your prespecified answer are typed out automatically.)

A second example will demonstrate how the "answer" command can be used effectively from command level. Suppose you have an archive named "xanadu.archive" containing program source segments, some of which you have extracted and modified. You now want to recompile everything from the archive; to do this you must extract all its components. Since the "archive" command, when asked to extract components, will not overwrite (delete) an existing file in the case of a duplicate name without asking, you could get "archive" to extract only those components you have not already extracted by using the command:

```
answer no -brief  archive  x  xanadu
```

This would automatically (and silently) reply "no" to the "archive" command when it asks you if you want the extract operation to delete your new, already-extracted segments. (Unfamiliar with the Multics "archive" facility? See the Multics Programmers' Manual (MPM): Commands and Active Functions, AG92, for information and examples.)

The full syntax of "answer" allows for considerable versatility: other control arguments permit you to specify how many times a given answer should be supplied (default: as many times as there are questions asked), or to specify a sequence of answers. For instance,

you can set up a command line to supply one answer to the first question asked, supply another answer to the second, give still another answer to the third, fourth, and fifth questions, and ask you for anything else it wants after that. For a complete explanation of these options, see the MPM, or type "help answer".

WRITING MULTICS COMMANDS

PART I: OBTAINING ARGUMENTS AND REPORTING ERRORS

by Richard Scott
reprinted from the January-February 1979 Bulletin

Part of the beauty of programming on Multics is that all compiled "main" programs are commands.<5> That is, you can invoke them simply by typing their names--no loading, no linking, no muss, no fuss. And you can get information into programs by means of language I/O statements. You can even make these programs look like commands with arguments by using combinations of exec_coms and abbrevs. Or you can do the same thing in the PL/I language by writing programs as subroutines with character (*) parameters. These constructs may be satisfactory for a program only you are going to use. But as you begin to write programs for wider use, in combination with standard Multics commands, what you really want is the efficiency, control structure, and error-handling capability of a true Multics command written in the PL/I language.<6>

Unfortunately, as far as I know, no existing manual clearly explains how to write Multics commands. People find out how by a combination of perusing the MPM Subroutines (AG93), looking at the source code of existing commands, and asking people who already know how. Through this series of articles, I hope to clear up some of the mystery, and show you how to write your own commands with no Multics "guru" present. The only tools you will need are the MPM Reference Guide (AG91), the MPM Subroutines (AG93), and a knowledge of PL/I (worth obtaining in its own right).

ERROR MESSAGES

Since one of the big drawbacks of many programs is their inability to recover gracefully from human error, let's start by talking about error handling. Commands report errors by calling the subroutines "com_err_" and "com_err_$suppress_name". These subroutines take arguments of varying number and type, and must therefore be declared with "options (variable)", e.g.,

```
declare  com_err_ entry options (variable);
```

The first two arguments are a Multics system status code and the name of the command. The status code may be one returned by a standard Multics subroutine you have called, the value of an external variable in an error table, or zero. A list of most of the external variables in the Multics system error table (>system_library_1>error_table_) is given under "Handling Unusual Occurrences" in Section VII of the MPM Reference Guide, along with the messages that "com_err_" prints when passed the value of each variable. All of these variables are declared "fixed binary (35) external". Therefore, if your program "do_good_stuff.pl1" contains the declaration:

```
declare  error_table_$noarg fixed bin (35) external; /* missing arg */
```

then the statement:

```
call com_err_ (error_table_$noarg, "do_good_stuff");  /* report error */
```

would cause the message:

```
do_good_stuff: Expected argument missing.
```

to be printed on the terminal. Such a message is printed on the "error_output" I/O stream, so that you see error messages even when other command output to the terminal (printed on the "user_output" I/O stream) has been diverted elsewhere (e.g., via a "file_output" command).

Using the system status codes has the advantage of providing consistency with other Multics commands: you need to memorize the meaning of fewer messages. However, if none of the standard status codes' messages seem appropriate for your situation, call "com_err_" with a status code of zero and and give your own message as the third argument. The statement:

```
call com_err_ (0, "do_good_stuff", "Try reading the documentation."); /* advise
     user */
```

<5> In a larger sense, all programs are subroutines--commands are actually called by the command processor; but we won't worry about that.

<6> Such a command may, of course, call FORTRAN subroutines for computational tasks.

would print:

```
do_good_stuff: Try reading the documentation.
```

when executed. (A more elegant way of doing the same thing is to compile your own private error table; that, however, is beyond the scope of this article.) If the status code is not zero, the third argument is appended to the message the code generates, as we'll see later. This argument is actually an "ioa_" control string, which may contain variable "keys" into which the fourth and subsequent arguments to "com_err_" are formatted and substituted. (See the description of "ioa_" in the MPM Subroutines.)

Generally, only commands call "com_err_"; subroutines do not. A subroutine instead returns a status code to the calling command, which then calls "com_err_". This allows you to call the same subroutine from several commands, and decide (on a case-by-case basis) whether and in what format to print an error message. It also relates the message more directly to the user action which led to the difficulty.

ARGUMENTS

We must also concern ourselves with getting the arguments given on the command line into the program. A thorough discussion of how to interpret these arguments is beyond the scope of this article. Some conventions to keep in mind include:

(1) Control arguments, i.e., keywords that modify the behavior of the command or indicate the interpretation of a single immediately-subsequent argument (e.g., "-brief" or "-output_file XXX"), begin with a hyphen and may be given in any order.

(2) Informational arguments, i.e., variables such as filenames, may be required to appear in a particular order.

(3) Intermixing informational and control arguments should not change the way the command behaves.

To find out how many arguments were supplied to your command, call the subroutine "cu_$arg_count", which must be declared:

```
declare  cu_$arg_count entry (fixed binary);
```

If you then declare a variable like:

```
declare  nargs fixed binary; /* number of arguments supplied */
```

the statement:

```
call cu_$arg_count (nargs); /* find out how many args */
```

will set "nargs" equal to the number of arguments supplied.<7>

Suppose your program expects at least one argument. If "nargs" comes out zero, you would want to return an error message. In such a case, the convention (these days) is to print a message describing the syntax of the command, e.g.,

```
if nargs = 0 then do;                  /* not enough arguments */
   call com_err_ (error_table_$noarg, "do_good_stuff",
      "^/^5xUsage: do_good_stuff path");
   return;                             /* tell user & abort command */
end;
```

prints a message like this:

```
do_good_stuff: Expected argument missing.  Usage: do_good_stuff path
```

The usage message indicates (a) that an argument is expected, and (b) that it should be a pathname. The "return" statement in the command procedure shown causes the command to be aborted.

If command arguments were provided, get their values by calling "cu_$arg_ptr", which is declared:

```
declare  cu_$arg_ptr entry (fixed binary, pointer, fixed binary,
     fixed binary (35));
```

<7> In general, entry points to "cu_" (command utility) deal with manipulating the command environment.

If you then make the following declarations:

```
declare  argno fixed binary;              /* argument sequence number */
declare  argp pointer;                    /* pointer to argument */
declare  argl fixed binary;               /* length of argument */
declare  arg character (argl) based (argp);
                                          /* argument */
declare  code fixed binary (35);          /* system status code */
```

executing the statements:

```
          argno = 1;                      /* looking for first argument */
          call cu_$arg_ptr (argno, argp, argl, code);
                                          /* get argument */
```

will set the value of "arg" to the first argument of the command. The variable "code" is a system status code. This is most often set to zero (meaning "no error") or to the value of "error_table_$noarg", if, for example, you have asked for the fifth argument (argno = 5) when only three were supplied. Proper use of "cu_$arg_count" will, of course, prevent this.

Suppose your program requires one pathname argument and may take an optional control argument, "-brief", or its abbreviation, "-bf".<8> You could use a sequence like this to process the arguments:

```
 1  declare (error_table_$badopt,            /* bad control argument */
 2           error_table_$noarg)             /* missing argument */
 3           fixed binary (35) external;
 4  declare  brief_flag bit (1) aligned initial ("0"b);
 5                                           /* on if "-brief" supplied */
 6  declare  pathp pointer initial (null ());
 7                                           /* pointer to pathname */
 8  declare  pathl fixed binary;             /* length of pathname */
 9  declare  path char (pathl) based (pathp);
10                                           /* pathname */
11  declare (null, substr) builtin;
12
13  /* Get arguments (cu_$arg_count already called above) */
14        do argno = 1 to nargs;
15           call cu_$arg_ptr (argno, argp, argl, code);
16                                           /* get argument */
17
18           if code ^= 0 then do;           /* couldn't get argument */
19              call com_err_ (code, "do_good_stuff", "Argument ^d.", argno);
20                                           /* report problem */
21              return;                      /* abort command */
22           end;
23
24           if substr (arg, 1, 1,) = "-" then do;
25                                           /* control arguments start with "-" */
26              if arg = "-brief" | arg = "-bf" then do;
27                 if brief_flag then do;    /* already specified */
28                    call com_err_ (0, "do_good_stuff",
29                       "Redundant control argument.  ^a");
30                                           /* report error */
31                    return;                /* abort command */
32                 end;
33
34                 brief_flag = "1"b;        /* show "-brief" specified */
35              end;
36              else do;                     /* unknown control argument */
37                 call com_err_ (error_table_$badopt, "do_good_stuff", "^a", arg);
38                                           /* report error */
39                 return;                   /* abort command */
40              end;
41           end;
42           else do;                        /* pathname argument */
43              if pathp ^= null() then do; /* already specified */
44                 call com_err_ (0, "do_good_stuff", "Redundant argument.  ^a", arg);
45                                           /* report error */
46                 return;                   /* abort command */
```

<8> Appendix A of the MPM Subsystem Writers' Guide lists names and conventional abbreviations of commonly-used Multics control arguments. Choosing applicable names from this list and implementing the same abbreviations reduces the probability of errors in the use of the command by people already acquainted with Multics.

```
47              end;
48
49              pathp = argp;               /* save pointer and */
50              pathl = argl;               /* length of pathname */
51           end;
52        end;
53
54        if pathp = null () then do;       /* no pathname specified */
55           call com_err_ (error_table_$noarg, "do_good_stuff", "Pathname required.");
56                                          /* report error */
57           return;                        /* abort command */
58        end;
```

Use comments to explain what is happening in your program. The "^d" in "Argument ^d." (line 19) is an "ioa_" substitution key for formatting fixed-point values for decimal printing--in this case the value of "argno". An error encountered in getting the argument when argno = 2 therefore produces a message like:

```
do_good_stuff: ....  Argument 2.
```

Here we can't print the actual argument in the error message; the nonzero error code may indicate that it was unobtainable. But later (line 37), when the argument has been successfully obtained, we can use it in the error message so that the user knows exactly what has tripped things up. The "^a" is an "ioa_" substitution key for formatting character values, such as the next argument to "com_err_", "arg". We use "^a" as the third argument only to circumvent the (admittedly remote) possibility that the value of "arg" might be a valid "ioa_" control string; otherwise, the third argument could be specified as simply "arg". If the control argument "-fred" were supplied on the command line, the error message generated here would read:

```
do_good_stuff: Specified control argument is not implemented by this command.
     -fred
```

Although one is tempted to overlook a multiple specification of the same control argument, such a case can indicate a typing mistake. It's usually safer, therefore, to report it as an error (lines 27-32). And if you might have many cases of the sequence:

```
if ...
   then do;
   call com_err (...);
   return;
end;
```

it's reasonable to replace them with calls to one or more internal procedures which call "com_err_" and then perform a non-local "goto", transferring control to a "return" statement in the main procedure.

Once you have the arguments, which are all character values, you may need to transform some of them into numeric values. Do this with a sequence like:

```
declare  error_table_$bad_conversion fixed binary (35) external;
declare (conversion,
         size) condition;
declare  number fixed binary;             /* a number */
declare  binary builtin;

      on conversion begin;                /* handler for invalid number */
         call com_err_ (error_table_$bad_conversion, "do_good_stuff", "^a", arg);
                                          /* report error */
         goto exit;                       /* abort command */
      end;

      on size begin;                      /* handler for too-large number */
         call com_err_ (0, "do_good_stuff",
            "Magnitude of ^a greater than 131071.", arg);
                                          /* report error */
         goto exit;                       /* abort command */
      end;

      number = binary (arg, 17, 0);       /* convert the argument */

      revert conversion, size;            /* disable the handlers */
         .
         .
         .
exit:
      return;                             /* abort command */
```

The "on" statements and "begin" blocks are condition handlers for the "conversion" and "size" conditions that might be raised if the value of "arg" could not be properly transformed into a numeric value. For example, if "arg" were "2%" instead of "25" (due to a typo), the call to "com_err_" in the "begin" block for the "conversion" condition would print the message:

```
do_good_stuff: Error in conversion.  2%
```

A well-written command provides handlers for a wide range of possible error conditions, seldom (if ever) letting control pass to the system handlers. Even cleaner (though more complex) approaches attempt to "catch" illegal values before they raise error conditions.

WRITING MULTICS COMMANDS

PART II: ACCESSING AND CREATING SEGMENTS

by Richard Scott
reprinted from the March-April 1979 Bulletin

Last time we talked about obtaining arguments and reporting errors. These operations are fundamental to all commands--even those that get all their input from the terminal. For commands that process a lot of input data or produce a lot of output, the terminal is not usually a suitable I/O device. In this article we will talk about the most efficient way of doing I/O to and from Multics segments.

You could, of course, use PL/I I/O statements to read data from segments and write results into them. For operations on databases bigger than (roughly) one million bytes, this may be the easiest approach.<9> To handle smaller amounts of data, however, it's more efficient to make use of Multics' virtual memory and treat segments' contents as based variables in the program.

In standard PL/I, a pointer is a locator value used to designate a generation of storage of a program variable. Since, in Multics, all generations of storage occur in segments, Multics PL/I extends the pointer concept somewhat. In Multics, the internal representation of a pointer is a combination of a segment number and an offset within the corresponding segment, commonly written "segno|offset"--e.g., 244|104, where "244" and "104" are octal numbers.

In a given process,<10> the Multics supervisor assigns a unique (for that time and process) segment number to each segment referenced. Asking the supervisor to assign such a number to a given segment is called initiating the segment or making the segment known to the process. And asking the supervisor to disassociate the segment number from a given segment, so that the number may be re-used, is called terminating the segment, or making the segment unknown.

GETTING A POINTER TO A SEGMENT

In Multics PL/I, as in standard PL/I, you can obtain a pointer to an existing generation of storage by using the "addr" or "pointer" builtin functions, or to a new generation of storage by using the "allocate" statement. However, in Multics PL/I you can also get a pointer to a named segment by calling the supervisor. The subroutine most commonly used for this purpose is "hcs_$initiate_count",<11> which must be declared:

```
declare  hcs_$initiate_count entry (character (*), character (*), character (*),
         fixed binary (24), fixed binary (2), pointer, fixed binary (35));
```

If you declare this and the following variables:

```
declare  bit_count fixed binary (24);    /* segment length in bits */
declare  seg_ptr pointer initial (null ());
                                         /* pointer to base of segment */
declare  code fixed binary (35);         /* system status code */
```

then executing:

```
call hcs_$initiate_count (">udd>ARK>Noah", "hippo.data", "", bit_count, 1,
   seg_ptr, code);                       /* get pointer to base of segment */
```

sets "seg_ptr" to the location of the segment "hippo.data" in the directory ">udd>ARK>Noah"--i.e., to ">udd>ARK>Noah>hippo.data", provided that the segment exists and you have at least "r" ("read") access to it. It also sets "bit_count" to the length of the segment, in bits.

If the segment is not already initiated, this call initiates it. The null string ("") passed as the third argument tells the supervisor to associate a null reference name with the segment. If you pass a non-null character string here, the supervisor associates that string with the segment as a reference name. A segment may have many reference names, but

---

<9> Another approach would use the entry points to "msf_manager_", documented in the MPM Subsystem Writers' Guide (AK92).

<10> "Process" refers to the complex of virtual address space and programs in execution associated with a given Multics user at a given time. Except that it calls upon the Multics supervisor to perform certain functions, it is very much like a "virtual machine".

<11> Supervisor calls are usually entry points of "hcs_" ("hardcore supervisor").

each non-null reference name in a process may be associated with only one segment at a time. Other programs in the same process (including the dynamic linker) may then refer to the segment by one of its reference names until that reference name (or the segment) is terminated. For most purposes, however, a null reference name will suffice.

The constant "1", passed as the fifth argument, tells the supervisor to give you a pointer to the original segment. If you pass "2" instead, the supervisor creates a temporary copy of the segment in your process directory and sets "seg_ptr" to that segment's location.

You might think that you have an error if the supervisor returns a nonzero value in "code". However, that isn't necessarily the case. For example, if "hippo.data" has already been initiated, the supervisor returns the value of "error_table_$segknown", whose corresponding message (as printed by "com_err_"; see Part I of this series, in the January-February issue) is "Segment already known to process." However, "seg_ptr" is still set to the location of "hippo.data". This is why we call the value returned in "code" a status code, not an error code. The test for whether initiation has failed is the value of "seg_ptr": if it is null, then the supervisor could not initiate the segment for you, and you should call "com_err_" with the value of "code" to find out why. (See the example under INTERPRETING PATHNAME ARGUMENTS, below.)

Once you have a pointer to "hippo.data", you can use a based variable to overlay the storage however you like. If there are already data in the segment, the value of "bit_count" will tell you how many bits of data there are. However, the bit count of a segment may not necessarily reflect the actual contents of the segment. It's kind of like the label on a package, which may claim there are 7 ounces of potato chips inside, when in fact there may really be only 5 (or even 9--a somewhat rarer case with potato chips). But we assume that this value is correct. It is the duty of any command that changes the contents of a segment that may be used by other commands to reset that segment's bit count correctly. So if you expect the segment to contain ASCII (character) data, you can calculate the number of characters from the bit count by dividing by 9 (since there are 9 bits in each character on Multics). For example, if we declare the following variables:

```
declare  nchars fixed binary (20);      /* number of characters in segment */
declare  hippo_string character (nchars) based (seg_ptr);
                                        /* segment contents as one long */
                                        /* character string */
```

then executing:

```
nchars = divide (bit_count + 8, 9, 20, 0);
                                        /* get number of characters in segment */
```

lets us treat the contents of "hippo.data" as one long character string, which we can then manipulate with the PL/I string builtin functions such as "index" and "substr". The variable "nchars" is declared with a precision of 20 bits because the maximum number of characters a segment can hold is between 2^{19} and 2^{20}. We used the "divide" builtin function instead of just saying:

```
nchars = (bit_count + 8)/9;
```

because the builtin function is more efficient.

On the other hand, if you expect the segment to contain an array of aligned, single-precision, floating-point, binary numbers, the declarations and executable code are analogous:

```
declare  nvalues fixed binary (18);     /* length of array */
declare  hippo_array (nvalues) float binary based (seg_ptr);
                                        /* segment as array of numbers */

nvalues = divide (bit_count + 35, 36, 18, 0);
                                        /* get number of words in segment */
```

Here, the largest number of 36-bit words a segment can hold is between 2^{17} and 2^{18}. You can do the same sort of thing with any imaginable type of data; just keep in mind that in most cases (unless it has been set otherwise) the maximum length of a segment is 261,120 36-bit words (9,400,320 bits). Of course, with complicated structure variables, the calculations get worse.

Now you can read the contents of the segment by referring to the associated based variable on the right-hand side of an assignment statement:

```
x = hippo_array (7);
```

or as an argument to a function or subroutine:

```
call print_values (hippo_array);
```

And you can write into the segment by referring to the associated based variable on the left-hand side of an assignment statement:

```
hippo_array (11) = x * y;
```

or as arguments to subroutines. The supervisor takes care of bringing the data into main memory as it is needed, and you can forget about "read" and "write" statements.

VERIFYING ACCESS

Of course, to read from or write into a segment, you must have the appropriate access to it.<12> If you do not, the supervisor raises the "no_read_permission" or "no_write_permission" conditions. Many commands assume that, if you have any access at all to a segment (so that "hcs_$initiate_count" returns a pointer to it), you have at least "r" ("read") access; this is usually true. However, "w" ("write") access is more likely to be lacking, so you should check for it before trying to change the contents of the segment. You can do this by calling "hcs_$fs_get_mode" as follows:

```
declare  hcs_$fs_get_mode entry (pointer, fixed binary (5), fixed binary (35));
declare  mode fixed binary (5);          /* access mode */
declare  write_ok bit (5) internal static initial ("00010"b) options (constant);
                                         /* bit pattern for "w" access */

  call hcs_$fs_get_mode entry (seg_ptr, mode, code);
                                         /* get access mode */

  if code ^= 0                           /* can't get mode */
  then...                                /* report problem somehow */

  if ^(bit (mode, 5) & write_ok)         /* no "w" access */
  then...                                /* report problem somehow */
```

The call to "hcs_$fs_get_mode" sets "mode" to a fixed binary value whose bit pattern represents the access mode to "hippo.data" for the person using your command. A 2's bit of "1" (e.g., a "mode" of 00010b, 01010b, or 01110b) indicates that the calling process has "w" access. We use the "bit" builtin function to transform the binary number to a bit string and "and" this bit string with the bit string constant "write_ok", in which only the value of the 2's bit is "1". The result of the "and-ing" is a string of all zero bits, "false", unless the value of the 2's bit in the bit-string representation of "mode" is also "1". The person using your command has "r" access to "hippo.data" if the 8's bit of the bit-string representation of "mode" is "1"; you can make an analogous test for that by declaring and using a constant like the following:

```
declare  read_ok bit (5) internal static initial ("01000"b) options (constant);
                                         /* bit pattern for "r" access */
```

The status code with which to call "com_err_" in the case of insufficient access is the value of "error_tablm_$moderr" (whose associated message is "Incorrect access on entry."). You should include the full pathname of the segment in the error message, as we will illustrate later.

CREATING SEGMENTS

So far we have discussed only segments that already exist. You can also ask the supervisor to create segments by calling "hcs_$make_seg" as follows:

```
declare  hcs_$make_seg entry (character (*), character (*), character (*),
         fixed binary (5), pointer, fixed binary (35));

  mode = 01010b;                         /* request "rw" access mode */
  call hcs_$make_seg (">udd>ARK>Noah", "chicken.data", "", mode, seg_ptr, code);
                                         /* make and/or get pointer to segment */
```

Here the first three arguments are the same as those described for "hcs_$initiate_count" above. And "mode" is declared the same as for the call to "hcs_$fs_get_mode" above; but in this case it is an input argument indicating what you want the access mode of the segment to be for the process that creates it. In the example above, the values of both the 8's bit and the 2's bit are "1", indicating "rw" access.

If the value of "code" is set to zero, the supervisor has created the segment and set the

---

<12> See Section VIII of the Multics Introductory Users' Guide (AL40), or Section IV of the MPM Reference Guide (AG91), for an explanation of access modes.

access as requested. However, even if "code" is nonzero, the value of "seg_ptr" may still be non-null. This happens when either (a) there is already a segment named "chicken.data" in ">udd>ARK>Noah" or (b) the reference-name (third) argument to "hcs_$make_seg" was non-null and a segment was already initiated with that name. We will assume that the reference-name argument was null; then the value of "code" will be either the same as "error_table_$namedup" (if the segment exists) or "error_table_$segknown" (if the segment exists and has been initiated). In either case, you must decide whether to reject the segment if you can't get a new one, or use the one that's already there. If the latter, you can get your access mode by calling "hcs_$fs_get_mode", and the segment's length by calling "hcs_$status_mins":

```
declare  hcs_$status_mins entry (pointer, fixed binary (2), fixed binary (24),
         fixed binary (35));
declare  type fixed binary (2);          /* type of directory entry */
declare (error_table_$namedup,           /* entryname already in directory */
         error_table_$segknown) fixed binary (35) external;
                                         /* segment already known to process */

  call hcs_$make_seg (">udd>ARK>Noah", "chicken.data", "", mode, seg_ptr, code);
                                         /* make and/or get pointer to segment */

  if seg_ptr = null ()                   /* can't get a segment */
  then...                                /* handle the error */

  if code = error_table_$namedup | code = error_table_$segknown
                                         /* segment already exists */
  then do;                               /* get needed information */
     call hcs_$fs_get_mode (seg_ptr, mode, code);
                                         /* get access mode */
        .                                /* handle errors */
        .
        .
     call hcs_$status_mins (seg_ptr, type, bit_count, code);
                                         /* get length */
        .                                /* handle errors */
        .
        .
  end;
  else...                                /* handle the error */
```

TEMPORARY ("SCRATCH") SEGMENTS

To get temporary segments for scratch space, call "get_temp_segments_". For example, if you need three temporary segments, you can get them as follows:

```
declare  get_temp_segments_ entry (character (*), (*) pointer,
         fixed binary (35));
declare  temp_seg_ptrs (3) pointer initial ((3) null ());
                                         /* array of pointers */

  call get_temp_segments_ ("do_good_stuff", temp_seg_ptrs, code);
                                         /* get scratch segments */

  if code ^= 0                           /* can't get segments */
  then...                                /* handle the error */
```

The first argument to "get_temp_segments_" must be the name of your command (here, "do_good_stuff"). You tell "get_temp_segments_" how many segments you need by the extent of the pointer array "temp_seg_ptrs". This must be an array, even if you want only one segment:

```
declare  temp_seg_ptrs (1) pointer initial ((1) null ());
                                         /* array of extent 1 */
```

The advantage of using "get_temp_segments_" instead of "hcs_$make_seg" is that, once you are finished with the segments, another program may reuse them (see CLEANING UP, below) and thereby avoid the cost of creating new segments.

MODIFYING SEGMENTS SAFELY

One important use of temporary segments is for making extensive modifications to permanent segments. If a command were to make massive changes directly to a permanent segment, a program error or system crash might leave the segment's contents mangled or otherwise useless. You can considerably reduce the likelihood of this kind of catastrophe by copying the contents of the permanent segment into a temporary segment, modifying the temporary

segment, and then copying the new temporary segment back into the permanent segment, either upon completing all the modifications or at strategic points in the modification process. Thus, if something goes wrong, the contents of the permanent segment are left in a manageable state. You can copy the permanent segment into the temporary segment with a single assignment statement, e.g.:

```
temp_seg_ptrs (1) -> hippo_array = seg_ptr -> hippo_array;
                                        /* copy segment contents to */
                                        /* temporary segment */
```

or after making the modifications:

```
seg_ptr -> hippo_array = temp_seg_ptrs (1) -> hippo_array;
                                        /* copy modifications back into */
                                        /* original segment */
```

Truncate the original segment and reset its bit count (see CLEANING UP, below) after each copy from the temporary segment to the original.

INTERPRETING PATHNAME ARGUMENTS

The above examples used character-string constants for the directory and entryname arguments. However, you do not usually want to program in constant values; you want to get the pathnames from command arguments. In the preceding article in this series (January-February issue), we obtained such an argument and stored its pointer and length:

```
declare  pathp pointer initial (null ());
                                        /* pointer to pathname */
declare  pathl fixed binary;            /* length of pathname */
declare  path character (pathl) based (pathp);
                                        /* the pathname */
```

The pointer and length were obtained by calling "cu_$arg_ptr". To transform this pathname, which could be an entryname ("giraffe_stats"), a relative pathname ("<Noah>giraffe_stats"), or an absolute pathname (">udd>ARK>Noah>giraffe_stats"), into the absolute directory pathname and entryname required by "hcs_$initiate_count" and "hcs_$make_seg", call "expand_pathname_":

```
declare  expand_pathname_ entry (character (*), character (*), character (*),
         fixed binary (35));
declare  dirpath character (168);       /* absolute pathname of directory */
declare  entryname character (32);

  call expand_pathname_ (path, dirpath, entryname, code);
                                        /* get directory and entry */

  if code ^= 0                          /* couldn't interpret "path" */
  then do;                              /* handle error */
     call com_err_ (code, "do_good_stuff", "^a", path);
                                        /* include value of "path" in message */
     return;                            /* abort command */
  end;
```

This sets the value of "dirpath" to the directory portion of the absolute pathname of "path" and sets the value of "entryname" to the entry portion. The lengths of "dirpath" (168) and "entryname" (32) are the respective maximum lengths for an absolute pathname and an entryname.

Now you can use "hcs_$initiate_count", for example, to get a pointer to "giraffe_stats":

```
declare  rtrim builtin;                 /* trims trailing blanks */

  call hcs_$initiate_count (dirpath, entryname, "", bit_count, 1, seg_ptr, code);
                                        /* get pointer to segment */

  if seg_ptr = null ()                  /* couldn't get it */
  then do;                              /* handle the error */
     call com_err_ (code, "do_good_stuff", "^a>^a", rtrim (dirpath),
        rtrim (entryname));             /* include full pathname in message */
     return;                            /* abort command */
  end;
```

The builtin function "rtrim" removes trailing blanks from the values of "dirpath" and "entryname" so that, if an error occurs, the message comes out looking like, for example,

```
do_good_stuff: Entry not found. >udd>ARK>Noah>giraffe_data
```

without a lot of extra blanks after ">udd>ARK>Noah" and "giraffe.data". You should always include the full absolute pathname in error messages as soon as it becomes available from "expand_pathname_". Doing so makes it much easier for the person using your command to figure out what's wrong when he or she mistypes a pathname or thinks the working directory is sommthing other than what it is.

CLEANING UP

When your command is finished working with segments, it is important that it "clean them up." This involves four things:

- truncating modified permanent segments,
- resetting these segments' bit counts,
- terminating all permanent segments, and
- releasing all temporary segments.

If you have reduced the number of words of data in the segment, truncate it to its new length. This means setting to zero all the words beyond those containing actual data. First, calculate the bit count (which you will use later) from the new data size. (This operation is essentially the reverse of getting the size of the data structure from the bit count.) Next, calculate the new length of the segment in words from the bit count (if you don't have it already) and call "hcs_$truncate_seg" to do the truncating:

```
declare  nwords fixed binary (18);      /* length of data structure in words */

  bit_count = 9 * nchars;               /* calculate bits from characters */
  nwords = divide (bit_count + 35, 36, 18, 0);
                                        /* get integral number of words */
  call hcs_$truncate_seg (seg_ptr, nwords, code);
                                        /* zero out unused words */

  if code ^= 0                          /* couldn't truncate */
  then...                               /* report error */
```

You truncate the segment because this frees the unneeded storage so that it can be reused and so that the person using your command doesn't have to pay for it.

Resetting the bit count on a permanent segment is important, because the bit count will be used later by other commands to determine that segment's length. A call to "hcs_$set_bc_seg" sets the count. For example, using the value of "nwords" calculated above, we set the bit count as follows:

```
declare  hcs_$set_bc_seg entry (pointer, fixed binary (24), fixed binary (35));

  call hcs_$set_bc_seg (seg_ptr, bit_count, code);
                                        /* set the bit count */
  if code ^= 0                          /* couldn't set it */
  then do;                              /* handle error */
     .
     .
     .
```

Finally, terminate the permanent segments and release the temporary segments you have used. You terminate segments so that their numbers may be re-assigned and, more important, because the efficiency of your process decreases if the number of segments known to it (called the working set) becomes too large. To terminate segments initiated by "hcs_$initiate_count" with null reference names, call "hcs_$terminate_noname:"<13>

```
declare  hcs_$terminate_noname entry (pointer, fixed binary (35));

  call hcs_$terminate_noname (seg_ptr, code);
                                        /* terminate segment */
```

To release the temporary segments, call "release_temp_segments_":

```
declare  release_temp_segments_ entry (character (*), (*) pointer,
         fixed binary (35));

  call release_temp_segments_ ("do_good_stuff", temp_seg_ptrs, code);
```

<13> If a command initiates a segment before an earlier command has terminated it, a call to "hcs_$terminate_noname" by the second command does not terminate the segment; it remains accessible to the first command.

```
                                        /* return temporary segments to pool */
```

The arguments to "release_temp_segments_" are exactly the same as those passed to "get_temp_segments_" to obtain the pointers.

The terminating and releasing of segments is so important that you should take steps to see that they are performed even when your command is interrupted. As soon as you initiate your first segment or get the pointer to your first temporary segment, set up a handler for the "cleanup" condition.

The "cleanup" condition is signalled whenever a PL/I procedure performs a nonlocal "goto"--i.e., when it transfers to some statement outside its scope. For example, if Procedure A calls Procedure B, and Procedure B (instead of returning) transfers to a label in Procedure A, the "cleanup" condition is signalled. A common example of this is when you interrupt a command and type "release". The "release" command performs a nonlocal transfer to a label in the system subroutine that reads command lines from the terminal ("listen_"). However, when a command is aborted because of a nonlocal transfer, it doesn't have a chance to do the cleaning up it would normally have done before the "return" or "end" statement. The "cleanup" handler gives it a second chance to do this. When a procedure finishes because of a nonlocal transfer, its "kleanup" handler, if any, is invoked. There can be only one active "cleanup" handler in a PL/I procedure or "begin" block, so it has to handle everything:

```
       call hcs_$initiate_count (dirpath, entryname, "", bit_count, 1, seg_ptr,
          code);                        /* get pointer to segment */

       if seg_ptr = null ()             /* couldn't get it */
       then do;                         /* handle error */
          .
          .
          .
       end;

       on cleanup call cleanup_handler ();
                                        /* establish cleanup handler */
          .
          .
          .
     cleanup_handler:
       procedure ();

     /* This gets called in case of a nonlocal transfer */

          if seg_ptr ^= null ()         /* segment was initiated */
          then call hcs_$terminate_noname (seg_ptr, code);
                                        /* so terminate it */

          call release_temp_segments_ ("do_good_stuff", temp_seg_ptrs, code);
                                        /* back to the pool */
          return;
       end cleanup_handler;
          .
          .
          .
     end do_good_stuff;
```

Notice the test "if seg_ptr^=null ()" in the internal procedure "cleanup_handler". This is why we initialized "seg_ptr" to "null ()". Likewise we initialized "temp_seg_ptrs" to null pointers because "release_temp_segments_" does not mind being called with null pointers but does complain about invalid ones. We don't examine error codes here because there is not much we could do if they were nonzero.

* *

This is essentially what you need to know to manipulate segments with your commands. There are variations on the procedures we have discussed, and additional operations--such as manipulating a segment's access control list (ACL)--which are beyond the scope of this article. A perusal of the entry points to "hcs_" in the MPM Subroutines (AG93) will givm you an idea of the range of possibilities.

THE NEW MULTICS MAIL SYSTEM

by W. Olin Sibert
reprinted from the May-June 1979 Bulletin

The three commands that make up the new Multics Mail System ("print_mail", "send_mail", and "read_mail") were installed in March. They provide a far more powerful and flexible interface for sending and receiving Multics mail than did the old "mail" command, and also contain several improvements over the older (experimental) versions of "read_mail" and "send_mail", with which some users may already be familiar.

THE "print_mail" COMMAND

The simplest of these new commands is "print_mail" (prm). It is designed to replace the mail-reading capability of the old "mail" command, but differs in one important respect: rather than waiting until you have read all the messages in your mailbox and then asking if you want to delete all of them, it prints your messages one by one and asks, after each, whether you want to delete just that one. This way, you are spared the "all-or-nothing" decision, and can keep individual messages around as long as you want. Here is a sample session using "print_mail". User typing is preceded by an arrow (=>). In the example, the Multics ID of the user checking his mail is PJamison.DATA.

```
=> print_mail
You have 3 messages.

 #1 (5 lines)  03/22/79 23:32  Mailed by: AMandel.DATA
Date:  22 March 1979 11:35 est
From:  AMandel.DATA at MIT-Multics (Albert Mandel)
Subject:  set_database command
To:  Michaels.DATA at MIT-Multics, PJamison.DATA at MIT-Multics

I have a new version of the set_database command--try it.
It's in >udd>DATA>AMandel>new, and has most of the new options
we talked about at the last meeting.  It also implements the new
multiplex locking protocol.
 -- Al

print_mail: Delete #1?  => yes

 #2 (2 lines)  03/22/79 23:33  Mailed by: Michaels.DATA
Date:  22 March 1979 11:47 est
From:  Michaels.DATA at MIT-Multics
Subject:  Re:  set_database command
To:  AMandel.DATA at MIT-Multics
cc:  PJamison.DATA at MIT-Multics

I tried it, but all it did was take faults.  Looks
like we shouldn't switch just yet....  --Carl

print_mail: Delete #2?  => yes

 #3 (3 lines)  03/22/79 23:34  Mailed by: AMandel.DATA
Date:  22 March 1979 16:02 est
From:  AMandel.DATA at MIT-Multics (Albert Mandel)
Subject:  Re:  set_database command
To:  Michaels.DATA at MIT-Multics, PJamison.DATA at MIT-Multics

You may have tried it while I was recompiling it.
Try it again & see if it works this time.
 -- Al

print_mail: Delete #3?  => yes
r 1036 1.023 4.496 117
```

The session above shows the common types of message headers produced both in Multics mail by the new "send_mail" command (see below) and by the mail-sending programs on other ARPANET host computers. The field names ("Date", "From", "Subject", "To", and "cc") are largely self-explanatory.

The "print_mail" command has several control arguments; of particular interest to many users is the "-no_interactive_messages" (-nim) option, which causes "print_mail" to ignore any interactive messages being held in the mailbox, and the "-no_header" (-nhe) option, which suppresses most of the message headers, printing only the "From:" and "Subject:"

fields. Further information about these headers can be found in a description of "print_mail" in a forthcoming addendum to the MPM Commands and Active Functions (AG92).

Because "print_mail" is designed to be very simple to use, it has none of the more advanced features provided by "read_mail" (logging, forwarding, etc.) When you need these facilities, you will want to use "read_mail" rather than "print_mail".

THE "send_mail" COMMAND

Use the new "send_mail" (sdm) command to send mail. PJamison.DATA might use "send_mail" like this, after printing the mail seen above:

```
=> send_mail AMandel.DATA -cc Michaels.DATA -log
   Subject:  => new set_database command
   Message:
=> I tried it, and it seems to work fine for me.
=> Did you fix the bug that caused it to randomly
=> delete directories?
=> Also, I think it should have a "-brief" control argument to
=> suppress the messages.
=>
=> Pete
=> .
   Mail delivered to your logbox.
   Mail delivered to AMandel.DATA.
   Mail delivered to Michaels.DATA
   r 1043 0.571 3.585 56
```

Here, "send_mail" was used to send a message to two people, and to send a copy to the user's "logbox" (a segment in his home directory named "PersonID.sv.mbx", in which the message is retained for posterity). The header for that mail would look like this:

```
Date:  23 March 1979 10:42 est
From:  PJamison.DATA at MIT-Multics
Subject:  new set_database command
To:  AMandel.DATA at MIT-Multics
cc:  Michaels.DATA at MIT-Multics, PJamison.DATA at MIT-Multics
```

PJamison.DATA appears in the "cc:" field because he specified the "-log" control argument in the "send_mail" command line. There are many more options and frills available for "send_mail"; they are described fully in the new MPM addendum.

One other feature deserves mention here: you can define an abbrev that inserts your full name in the "From:" field as a comment (as is shown in messages 1 and 3 above):*

```
.ab  Sdm  send_mail -from AMandel.DATA -comment "Albert Mandel"
```

This kind of abbrev could also be used to set various other options. Since all options are available in both the positive and negative forms (e.g., "-fill" and "-no_fill"), any option specified in an abbrev can be overridden by specifying its opposite form later on the command line. In particular, the "-fill" control argument is recommended; it automatically "fills" the input text to a line length of 72 characters before sending it, greatly improving the message's readability.

The "send_mail" command also allows you to edit your message before sending it. To do this, type the characters "\f" (rather than ".") as the last line of input. This puts you into the "qedx" editor, with the main buffer containing the text of the message (which you can then edit). When you give the editor's "q" (quit) request, you enter "send_mail" command level, and your message contains the newly edited text. If all you want to do at that point is send the message, type the two requests "send" and "quit". However, a large variety of other "send_mail" requests (all described in the new MPM) are available for doing other things (adding or removing recipients, etc.).

---

*For more information on the Multics "abbrev" facility, see the MPM description of the "abbrev" command, and the article "MULTICS TIME-SAVERS: A BEGINNER'S GUIDE TO (PERSONAL) ENERGY CONSERVATION" in the May 1978 issue of the Bulletin.

ABOUT THE "read_mail" COMMAND

The "read_mail" (rdm) command is quite powerful. It lets you peruse the contents of a mailbox; selectively delete, print, or list messages; save copies of messages in other mailboxes; "forward" mail to other users; and invoke the "send_mail" subsystem to compose replies. For complete information on using the "read_mail" command, type "help read_mail".

MULTICS TAPES FROM SCRATCH

Adapted with permission from Two Bits Worth, the newsletter of the University of Southwestern Louisiana Computing Center. Alicia Towster wrote the original article on the basis of interviews with Sam Bullard, Bob Sonnier, and James Dugal. Appeared in the September-October 1979 Bulletin.

Magnetic tapes are simply very long thin strips of mylar backing coated with a thin layer of some material which can retain magnetic information. They are an economical and relatively durable way to store information; for less than $20 you can acquire a tape capable of storing (on Multics) up to about 45 million bytes of useful information; and, if you are able to keep this tape away from heat, humidity, dirt, magnetic fields, and malfunctioning tape drives, it can give you reliable service for several years. (Even under the best circumstances, tapes will eventually begin to show signs of wear. Thus, cautious tape users will often keep more than one copy of their important data.) Magnetic tapes have a standard width of one-half inch; several lengths are available: 400', 1200', 2400', and 3200'. At either end of a tape is a shiny aluminum patch to mark the Beginning of Tape (BOT) and End of Tape (EOT).

It sounds simple enough.

Why, then, when you approach a computer installation carrying a "foreign tape" (that is, one which was not created on that particular computer), does the staff eye you with misgivings, rather as though you were carrying a foreign virus? They are hoping that you can describe it clearly enough that they will quickly know what treatment to prescribe.

You see, it is not nearly as simple as tape cassettes--tape them on one machine, play them back on another. There is a considerable variety of ways that information can be written on a magnetic tape. And it is entirely possible to produce a tape which is totally incompatible with the machine on which you desire to use it. These incompatibilities may be due to either hardware (the kinds of tape drives which are available) or software (programs which read and write tapes, normally supplied by the manufacturer).

First, there are differences in the ways that tape drives can physically arrange and/or access information on a tape. Some drives will read or write nine bits (binary digits, either a zero or one) in a row across the width of the tape; these are called "nine-track" drives, and the tapes they produce are called, reasonably enough, nine-track tapes. There are also seven-track drives which deal with seven-track tapes on which rows of seven bits are stored.

Information can also be arranged differently down the length of the tape: this is called tape "density" and is measured in "bpi" which stands either for bits per inch (if you think in terms of only one track at a time) or bytes per inch (if you think of the entire row of tracks as a byte). Possible densities are 200 bpi, 556 bpi, 800 bpi, 1600 bpi, and 6250 bpi. A particular drive is limited in the number of different densities that it can handle.

Higher densities are not simply more compressed than lower densities--they also contain additional information that enables the drives to continually be checking on whether or not they are reading your data successfully. For example, on a nine-track 800 bpi tape, one bit in each row is used as a "parity bit"--that is, it is not actually part of your data; rather, its value is a function of the value of the other bits in the same row.

Most typically, "odd parity" is used; that is, the ninth bit is set so that the sum of the bits in the row will be an odd number. On 1600 bpi tape, parity information is also computed for groups of rows and written at regular intervals along the tape. In addition, unique patterns of bits are written at intervals for the purpose of synchronization. Normally you can remain totally unaware of these extra bits; the tape drive hardware/firmware generates them and/or checks them automatically. If they fail to check out, the proper action is left to the program which is controlling the attempted read. Often such a program will elect to retry the read; if this does not work, you will, of course, get an error message. Such a message could indicate a bad tape, a malfunctioning drive, or a mismatch between the way your tape actually is (tracks, density, or parity type) and what the drive expects it to be.

Information is not written continuously along the length of the tape; it is written in "blocks" or "physical records". These physical records may well differ in size from the logical divisions of the data ("logical records") which you have placed on the tape. If logical records are short, several may be grouped together into one physical record; long logical records may also span several physical records. Most typically, physical records will be of a uniform size ("fixed block"), but it is also possible for their sizes to vary ("variable block"). At the end of a physical record is a unique bit pattern (the End of Record Mark), as well as any bit patterns used for synchronization. The space between records is called the "interrecord gap".

The tape drive itself deals only in terms of these physical records, moving them from the tape to a buffer in the computer's memory (or vice versa, in the case of writing a tape). As information is transferred between the tape drives and the buffer, the programs which control the tape usage deal with the current buffer and get ready to handle the next buffer. Because one or more of these buffers must reside in the computer's real memory, many installations will have some upper limit on the size of physical tape records that they can handle.

But this is only a small part of what makes tapes difficult. Consider--how are those meaningful bits of data to be interpreted? Remember that a Multics word consists of 36 bits and your tape consists of numerous rows of 8 bits and 8 does not go evenly into 36....And if your tape was perhaps created on some other computer with a different word size?

Well, luckily there are standards. But unfortunately there are more of these than we might like. First there is the theoretical standard: the American National Standards Institute's specification of how to map those rows of bits on tape back and forth between words on the computer; this accounts for the Multics i/o module tape_ansi_. Then there is the de facto standard, which has sheer numbers on its side; that accounts for tape_ibm_. But there is nothing to prevent any of the computer manufacturers from developing their own internal standards, tailored to their own hardware, and so they do. This can potentially produce more efficient or appropriate use of tapes so long as you are committed to a particular brand of computer, but will almost certainly cause problems if you try to switch brands and take your data with you. To deal with these standards fiefdoms, Multics has two more tape-handling i/o modules: tape_mult_ (for Multics Standard Tapes) and tape_nstd_ ("tape nonstandard" for everybody else).

"Gee," you may be saying, "I use tapes a lot and I never even heard of those i/o modules; I can't be using any of them." Yes, you are; they just get invoked for you behind the command-level scene. If you have switched to tape-archive, you will normally be using tape_ansi_.

To complicate matters still further, these various standard tapes can have subtypes. Tapes may be either labeled or unlabeled. Labels are, in theory, extremely helpful, since the information encoded in a tape label will, among other things, tell you what kind of a tape you are actually dealing with. But there's a catch: label formats can vary, too, and some computer installations may not have programs available to interpret tape labels. Thus, they could actually make your tape harder to read. (For further information, see "NOTES ON USING MULTICS TAPES", elsewhere this issue.)

Another complication arises from the different ways that information can be represented inside various computers. The internal binary representation of machine instructions or data will normally be specific to a particular computer; thus, it is useful to tape such binary information only if the tape will be reread on exactly the same sort of computer. "Portable tapes" (that is, tapes which are to be carried to a different sort of computer) should use one of the standard sets of character codes to represent the information; there are two widely used standards: EBCDIC, which is promulgated by IBM, and ASCII, the American Standard Code for Information Interchange. In the absence of other information, you may expect most IBM format tapes to use EBCDIC and most ANSI tapes to use ASCII.

With so many variables involved, clearly it is only sensible to write down the appropriate information about the tape when it is created and to keep this information with the tape. However, this does not ensure that it can be read by the computer of your choice, which simply may lack the hardware or software that you need.

This is a lot of information to keep straight, so here are some checklists.

Ways in which tapes (and installations) can vary:

- -number of tracks
- -density
- -type of parity
- -size of records
- -size of block
- -type of format
- -labeled or unlabeled
- -type of encoding

Portable tapes which Multics can handle:

- -nine-track
- -800 or 1600 bpi
- -odd parity
- -any record size
- -any block size between 20 and 8192 characters (for output, block size must be evenly divisible by 4)
- -a variety of formats, but IBM and ANSI are safest

-most types of labels
-ASCII or EBCDIC encoding

A tape of this sort could be read at many installations:

-nine-track
-800 bpi
-odd parity
-a fixed record size
-a fixed block size no larger than 2048 characters
-IBM format
-unlabeled
-EBCDIC encoding

If you need an unlabeled tape, you must use the "copy_file" command which will copy a structured file from one place to another. Suppose you have an ASCII segment called "random_thoughts" consisting of lines of various lengths, none of them longer than 136 characters. Now, this is not a structured file, but you can temporarily create a structured version of it by using the records_stream_i/o module. Thus, you could create a file in a generally portable format on tape reel number 1234 by the command line:

```
copy_file -input_description "record_stream_ -target vfile_ random_thoughts" \
-output_description "tape_ibm_ 1234 -create -no_labels -format fb -record \
136 -block 1360 -density 800 -mode ebcdic -number 1"
```

Yes, that is one command line; you can shorten it a little by taking advantage of the defaults and short forms of the various options. This would produce:

```
cpf -ids "record_stream_ -target vfile_ random_thoughts" -ods "tape_ibm_ \
1234 -cr -nlb -fmt fb -rec 136 -bk 1360 -den 800 -nb 1"
```

But this is still a lot to type, so you may well want to make sure there is a tape drive available before you type it. For an explanation of all the options which tape_ibm_ can use, see the MPM Peripheral Input/Output Manual.

The "copy_file" command can also be used to read a foreign tape onto Multics. Suppose a friend has sent you a labeled ANSI tape; he tells you that it contains two files consisting of 80 character records in blocks of 800 characters written at 1600 bpi. He neglected to tell you what character set was used, but since it is an ANSI tape, it is probably ASCII. You register the tape as reel number 4321 and attempt to persuade copy_file to read the second file from the tape by typing:

```
cpf -ids "tape_ansi_ 4321 -nb 2 -fmt fb -rec 80 -bk 800 -den 1600" -ods \
"record_stream_ -target vfile_file2"
```

Looks good. But it doesn't work at all; you get read errors whatever drive you use. Since it's a labeled tape, you decide to try:

```
list_tape_contents 004321 -io_module tape_ansi_
```

And guess what! It turns out that it's really an 800 bpi tape.

Tapes may be a little difficult, but misinformation about them can make them much, much worse.

NOTES ON USING MULTICS TAPES

reprinted from the September-October 1979 Bulletin

If you've ever tried to use tapes on Multics, you've probably discovered that, while it's not quite as difficult as stealing the Golden Fleece, it's no picnic either. This can be a real problem, but Honeywell is working to try and solve it for you. In fact, several programs have recently been installed that improve the process significantly; they make it easy to do things that were once impossible.

"I have a tape with a FORTRAN program on it. It came in the mail yesterday, and I don't know anything about it. How can I read it onto Multics?"

Well, if you are fairly certain you know the tape's density, format, and label standard, the easiest thing to do is to use the "tape_in" command. This command is documented in the Multics Programmer's Manual Peripheral I/O (AX49) and on line (type "help tape_in -title"). However, rarely will you be fortunate enough to have this much information about a random tape. Frequently, you may even be uncertain about the kind of machine on which it was written--let alone how it's labeled or how big the records are. If this happens to you, cheer up! Relief is available, in the form of a new program named "read_tape_and_query" (rtq).* (You need to be somewhat knowledgeable about tape formats to use this program. An alternative possibility is to use the "Mystery Tape" service available from Operations at a cost of $10.00.)

To examine a tape from slot 064437, you must type:

```
rtq 064437
```

There is no MPM documentation yet available for this command; to learn how to use it, get a dprint of the file >dox>is>rtq.info or type "help rtq".

The "read_tape_and_query" command is extremely handy for examining tapes of unknown origin. Since it allows you to change the density at which the tape is being read, you can figure out the density at which the tape was recorded by reading at different densities until you find the one that doesn't get read errors. You can also have "read_tape_and_query" read a record into its internal buffer and dump it in a variety of formats, using 6-bit, 8-bit, or 9-bit characters. Finally, it has facilities for "guessing" what kind of label (if any) is on the tape.

"I have a lot of large files that I hardly ever use, but I don't want to delete them. How can I get them off my disk space?"

The best solution for this problem is to use the "tape_archive" command, documented in the latest edition of the MPM. This lets you maintain a tape in much the same manner as you would a Multics "archive" segment: you can add, delete, and replace files, without ever having to worry about tape format, density, and so on.

The "tape_archive" program creates and maintains an on-line control segment (with suffix ".ta"), in which it records the contents of each tape. It is not absolutely necessary for you to keep this segment; it can be re-created by giving "tape_archive" with the "load_table" key. It must, however, be available whenever you prepare to modify the contents of the tape, so it may be more convenient for you to keep it on line.

The basic method of use is this: you issue a series of "tape_archive" commands specifying the appropriate keys for adding, deleting, or replacing files; these operations are "remembered" in the control segment, and are all executed before the tape is mounted. When you have requested all the changes you want made to the tape at this time, give the "tape_archive" command with the "go" key. This mounts your tape and makes all the specified changes.

You can perform a number of other operations on a control segment: you can get a list of the tape's current contents, or a list of the requests you have issued so far, but which have not yet been fulfilled, and you can cancel requests not yet performed.

The "tape_archive" command does a lot of your worrying for you (e.g., about a tape's content and format). When it needs a new tape (for instance, the first time it is run with a new control segment), it asks you for the slot number.

*As of this writing, "read_tape_and_query" is scheduled to be installed by early October.

At the end of a tape, "tape_archive" may inquire whether you want to overwrite an unexpired file. If this happens to you, answer "no". When you reach Multics command level, use the tape_archive "t" key with the control argument "-pending" to find out what requests are pending. Use the tape_archive "cancel" key to cancel those requests. You can then use the tape_archive "t" key with the control argument "-long" to find out about wasted space (the "waste factor") on the tape. If this waste factor is high, you may want to get another tape and use the tape_archive "compact" key to copy your useful files onto the new tape. If the waste factor is low, and if you don't plan to modify the files on the original tape, you can use it as is. The tape_archive "alter" key could be used to make this archive more than one tape volume long, but this is likely to prove less efficient than maintaining a separate archive in each tape volume.

A number of other facilities are available from "tape_archive"; they are described in the MPM and in the on-line info file. (Type "help ta" for complete information.) At this writing, the "reconstruct" operation is not yet implemented; thus, you cannot recover automatically if an "archived" tape becomes mangled.

"I'm going away for the summer, and I don't want to pay storage charges for my project all that time."

You could use "tape_archive" here; however, the best way to solve this problem is with "backup_dump" and "backup_load", the same programs used by the Multics Hierarchy Backup System. Essentially, this allows you to put a "snapshot" of a hierarchy on a tape. When you retrieve it from the tape, it is recovered in exactly the form it had originally--all the ACLs are the same, all segments and directories have the same contents, and so on. Unfortunately, using these programs is a rather arcane process, and the Honeywell documentation (in the Multics Operators' Handbook, AM81) is designed for operators rather than for ordinary users. We therefore recommend that you contact IPS Consulting Services at 253-7020 if you are interested in using the programs.

"Help--I don't understand!"

You can always direct questions about Multics tapes to the on-line consultant. (Type "help olc" for details.) The consultant will often refer you to a manual or to another consultant, or ask that you make an appointment. Before sending a distress cry, please be sure to examine the appropriate documentation carefully.

FUN WITH READ_MAIL

by Dorothy Corbett

reprinted from the March-April 1980 Bulletin

Last year, when the new Multics mail system first came out, we ran an article describing two of its three commands.* We managed to cover "print_mail" and "send_mail", but we decided to leave "read_mail", with its swarm of features, until later. Well, it's later. What follows is a less-than-exhaustive look at some of the things read_mail can do besides printing mail, among them forwarding messages, sending replies, saving messages in an auxiliary mailbox, and writing them into segments.

To do any of these things, you begin by issuing the read_mail command. It responds by telling you how many messages your mailbox contains. (If you don't have a mailbox, read_mail automatically creates one.) Then it enters a read_mail request loop, and prompts you for further instructions by typing "read_mail:" It will remain in the loop until you exit by giving the "quit" request.

If your mailbox contains more than one message, you can tell Multics which message to deal with by following each instruction with a message specifier. The easiest specifier to use is the "message number", which is assigned by Multics and typed out when you "list" the mailbox contents. Other valid specifiers are keywords (such as "all", "current", "first", "last", "previous", and "next"), and ranges. For example, the request "pr 2:last" prints all messages but the first. If you don't give a message specifier, your command applies to the "current message", which is the first message when you enter read_mail and is reset as you delete a message or explicitly specify another in a request. The current message is marked by an asterisk when you list messages.

By default, read_mail permits you to manipulate only messages sent by send_mail. However, if you specify the -interactive_messages control argument (-im), read_mail will also process those sent by the send_message command.

USING read_mail TO SEND MESSAGES

You may need to reply to a message, or just to pass it along to another user. You can do this most easily while in the read_mail request loop.

To forward mail, give the "forward" request, a message specifier, and a destination. Multics will tell you whether the message was received.

A "reply" request ("reply" followed by the specifier of the message you're answering) puts you in the middle of the "send_mail" command. Read_mail automatically creates the necessary header, so it just prompts you for the message text. Whenever it sends messages, "read_mail" adds extra header fields which indicate origin, topic, and date of the reply or forwarded message. Together with the header from the original message, this information can be very lengthy. If you receive a reply or forwarded message, you can suppress most of the header fields by giving the no_header (-nhe) control argument with the print request.

USING read_mail TO SAVE MESSAGES

You'll probably use "delete" more often than any other "read_mail" request but "print". Not only is it incomvenient and expensive to have old messages cluttering your mailbox, but, more important, it can only hold a limited amount. Once it is full, you can no longer receive mail. You can alter the size of your mailbox, using the mbx_set_max_length command, but you can only do this when your mailbox is empty. Use the "delete" request to remove unwanted messages. (If you accidentally delete a message you still need, the "retrieve" request can bring it back. You must "retrieve" the message before you quit read_mail, though.) Sometimes you may want both to retain a copy of a message and to delete it from your mailbox. Read_mail provides you with many ways to do this. Using the "write" request you can copy the message into a segment, to whose entryname read_mail will assign a final component of "mail", which you can then edit, execute, or dprint. Or you store the message in a save mailbox. Use the "log" request to transfer messages to your logbox, which is the default save box. (Use the "save" request to specify other save boxes.) For example, to log your second message, type "log 2".

If you don't have a logbox, "read_mail" will automatically create one, in your home directory. It will be called person_id.sv.mbx. (To examine your logbox, use the -log control argument of read_mail.)

---

*See "The New Multics Mail System" in the May-June 1979 Bulletin.

ISSUING MULTICS COMMAND WITHIN read_mail

Lines typed to read_mail that are preceded by two periods ("..") are passed to the Multics standard command processor. Thus, you can issue Multics commands without leaving the read_mail request loop. For example, the following exchange shows how to write your second message into a segment and dprint it immediately (what you type is preceded by =>):

```
read_mail: => write 2 my_mail
read_mail: => ..dprint my_mail.mail
```

Since you haven't left the read_mail loop you can continue to process your messages.

Properly used, read_mail can make life on Multics much easier. We've described only some of read_mail's capabilities; if you want to learn more about this command, type "help read_mail", or see the Multics Programmer's Manual: Commands and Active Functions (AG92).

SNOW WHITE CHECKS HER MAIL (all user typing is preceded by ==>)

```
==>  read_mail
     You have 3 messages.

     read_mail: ==>  list

     Msg   Lines   Date    Time   From                        Subject
       1*    (4) 02/21/80 13:22  Sneezy.FOREST at MIT-Mult dinner
       2     (3) 02/21/80 13:56  Dopey.FOREST at MIT-Mult  not sure
       3     (3) 02/21/80 14:15  Grumpy.FOREST at MIT-Mult complaints

     read_mail: ==>  print all

       1 (4 lines)  02/21/80 13:22  Mailed by: Sneezy.FOREST
     Date:  21 February 1980 13:22 est
     From:  Sneezy.FOREST at MIT-Multics
     Subject:  dinner
     To:  SWhite.FOREST at MIT-Multics

     Please don't hold dinner for me tonight. I'm coming down with
     a cold and don't have much of an appetite. Maybe I'll just have
     a little chicken soup later on.
                                        Sneezy
      ---(1)---

       2 (3 lines)  02/21/80 13:56 Mailed by: Dopey.FOREST
     Date:  21 February 1980 13:56 est
     From:  Dopey.FOREST at MIT-Multics
     Subject:  not sure
     To: SWhite.FOREST at MIT-Multics

     Oh I had sumthin real important to tell you but now I forgot
     an now I can't remember how to get out of this send_mail
     command. lessee here QUIT STOP how about a period ummm
      ---(2)---

       3 ( 3 lines)  02/21/80 14:15  Mailed by: Grumpy.FOREST
     Date:  21 February 198014:15 est
     From:  Grumpy.FOREST at MIT-Multics
     Subject: complaints
     To:  SWhite.FOREST at MIT-Multics

     I'm really fed up with conditions in the commune. Sleepy never
     does his share. Sneezy keeps me awake all night. And all this
     whistling is driving me nuts. --Grumpy
      ---(3)---

     read_mail: ==>  forward 1 Doc.FOREST
     Mail delivered to Doc.FOREST.

     read_mail: ==>  reply 2
     read_mail (reply): Replying to 1 total recipient.
     Message:
==>  I guess you figured out that a period on a new line will
==>  get you out of send_mail.  But you could have typed the
==>  character sequence "\fq".  This would have put you into
==>  the send_mail request loop, from which you could have quit
==>  without sending me junk mail. --Snow White
==>  .
     Mail delivered to Dopey.FOREST.

     read_mail: ==>  log 3

     read_mail: ==>  delete all
     All messages have been deleted.

     read_mail: ==>  quit
```

BAFFLED BY BUFFERS

by Michael Thornton

reprinted from the May-June 1980 Bulletin

[The scene: Emanuel Amanuensis, his eyelids finally dropping shut, is lapsing into sleep when he hears a knock at the door. He hobbles across his monastic quarters, dimly lit by the blue glow of his CRT--left on once again. Emanuel opens the door to discover his boss, Dante Alighieri.]

DANTE [stepping in with another man]:
Sorry to bother you so late, but we've got a problem. This is Cardinal Awesome. He's just come from the Council with a ruling on the Inferno. They say we can't have the simoniacs in a lower circle than that of the grafters and thieves. Looks like we'll have to switch Cantos 18 to 22 with Cantos 23 to 25.

EMANUEL:
But that means I have to retype 441 lines!

DANTE:
There's an easier way. I guess it's time you learned about buffers.

THE CARDINAL:
You mean your man doesn't know about buffers? I've been using buffers for over five years.

EMANUEL:
Well, I've been using qedx ever since Saint Edm was pronounced insane, and have fared perfectly well without them.

DANTE [who, while the others bickered, has logged into Multics]:
Think of a buffer simply as another, separate editing area just like the one you find yourself in when you enter qedx as usual. In fact, your usual working area is a buffer too; its name is "0" (zero). You get buffer "0" by default, but qedx automatically creates others for you as you move stuff into them, or go to them to create their contents.

Okay, Emanuel, let's try removing those four cantos from their current place in the poem and putting them into a buffer. Invoke qedx, read in the poem, and find the line numbers of the beginning and end of the section we want to move:

```
qedx
r inferno.runoff
/Cosi di ponte in ponte/
Cosi di ponte in ponte, altro parlando
=
2901
/quel che tu, Gaville/
l'altr' era quel che tu, Gaville, piagni
=
3342
```

Now give the qedx request "m" (for move) preceded by the address that delimits the block you want to move and followed by the name of the buffer into which you want the block to go. So, to move the section to buffer "2", type:

```
2901,3342m2
```

One-character buffer names are easiest to use. You can use longer names, but you must enclose them in parentheses. (If you ever work with many buffers in one session, you'll appreciate the mnemonic value of long names.) You could have moved the section to a buffer named "deceit" by typing:

```
2901,3342m(deceit)
```

We can verify that we in fact did move the section to buffer "2" by going there, that is, by making buffer "2" (rather than buffer "0") our current working area. Give the "b" request, followed by the name of the buffer to which you want to go:

```
b2
```

Now we are in buffer "2". We can edit, print, or save its contents, just as we could the poem when we were in buffer "0". Let's just print a few lines:

```
226,228p
Ed ei rispose: Fu fate la loona
```

```
        quel di Albina, vasel d'ogne froda,
        ch'ebbe i nemici di suo donno in manno,
```

EMANUEL:
That's from the grafters Canto, awright. But how do we append this stuff behind the simoniacs and soothsayers Cantos?

DANTE:
First, go back to buffer "0", which still contains the bulk of the poem, and locate the line after which we want to append the stuff we moved to buffer "2".

```
        b0
        /e andavamo introcque/
        Si mi parlava, e andavamo introcque.
```

Now type:

```
        a\b2\f
```

You're familiar with "a" and "\f"; one is the qedx append request, and the other is the return-to-edit-mode signal. The "\b2" essentially means "put the contents of buffer '2' here". The result is the same as if you had actually typed every character (including newline characters, or "carriage returns") that is currently in buffer '2'.

THE CARDINAL:
I think it would be useful if we utilized a summarization strategy at this point in time. Subsequent to reading the segment "inferno.runoff" into qedx's buffer "0", Dante moved a portion of it to buffer "2", and then appended that portion at new location in the segment contained in buffer "0".

EMANUEL:
Wonderful. Now we can all get back to bed.

THE CARDINAL (to Dante):
It might enhance performance facility if you taught him how to list currently-established existent buffers.

DANTE:
Uh, yeah. You can list buffers you have around by typing "x". Behold:

```
        x
          4785 -> (0) >udd>Dante>Dante>inferno.runoff
           442    (2)
```

Here you can see we've used two buffers; their names are listed in parentheses. The number of lines in the buffer is listed to the left; to the right is listed the pathname, if any, associated with the buffer.

THE CARDINAL:
When I type "x" I see at least six or seven buffers listed because I have qedx macros set up automatically.

EMANUEL [with trepidation]:
What are macros?

DANTE:
Remember when we used "\b" to call forth the contents of a buffer (as if it were typed literally) when in input mode? You can also, while in edit mode, invoke a buffer containing editing requests. By doing so, you can easily execute the series of editing requests contained in the buffer.

Suppose I gave you the tedious task of marking the first and last lines of, say, thirty tercets with asterisks. You could go to buffer "9" and create a two-line macro:

```
        b9
        a
        s/^/***  /
        +2s/$/  ***/
        \f
```

This makes the contents of buffer "9" these two lines:

```
        s/^/***  /
        +2s/$/  ***/
```

(If you don't believe me, type "1,$p".) Now you can go back to buffer "0", and, once you've located a selected tercet, execute the macro by typing:

```
\b9
```

This has the same effect as issuing the substitute requests directly, but (if you have to do it thirty times) requires less typing. Let's try it on the tercet that begins "Venir se", and then print the three lines to see how they've changed:

```
/Venir se/
Venir se ne dee giu tra miei meschini
\b9
-2,.p
***  Venir se ne dee giu tra miei meschini
perche diede 'l vsi e cms systemata,
dal quale in qua stato li sono a' crini;  ***
```

Obviously, the macro used here has limited application, but I have more complicated ones that do such things as underscoring and capitalization.

THE CARDINAL:
So do I, and, as I said before, I arrange for them to be set up automatically each time I invoke qedx.

EMANUEL:
And how, pray tell, does one do that?

DANTE:
Here things get complicated. You have to understand that qedx behaves a little differently when you invoke it with arguments. If you type:

```
qedx read sloth lust
```

qedx will first go to a buffer named "exec" and read in a segment named "read.qedx". Then qedx goes to a buffer named "args" and puts the remaining arguments (two in this instance, "sloth" and "lust") on separate lines in this buffer. Finally, qedx goes to buffer "0" and executes buffer "exec" as a macro.

Now suppose you've created a segment "read.qedx" which contains this line:

```
r \b(args)
```

EMANUEL:
What a minute! How can I enter a backslash as a line of text?

DANTE:
Well, you can type "\c" in front of the backslash. For example:

```
a
r \c\b(args)
\f
```

Anyway, you have a segment containing that line. If you invoke qedx thus:

```
qedx read gluttony.runoff
```

the following occurs: the line "r \b(args)" is put into buffer "exec", and the line "gluttony.runoff" is put into buffer "args". Then qedx goes to buffer "0" and executes buffer "exec", which is equivalent to typing:

```
r gluttony.runoff
```

You can use this feature to initialize macros and to abbreviate the typing of the names of segments you want to edit. Suppose your "read.qedx" segment contains these line:

```
bu
r underliner_macro
b0
r \b(args)
```

(We'll assume you have a macro stored in the segment "underliner_macro".) Suppose further that you have an abbrev "qr" that stands for:

```
do "qedx read &1.runoff &rf2"
```

Then you can establish your underlining macro in buffer "u" and read in the segment "gluttony.runoff" into buffer "0" by typing:

```
qr gluttony
```

THE CARDINAL:
This fellow has had a great deal to absorb tonight; we don't want to overload his mind. After all, he's only a typist. Besides, I must be off: I'm to interface with His Holiness on matters that will greatly impact end-communicants. But keep with it, Samuel; I'm sure that with perseverance someday you'll become a great buffoon.

EMANUEL [after waiting for the chortling Cardinal to leave]:
Did you hear that? I swear, he's gonna need Bufferin and a lot more when I----

DANTE:
You may as well calm yourself. He has the Pope's ear. There's no toleration for those, however rational, who oppose him. His rise to favor was rapid, even for these unsettled times. I suggest you use buffers while you can, Emanuel. Over in Tuscany, Emacs the Unquenchable is amassing his forces, and many rejoice his advent. Before long, the world may be very different from the one we know today.

MINE!

by Steven H. Schwartz

reprinted from the September-October 1980 Bulletin

Multics access controls are a popular topic of conversation between users and consultants. Although at first glance access controls may seem to be a hodge-podge of random letters, they are really quite easy to use.

Basically, segments on Multics can only be used by those who have access (permission). Multics grants some access by default and allows you to grant more. For instance, when you create a segment, you are automatically given "rw" access to it. This means you can "read" it--look at and copy it--and "write" it--change its contents. Or, if the segment is a compiled program, you are given "re" ("read" and "execute") access.

Directories are also subject to access controls. When you create one, you are given all three kinds of access: status (s), which lets you list the names and characteristics of its entries (i.e., segments and subdirectories); modify (m), which lets you change the characteristics of (and delete) existing entries; and append (a), which lets you add entries.

Multics knows who has access to what, because each segment and directory has an access control list (ACL)--a list of users who have been given access, and what access they have been given.

Each name (access identifier) on an ACL has three parts: a person_id, a project_id, and an "instance tag"--a letter which represents a process type. For example, an instance tag of "a" stands for an interactive user; "m" for an absentee user; and "z" for daemon and other systems processes. An asterisk can replace any part of an access identifier to indicate that that part matches any user. Instance tags, for example, are usually set to "*". Multics determines a user's access by trying to match his access identifier with one on the access control list. For instance, an interactive user called Pinocchio.NOSEY would match any of the following access identifiers on an ACL:

```
Pinocchio.NOSEY.a
*.NOSEY.*
Pinocchio.*.a
Pinocchio.NOSEY.*
*.*.*
```

Note that *.*.* matches any user.

Frequently a person's user_id matches more than one access identifier in the ACL. Multics arranges the access identifiers on the ACL in order of specificity, with person_id, project_id, and instance tag considered in that order. Multics uses the first occurrence of a match to determine access. If a person's user_id does not match any of the access identifiers on the ACL, that person is not given access to the segment or directory. Here is how Multics would arrange the access identifiers given above:

```
Pinocchio.NOSEY.a
Pinocchio.NOSEY.*
Pinocchio.*.a
*.NOSEY.*
*.*.*
```

If you have "sm" access to a directory, then you can modify the access control lists for that directory's segments and subdirectories. As mentioned previously, when you create a segment or directory, you are automatically given access to it. Users on the project SysDaemon are also given access to all segments and directories. They need this access in order to back up your files, print them, punch them onto cards, and so forth--this should not present a security problem. If necessary, you can delete the access for *.SysDaemon.*, but then these services will not be provided for you.

Three commands are used to manipulate ACLs. "list_acl" (la) lists the ACL for any segment or directory whose pathname is specified. If you omit the pathname, Multics prints the

ACL for your working directory.

The "set_acl" ("sa") command both adds an access identifier to an ACL and changes the access for an access identifier already on the ACL. For instance, to give Hatfield.MCCOY "read" access to a segment called "cannons", you enter:

```
set_acl cannons r Hatfield.MCCOY
```

You don't need to include the instance tag; Multics assumes it to be "*". And, you can include more than one access-access identifier pair in the same command. If you want to change your own access you can omit the access identifier.

You can assign users any of the appropriate types of access mentioned above ("r" "e" "w" and "s" "m" "a" for segments and directories respectively) in any combination, except that "m" access to a directory must be given with "s" access. For instance, you can assign a user "r" access to permit him to read, but not alter, a segment. You can also assign a user "null" (n) access, a valid access type on both directories and segments. This prevents the user from having any access at all, and is particularly useful if you have set access for *.*.*.

To delete an access identifier from an ACL, use "delete_acl" ("da"). For instance:

```
delete_acl mink Gold.DIGGER
```

removes Gold.DIGGER from the access list for segment mink. If you omit an access identifier, Multics assumes you mean yourself.

Mailboxes are also subject to access control lists. However, mailboxes have seven kinds of access modes instead of just three. "append" (a) allows users to send messages to a mailbox. "delete" (d) access lets users delete any messages from a mailbox; "read" (r) lets them read any messages in it; "own" (o) access allows users to read and delete only messages they have sent. "status" (s) access lets them find out how many messages a mailbox contains. "wakeup" (w) permits users to send interactive messages. (Messages sent by users who lack "w" status are shunted into the mailbox; they are not printed immediately.) "null" (n) access prevents users from having any access to a mailbox.

When you create a mailbox, your initial access is "adrosw". This allows you complete control. Everyone else, including *.SysDaemon.*, is given "aow" access--they can send you mail and interactive messages, and read and delete the mail and interactive messages they send.

To manipulate the ACL's on mailboxes, use the three commands "mbx_list_acl" ("mbla"), "mbx_set_acl" ("mbsa") and "mbx_delete_acl" ("mbda"). These work in the same way as the equivalent commands for segments and directories.

You can find more information about access control lists in the Multics Programmer's Manual--Commands and Active Functions (AG93), and in the Multics Programmer's Manual--Reference Guide (AG92).

MULTICS BACKUP SYSTEM

by Roger Roach

reprinted from the September-October Bulletin

The Multics backup system is actually two backup systems used together to provide the most comprehensive backup available on any commercial computer system. The backup system allows us to restore files in the event of hardware or software problems. And, it allows you to retrieve files that you have deleted or otherwise destroyed by accident. Here we explain how to use the backup system to retrieve files, and when you may want to supplement our backup system with one of your own.

The two backup systems are the hierarchy dumper and the volume dumper. Each of these has three modes of dumping (incremental, consolidated and complete) and each has a reloader and retrieval mechanism. The hierarchy dumper references data in a way similar to the method that you do. It uses the tree structure that links all segments and directories, and that Multics labels "hierarchy". The volume dumper references data by volume, or by the actual physical disk drive/pack on which the data (either segments or directories) resides.

The hierarchy dumper is the older of the two. It was available when Multics went public and is based on the CTSS backup facility. It works by scanning the hierarchy and dumping the directories and segments in alphabetical order (actually, ASCII collating sequence). When in incremental mode, it dumps only those segments (and directories) which have been modified (date/time modified) since they were last dumped incrementally (date/time branch dumped). It then updates the time. (The hierarchy dumper only updates this date/time when running in incremental mode, not in any of the other modes.) Currently we make one incremental pass with the hierarchy dumper each day. These tapes are saved for approximately one month.<1> The hierarchy complete dumper will dump all segments and directories in the hierarchy being dumped. For this purpose, the hierarchy has been divided into five sections. The user files are dumped once a week, usually on Friday night. These tapes are saved for approximately six months; however, the dumps done during the first week of each month are saved for a year. Also, the January complete dump is saved for five years.

The hierarchy consolidated dumper (sometimes called "catchup") dumps those segments (and directories) which were modified after a time supplied by the operator. Currently, IPS does not use this mode, but we may use it in the future to replace some of the weekly complete dumps.

Since the hierarchy dumper dumps files in hierarchical order and produces maps of what is dumped, these tapes are most useful for retrieving parts of the hierarchy that have been lost through deletion, or files that were destroyed more than five days earlier. This dumper is not efficient due to the nature of its scanning and is very clumsy when used to restore a damaged disk pack. For that reason, the volume dumper was implemented.

The incremental volume dumper scans each physical disk volume (disk pack) and dumps only those segments that have been modified on the volume since the last pass. This dumper makes a pass approximately once an hour and the tapes are saved for five days or until a complete dump is done, whichever occurs first. The consolidated volume dumper is used to dump those segments which were modified since the last consolidated volume dump. Currently we make a consolidated pass once a night and the tapes are saved for a month (back to the complete dump prior to the last complete dump). The complete volume dumps are done on the first and third weeks of each month, usually on Friday night. Two sets of complete dumps are saved, providing between 2 and 4 weeks of coverage.

The volume dumper does not generate maps. Instead, it keeps on-line tables of what disk volumes were dumped on what tapes and when. When a volume has to be restored, these tables allow the system to reconstruct the volume with a minimum number of tape mounts. Likewise, when a segment has been destroyed, these tables can be used to determine what tape was being used for backup on the volume on which the segment resides. Since the volume dumper was designed mainly for restoring a volume, its retrieval capabilities are not as complete as those of the hierarchy dumpers. For that reason, we recommend using the hierarchy retriever whenever the segment being retrieved has been in its desired state for more than 24 hours or whenever a hierarchy is being retrieved. If a segment has been damaged or if you need a particular copy of the segment that was only available for a few hours, then you should use the volume retriever.

To do a hierarchy retrieval, either go to Building 39 and fill out a Retrieval Request Form or use the "retrieval_request" ("rr") command. The hierarchy retriever requires the primary pathname to be used for all directories and segments specified in the retrieval

---

<1> Although Operations tries to maintain the schedule, circumstances may require changes. If you want to make sure a certain dump has been done, you should contact Operations at x3-7739.

request. For that reason, you should go to Building 39 and check the maps to make sure you have the correct pathname. The request (whether done with the "retrieval_request" command or on the form) must contain the pathname and the dump to be used (e.g., "complete dump from last weekend"). If you are retrieving a hierarchy, then put ">**" after the name of the directory. For example ">udd>Project>Username>subdir>**" would retrieve the contents of subdir. If the ">**" is not used, only the directory (not its segments) will be retrieved. Not very useful!

It is also possible to retrieve segments and hierarchies into new directories. This is called "cross-directory retrievals". To do this, you use an "=" followed by the new pathname. For example, to retrieve a file called "program.pl1" and store the retrieved copy in a file called "program.pl1.retrieved", you'd enter:

```
>udd>Project>Username>program.pl1=>udd>Project>Username>program.pl1.retrieved
```

You could retrieve a hierarchy originally in a directory called subdir and store it in a new directory called newdir by typing:

```
>udd>Project>Username>subdir>**=>udd>Project>Username>newdir
```

(Note, the ">**" is not used on the "to directory".)

In order to prevent unauthorized cross-directory retrievals, the Operations staff checks with the owner of the segment or hierarchy whenever the retrieval will go to a new user. This may delay the retrieval for a day or so while permission is being obtained. Otherwise, retrievals will be done twice a day on weekdays (noon and 8 PM) and once a day on weekends (about 2 PM). (The times are not exact and depend upon the workload of the operator.) Type "help retrieval_request" for more information on submitting on-line hierarchy retrieval requests.

The volume retriever is best used to retrieve a segment which has been damaged or for which the date/time modified is known. To submit a volume retrieval request, use the "enter_retrieval_request" ("err") command. You should leave the damaged segment on line so that the retriever can obtain the date/time modified from its directory entry. Usually you will want to use the "-previous" argument to err to specify that you want the previous version of the file retrieved. If the segment is no longer on line, you will also have to specify a time range covering the time it was last modified (use the "-from" and "-to" arguments). Otherwise, the retriever will mount the most recent tape and work backwards until it finds your file. Since we generate about 20 tapes a day, the operator may get worn out mounting tapes before the retriever finds your file.

The volume retriever attempts to notify you of what it has done. You may get a notice that a branch is being appended. This means that the retriever has found the directory entry for your file and has appended the entry to your directory. If you try to use the file before the "object is loaded" you will get an error message saying that the contents are not in the VTOC (Volume Table of Contents). This means that data has not been loaded onto disk yet. If the volume retriever cannot load your file, it will tell you that the object was not found. This might be due to an incorrect time range specification or to the tapes not being saved. Type "help err" for more details on submitting a volume retrieval request.

If you are dealing with super secure data and do not want to have your files put on our backup tapes at all, you may prevent the dumpers from backing up your files. The hierarchy dumpers respect normal access control. Therefore, you may set the ACL of the segments you don't want dumped to null for the appropriate SysDaemon. (Backup.SysDaemon for the incremental dumper, Dumper.SysDaemon for the complete dumper or *.SysDaemon for all SysDaemons including the IO SysDaemon.)<2> The volume dumpers, on the other hand, dump volumes and thus bypass normal access control. To keep them from dumping your files, use the "volume_dump_switch_off" command with either the "-incremental" or "-complete" arguments to specify that you don't want incremental or complete volume dumps for the file.

If you want to do your own backup, use "tape_archive". This command allows you to append and extract files from a tape much as you would do with a regular archive command. We feel that this is the best solution for the general problem of keeping files off line. If you are dumping a whole hierarchy (because, for instance, you are going away for a year) you may want to use "backup_dump". This is the same code as used by the hierarchy dumper. It is reasonably easy to use and uses Multics standard format tapes which are a bit more reliable than other formats. However, these tapes can only be read on a Multics system and you cannot add more files to the tape later. Also the code has less support than does tape_archive. If you want to use backup_dump, please see one of our consultants.

<2> See "Take Control", in this issue, for information about ACLs.

EC'S AND ALL THAT

This is a slightly modified version of an article that appeared in the October-November 1979 issue of Two Bits Worth, a newsletter published by the University of Southwestern Louisiana Computing Center. It is reprinted from the November-December 1980 Bulletin.

User TJefferson is determined to master "exec_coms" since he believes they can help him with some of his routine tasks. An "exec_com" ("ec") can be as simple as a sequence of command lines stored in a segment whose name ends in ".ec". In fact, TJefferson's first attempt consists of:

```
accept_messages
general_ready -set -inc_cost -total_cost
print_motd
check_info_segs
read_mail -immediate_messages
```

which he stores in a segment called "knowledge.ec". Then, whenever he wants to execute this sequence of commands, he simply types:

```
ec knowledge
```

Actually, this ec is so simple that he might have been better off simply defining an "abbrev" to represent this sequence of commands. However, since it's already an "ec", he decides to keep it and rename it "start_up.ec":

```
rename knowledge.ec start_up.ec
```

This name has a special meaning for Multics. Now, whenever he logs in (so long as he does not use the "-no_start_up" or "-ns" option), Multics will automatically run the start_up.ec for him. It will also rerun it whenever he does a "new_proc" or gets a fatal process error, which is sometimes more than he really cares for, but he is mainly satisfied with this technique.

However, one day while he has a great deal of work to do, he decides to compile a large PL/I program called "purchase" as an absentee job. He types:

```
pl1_abs purchase
```

He is fortunate that no one else is using the absentee facility at this time, so his compilation begins immediately. He is now free to continue with interactive tasks at his terminal.

However, something odd seems to be happening. A sequence of messages which he had been in the process of receiving from MLewis and WClark suddenly stops. He has not deferred messages; he does a "who" and notices that they are both still logged in, but his absentee job is no longer running. This cannot be correct; the compilation should have taken much longer. He knows that the record of what happened during the absentee job will be saved in an "absout" segment called (in this case) "purchase.absout", so he quickly displays this segment at his terminal screen. There are his missing messages; the absentee job has pre-empted them. In addition, the absentee job has invoked the "read_mail" command; since "read_mail" is totally confused by the commands which attempt to initiate the compilation, the remainder of the file consists of error messages.

Clearly the absentee job is also using TJefferson's start_up.ec. Now, while it might naively seem that an absentee job should be able to login without using any start_up.ec which may exist, this is not the case. What actually must be done is to construct the start_up.ec in such a way that it will not interfere with an absentee job.

After considerable research, TJefferson alters and expands his start_up.ec to look like this:

```
abbrev
&goto &2
&label interactive
accept_messages
general_ready -set -inc_cost -total_cost
&goto &1
&label login
print_motd
check_info_segs
read_mail -immediate_messages
&label new_proc
&label absentee
```

```
&quit
```

The items beginning with "&" are control statements which the exec_com facility recognizes; many of them are fairly self-explanatory. Others ("&1" and "&2") indicate the two arguments which Multics supplies to any start_up.ec and which describe the conditions under which the start_up.ec was invoked; "&1" may have the value "login" or "new_proc"; and "&2" may be either "interactive" or "absentee". Thus, you can use the "&goto" control statement to direct the exec_com facility to skip any inappropriate commands.

Thus, this start_up will turn on abbrevs regardless of the type of process; and if TJefferson gets a new interactive process (during a session in which he did not login with the "-ns" option), the start_up will also accept messages and tailor his ready message. But the expensive and time consuming portion of the ec will only be executed on his initial interactive login.

Ec's are useful in other situations, too. For example, to run TJefferson's program "purchase", it is necessary to use a temporary work file "work_file". To have this file automatically deleted on logout or at the end of a process, he creates the associated segment "purchase_work" in the process directory. So he writes a segment called "acquire.ec" containing:

```
io_call attach work_file vfile_ [pd]>purchase_work
purchase
&quit
```

However, TJefferson discovers that he is still doing a fair amount of typing because "purchase" asks him for input. Normally his responses do not vary greatly from one execution to another, although he is still experimenting with a few of the parameters to get the results he prefers. He decides to add his more standard responses to the ec itself and to pass the others in as arguments to the ec. This produces:

```
io_call attach work_file vfile_ [pd]>purchase_work
&attach
purchase
France
Monroe
&1
&2
&detach
&quit
```

The "&attach" statement causes "purchase" to read its input from the ec, and the "&detach" reverts to terminal input. Now TJefferson can control the entire execution of this program by typing, for example:

```
ec acquire 1803 15000000
```

TJefferson feels the amount of typing he must do is now quite reasonable. However, he believes that he has much more important things to do than sit at a terminal waiting for "purchase" to complete.

The solution? Run it "absentee".

And this is extremely simple to do. The major difference between an "ec" segment and an "absin" segment (one that can control an absentee job) is in the segment name, so all he needs to do is:

```
add_name acquire.ec acquire.absin
enter_abs_request acquire -arguments 1803 15000000
```

Since his segment now has both the name "acquire.ec" and "acquire.absin", he can run it either interactively or absentee without further change.

For additional information on ec's and absentee processing, see the MPM Commands and Active Functions manual (AG92). There are also a number of relevant "info segs". Try "help ec" and "help start_up.ec"; for a list of info segs of interest to absentee users, type "list_help abs". And for further uses of "&", type "help do".

SEND_MAIL FOR BEGINNERS

reprinted from the January-February 1981 Bulletin

Let's say you want to tell another Multics user something. One easy way is to send a message with the "send_mail" command. Just type "send_mail", followed by the userid of the recipient, and then answer the prompts for the subject and text of the message. You signal the completion of your message by entering a period (.) on a line by itself. For instance, you might enter (your typing preceded by =>):

```
=> send_mail Perkins.FIRKINS
   Subject: =>Saturday Night
   Message:
=> Howsabout a beer down at the Rontenac Rill?
=> .
```

But let's say that before you typed that final period, you reconsidered. Perhaps you feel a less flippant message would be more appropriate; perhaps you'd rather invite someone else. Fortunately for you, "send_mail" provides a way for you to change almost everything about your message after you've typed it and before you've sent it. Not only can you modify the message text, you can add or delete recipients, change the subject heading, request an automatic acknowledgment, and do many other things.

Just as the familiar Multics command environment accepts Multics commands, the "send_mail" environment accepts send_mail commands, called requests. You can enter the send_mail environment, or request loop, by substituting "\q" for "." when you finish entering the text of your message. You can list the send_mail requests by typing "?" from the send_mail request loop, and you can find out more about individual requests by typing "help <name of request>".

For instance, let's say you wanted to edit your invitation to Perkins. You could enter the "send_mail" request loop, and then call up the "send_mail" editor, as follows:<1>

```
=> Howsabout a beer down at the Rontenac Rill?
=> \q
   send_mail: => qedx
```

As you might guess, the send_mail editor resembles qedx closely. With both, you alter character strings with an "s", display lines with "p", insert text with "i" or "a", and so on. There are only two major differences. When you're using the send_mail editor, you don't need to read a segment into the buffer, because the editor automatically reads in the text of your message. And, it saves all changes automatically, too, so you don't need to use the "w" request. You could change your message as follows:

```
   send_mail: =>qedx
=> p
   Howsabout a beer down at the Rontenac Rill?
=> s/Howsabout/Would you join me for/
```

When you've finished correcting your message, you can dispatch it with the send_mail "send" request. However, you must issue this from the send_mail request loop, not from the send_mail editor. Type "q" to re-enter the loop from the editor. (You can always tell when you're in the send_mail request loop because of the send_mail prompts--there are no prompts when you are using the send_mail editor.)

If the message text is like a letter, the message header (the lines that describe the message's origin and destination) is like an address. The "send_mail" command generates the header automatically, but you can change it. By default, the header consist of four lines that give the date the message was sent, its subject, its recipient, and its sender. You can print the header of your message with the "print_header" ("prhe") request. (To print the text of the message, use the "print" request; To print both the message and the header, enter "print -header".)

```
send_mail: =>prhe
Date:  5 January 1981 10:17 est
From:  Splash.CRASH
Subject:  Saturday Night
To:  Perkins.FIRKENS
```

You can direct your message to more than one user with the "to" request. Typing "to" followed by a userid adds that userid to the "To:" header line, and, of course, causes the message to be sent to that user's mailbox, when you issue the "send" request. For instance:

<1> You can enter the editor directly by substituting "\f" for the final ".".

```
send_mail: =>to Blink.BLUNK
```

To view that line, type "to" without any arguments:

```
send_mail: =>to
To: Perkins.FIRKENS, Blink.BLUNK
```

Another way to add recipients is with the "cc" request. Intended to mimic the function of conventional carbon copies, this request adds another line (the "cc:" line) to the header, and sends the message to the specified destinations. It takes the same syntax as the "to" request: "cc" alone lists the line; "cc" followed by a userid appends that name to the header.

The "from" request (again, same syntax) appends userids to the "From" line, useful if you want to imply that more than one person is responsible for the message. However, if you use this request, "send_mail" will add yet another line to the header--the "Sender:" line, which gives your userid only. This assures that the recipient will know who actually sent the message.

To delete userids from the header, use the "remove" request. "remove" followed by a userid removes that userid everywhere it appears. Or, you can delete a name on a single line by typing, for instance:

```
send_mail: =>remove -to Blink.BLUNK
```

You can replace, but not add to, the "Subject:" line by following the "subject" request with the desired text:

```
send_mail: => subject "Saturday Night Bash"
```

Once you're satisfied with the message text and the header, you should send the message. You can do this by typing just "send"--but you can also append any of several "send" control arguments. These perform many functions. For instance, the "-acknowledge" argument will cause a message to be sent to you as soon as the message is read. (It also adds another line to the header.) Other control arguments ("-log" and "-save") store copies of the message in your auxiliary mailbox. And, if you follow the "send" request by a userid, the message is sent only to that person. The userid is not added to the header, nor is the message dispatched to the other users specified in the header.

You remain in the send_mail request loop even after you've issued the "send" request. This allows you to modify the message further, a useful feature if, for instance, you want to send slightly different messages to several people. To exit the send_mail request loop, type "quit".

If you decide you don't want to send the message after all, you can, of course, type "quit" before you issue the "send" request. But let's say you send the message and then realize that it contains an embarrassing error. Are you doomed to watch the news of your gaffe spread through the office? Not at all. Mailboxes are set up so that, by default, you "own" any messages that you send. In other words, you can use the "read_mail" command to process a message in another user's mailbox *if* you have sent that message. For instance:

```
=> read_mail Perkins.FIRKENS
   You have one message.
   read_mail: =>delete 1
   read_mail: =>quit
```

There are many things we haven't mentioned about "send_mail". For instance, we didn't explain how to use it to mail text already in a segment (the "-if" control argument). And we didn't explain how most of the requests that modify the header can be used as control arguments of the "send_mail" command. And we didn't even touch on advanced requests, such as "apply", which, for instance, allows you to edit your message text with "emacs" rather than "qedx". You can learn about these capabilities in the pre-publication draft of the *Mail Systems Users' Guide*, which details all of the Multics mail facilities; you should also read the the *Multics Programmer's Manual: Commands and Active Functions* (AG92). Both manuals are available in the Publication Office, Room 39-233.

ACTIVE FUNCTIONS

by Barbara Hughes reprinted from the March-April 1981 Bulletin

While frantically searching for an obscure control argument to a command you needed, you may have noticed that the full name of the thick volume you were using is the Multics Programmer's Manual--Commands and Active Functions. Commands you know about; but what are active functions?

Like a command, an active function is actually a program. But unlike commands, which direct the operating system to perform some task, active functions return values; the command then uses these values for its arguments. For example, suppose user LBMayer.MGM is working in a subdirectory, and wants to execute program "movie_star", which is in his home directory. Rather than having to change directories or type out the long absolute pathname, our user can just enter the command "[home_dir]>movie_star". The active function "home_dir" returns the value of LBMayer's home directory, which is substituted into the command line, giving our user just what he needed, ">udd>MGM>LBMayer>movie_star". And, since the active function "home_dir" has an abbreviation (as do most active functions), LBMayer.MGM can further simplify his typing to "[hd]>movie_star".

As you have just seen, active functions are most often found in active strings--character strings surrounded by square brackets. Active functions can be nested within an active string, so that user LBMayer, who now wants to capitalize the headline found in his home directory segment "hollywood.gossip" and see what it looks like, can type "string [upper_case [contents [hd]>hollywood.gossip]]". The innermost active function is expanded first, so the sequence of value substitutions becomes:

```
string [upper_case [contents [hd]>hollywood.gossip]]
string [upper_case [contents >udd>MGM>LBMayer>hollywood.gossip]]
string [upper_case Clark Gable loves Carole Lombard]
string CLARK GABLE LOVES CAROLE LOMBARD
```

The command "string" then prints its argument CLARK GABLE LOVES CAROLE LOMBARD on LBMayer's terminal.

Active functions can also be concatenated. Suppose you needed to find the average of two numbers, stored in segments "first_number" and "second_number". You could use the "string" command with nested and concatenated active functions to find the answer; for example: "string [quotient [plus [contents first_number] [contents second_number]] 2]". If first_number were 16 and second_number were 12, the command would be expanded as follows:

```
string [quotient [plus [contents first_number] [contents second_number]] 2]
string [quotient [plus 16 12] 2]
string [quotient 28 2]
string 14
```

There are some additional dimensions to the syntax of active functions,<1> but for now, let's move on to the more interesting question, what are active functions good for, anyway? For one thing, they can save you a lot of typing. If, for instance, you are carrying on a lengthy conversation via the message facility with user FredericktheGreat.Prussia, you will quickly get tired of typing his user-id on each line. Instead, use the "last_message_sender" ("lms") active function, and reply to the message:

```
From FredericktheGreat.Prussia 02/16/81 1436.7 est Mon:
Wie geht es Ihnen?
```

with the line:

```
sm [lms]
Input:
Sehr gut, danke.
.
```

Using active functions in abbreviations can also reduce your typing. Suppose you have a set of ten FORTRAN programs which are frequently changed, and recompiled as a group. Rather than typing ten commands each time, you could build a segment "prog.names" which contained the names of your ten programs, and then define the following abbreviation:

<1> For example, the normal expansion of active functions can be partially or fully inhibited. Also, some active functions can take control arguments, and some require their arguments to appear within quotation marks, whereas others strip quotation marks off. A good introduction to the basic syntax of active functions can be found in the New Users' Guide to Multics, part II, section 3. Details on specific commands are documented fully in the MPM Commands and Active Functions manual.

```
.ab fort do "fortran ([contents prog.names]) -table -map"
```

Then, whenever you type "fort", all ten programs will be compiled with the desired options. Also, if you should later want to add an eleventh program to the set, you need only add its name to the segment "prog.names"--and the "contents" active function will return all eleven names when the abbreviation is next used.

The ability of many active functions to return current information leads to additional uses. Consider this line from the start_up.ec of user Absent_minded.Professor.

```
&if [and [equal [month [date]] 5] [equal [day [date]] 12]]
&then &print It's your mother's birthday. Call her!!
```

On May 12 of each year, the active function "date" will return the value 05/12/yr, "month" and "day" will return 5 and 12 respectively, each "equal" will return the value "true", as its comparison succeeds, "and" will return "true" since both its arguments are true, and the professor will be reminded to telephone her mother and wish her a happy birthday.

Active functions can also query users and use their responses as arguments to commands or other active functions. Suppose you were writing an exec_com which would give the user the choice of having a program take its input data either from his terminal or from any segment. This can be accomplished with the single command:

```
&if [query "Do you wish to enter data from a segment?"]
&then io_call attach file05 vfile_ [response "Enter segment name" -non_null]
```

Active function "query" returns true or false depending on whether the answer supplied is "yes" or "no" (any other answer generates an error message and a repeat of the query). If the answer is "yes", the exec_com requests that the user "Enter segment name". The segment name supplied replaces the function and completes the command, attaching the switchname file05 to the desired segment. The control argument "-non_null" requires the user to enter something; otherwise an error message is printed, and the request is repeated.

These examples indicate only a fraction of the uses of active functions. Over one hundred active functions are listed in section 2 of the MPM Commands and Active Functions manual, indexed under a number of major groupings. They can perform basic arithmetic (e.g., plus, mod), character string manipulation (collate, reverse), numeric mode conversion (binary, octal), and they can return values for current or past dates and times (day_name, minute). Active functions do logical operations (greater, not), pathname manipulation (entry, suffix), ask questions (query, response), provide storage system names (files, process_dir), and supply user/process information (have_mail, user). They are particularly powerful when used with the command "do" in abbreviations, and they give the exec_com commands nearly the flexibility of a full programming language. Take the trouble to get acquainted with the range of Multics active functions; it will save you time later on.

ARCHIVES: LIVE AND ON TAPE

by Barbara J. Hughes

reprinted from the May-June 1981 Bulletin

Are your Multics storage charges beginning to resemble the national debt? Do you have so many segments that the "list" command takes ten minutes to complete? Have you forgotten which of the segments named "something.data" goes with which program? Are you going away for the summer and need a cheap, easy way to store your segments? Do you have flat feet and a tired aching back? For all but the last problem, the Multics archiving commands may be your best solution.

ON-LINE ARCHIVES

Storage for your segments is allocated in records, each of which holds 1024 words of information. If your segment is 1100 words long, it still needs two full records of storage and you are billed accordingly, even though most of that last record is not used. If you have a number of relatively short segments, this wasted space at the end of each can become a significant portion of your record quota and your billing costs. The "archive" command merges a specified list of segments into a single lengthy one, eliminating the wasted space while allowing you to retrieve intact any desired segments. Archiving has other advantages as well. It allows you to group related segments under a single descriptive name, shortening and organizing the directory list. Also, archiving a set of object segments is a necessary preliminary to binding them, an operation which establishes and saves the linkages between the programs, making their execution much more efficient. (For information on binding, see the "bind" command in the Multics Programmer's Manual: Commands and Active Functions).

The basic syntax of the "archive" command is:

```
archive key archive_path paths
```

"Archive_path" is the name of the archive segment with which you are dealing. This name always has the suffix ".archive", but you may omit the suffix on the command line. The "paths" are the names of the segments to be appended, replaced, updated, deleted, or extracted from the archive, and "key" specifies which of these operations is to be performed on the paths named.

The keys which are used with the "archive" command are documented at considerable length in the MPM Commands and Active Functions manual, under the "archive" command. In brief, a key of "a" appends, or adds, new components to the archive. (Once segments have been added to an archive, they are called "components".) The "r" key replaces components in the archive with segments of the same name. If the component is not already in the archive, it is added to it. If the archive segment itself does not yet exist, the first use of the "a" or "r" keys creates it, and a message is printed to that effect. Update, the "u" key, operates like "r" except that replacement occurs only when the corresponding segment was modified more recently than the archive component of the same name; the system determines this by comparing their date-time fields, which are generated automatically when a segment is created or changed. The "d" key deletes the named components from the archive, and the "x" key extracts components from the archive and places them in segments in a directory. Also, "archive" has a table-of-contents operation, with a key of "t", that prints information about named components in the archive, or, if no components are named on the command line, about every component.

Additional keys can modify the actions of the append, replace, and update operations. A "c" key preceding the basic "a", "r", or "u" keys, directs the archive command to carry out the specified operation on a copy of the archive which will be created in the current working directory. The original archive is left unchanged. A "d" key following the basic key deletes from the directory all segments that were successfully added to the archive (using append, replace, or update), provided they were not protected by a safety switch. A "df" key following the basic key forces deletion of added segments, whether or not they were protected. These additional keys can be used in combination, so that a key "cadf" would copy the archive into the working directory, append to it the specified files, and then force the deletion from the directory of the segments which were appended. The table of contents, delete, and extract operation keys can also take certain modifiers. Note that the deletion feature just described is used only in conjunction with the append, replace, or update keys, and deletes segments from the directory when they have been successfully archived. The basic delete operation, specified by a stand-alone key of "d", deletes components from the archive rather than segments from the directory. The "archive" documentation covers all the keys fully and gives a lengthy example of the creation and manipulation of archives, so only a brief example will be given here.

Suppose you have a main program called "main", three subroutines called "one", "two", and "three", and a data file named "data" that is used by these routines. They are all short,

and you decide to archive them to save space. Type:

```
archive ad old_bones main one two three data
```

Multics responds:

```
archive: Creating >udd>Jurassic>pterodactyl>old_bones.archive
```

You then list your directory to verify what has happened, and see:

```
rw    4   old_bones.archive
```

Note that the original five segments occupied at least five records of memory (one for each) and the archive now requires only four records. Now suppose you later develop a new subroutine "four" which you want to test with the main program and its data. You extract those two components from the archive:

```
archive x old_bones main data
```

and discover a bug in the "main" program, which you correct and recompile. Now that your tests are successful, you want to replace the buggy copy of "main" with the correct one and add routine "four" to the archive. To do this, type:

```
archive rd old_bones main four
```

Note that the "r" key replaces the archived copy of "main" with the segment from the directory, appends "four" to the archive since it did not already exist there, and deletes the directory copies of both. There was no need to specify "data" in the archive command, since it had not been changed and the extract operation makes a copy of the archived component in the directory, but does not alter the archive itself.

You can verify this with a table-of-contents query:

```
archive t old_bones
```

The system responds:

```
        >udd>Jurassic>pterodactyl>old_bones.archive
  updated                    name
05/17/81   1432.2          main
05/11/81   0915.6          one
05/11/81   0915.6          two
05/11/81   0915.6          three
05/11/81   0915.6          data
05/17/81   1432.2          four
```

You can, of course, do many other things with this archive segment, including using it as a component of another archive, thus creating an archive of archives, but there is one thing that you should NOT do. NEVER edit or otherwise alter the structure of an archive with any command other than those designed specifically to manipulate archives.<1> The archive segment contains certain header information which identifies the beginning and end of each component, and if these are changed without the corresponding changes being made in the header, the archive will be left in a chaotic state. Otherwise, the archive is just another segment, and can be copied, printed, dprinted, etc.

Unfortunately, the archive system has one serious limitation--it does not support the starname convention with the append, replace, delete, and update keys. If, for example, you want to create an archive of all your PL/I source programs, you must specify each individually:

```
archive a stegosaurus_pl1_progs head.pl1 spikes.pl1 legs.pl1 tail.pl1
```

Alternatively, and more simply, use the "segments" active function, which returns the pathname of all directory segments that match a given starname:<2>

```
archive a stegosaurus_pl1_progs [segments **.pl1]
```

TAPE ARCHIVES

But what if you have so many segments that even archiving isn't going to make a significant dent in your bill, or you are going away for a lengthy period of time and want a

---

<1> The "archive_sort" and "reorder_archive" commands are documented in the MPM Subsystem Writer's Guide (AK92).

<2> See the March-April Bulletin article on active functions.

cheap, safe, and convenient way to store all that information? Look into the "tape_archive" command, which operates like "archive" but creates the archive on magnetic tape rather than within the disk storage system.

A tape archive consists of one or more reels of magnetic tape (referred to in the documentation as a volume set), referenced through a table maintained in your working directory. The tape archive system is designed so that the commands which reference or alter the volume set are accumulated in the tape archive table before being physically applied to the tape. Thus, "tape_archive" commands can be issued and verified before the tape is even mounted. If this table is accidentally deleted, it must be loaded from the tape itself before other commands can be issued.<3> The tape archive table always has the name of the archive with the suffix ".ta" appended. When you have entered all the desired directives, you issue the "tape_archive" command with the "go" key, the tape is mounted if not already up, the requests which are pending in the tape archive table are carried out, the table itself is copied onto the volume set, and the tape is dismounted, all without further user intervention. The tape archive table is created automatically the first time you issue a "tape_archive" command that appends or replaces a component of the archive. When you first specify the "go" key on a new tape archive, the system asks for the slot number of the tape on which the archive is to be written; thereafter, that slot number is stored in the table and the proper tape automatically mounted.

The basic syntax and keys of the "tape_archive" command are very similar to the "archive" command.

```
tape_archive key table_path {arguments}
```

Keys "a", "r", "u", "x", and "d" perform append, replace, update, extract, and delete operations on the tape archive, just as they did on-line archives. The "d" and "df" modifier keys can also be concatenated with the append, replace, and update keys to delete or force-delete the corresponding segments. For all these keys, the "arguments" of the command line are the names of the components being referenced.

For example, suppose you want to create a tape archive called "extra_bones" containing segments "able", "baker", and "charlie". Enter:

```
tape_archive a extra_bones able baker charlie
```

The system responds:

```
tape_archive: Creating >udd>Jurassic>pterodactyl>extra_bones.ta
```

If this is all you want to do right now, tell "tape_archive" to actually write the archive on tape by using the "go" key:

```
tape_archive go extra_bones
```

Since this is a new archive, the system replies:

```
Enter volume name of new first volume
```

When you give a valid slot number, the tape is mounted, the requests pending in the table are carried out, and the tape is automatically dismounted.

Suppose you later want to retrieve segment "charlie", delete "able", and add a new segment "delta". Issue the commands:

```
tape_archive x extra_bones charlie
tape_archive d extra_bones able
tape_archive a extra_bones delta
```

But let's say that you really meant to add segment "easy" rather than "delta". All is not lost; you can cancel the pending request for "delta" and give the proper command:

```
tape_archive cancel extra_bones delta
tape_archive a extra_bones easy
```

To make sure that everything is all right now, ask for a table of contents of the tape archive:

```
tape_archive t extra_bones
```

<3> This is done with the "load_table" key of the "tape_archive" command. The "reconstruct" key, which is documented within "tape_archive", was never implemented and is, therefore, unavailable.

Multics responds:

```
3 components in extra_bones.ta;   3 pending requests.
Mount of volume set for write pending.

REQ     COMPONENT
d       able
        baker
x       charlie         (into >udd>Jurassic>pterodactyl)
a       easy            (from >udd>Jurassic>pterodactyl)
```

All is as it should be, so you use the "go" key, and this time the tape is mounted without you having to give its slot number.

In general, you do not have to concern yourself with the fact that your archive is being written onto a tape rather than a disk; the system takes care of format, density, and other tape specifications. One thing you should be aware of is that after a number of changes, your archive tape is likely to contain a significant proportion of waste space. Because of the nature of magnetic tape, files which are deleted or replaced cannot be physically removed without rewriting the entire tape; therefore, they are marked as deleted in the table and new versions (if present) are written at the end of the tape. When the wastage gets too large (this can be checked by using a table of contents key with the "-long" control argument) you may compact your tape archive by writing only the active files onto a new tape, which then automatically becomes your primary volume set. This is done with the "compact" key of the "tape_archive" command. Other keys issue warnings when the wastage reaches a certain level, or request that tapes be compacted automatically at preset times.

The tape archive system allows you to specify a number of other options besides those discussed here; all are fully documented under "tape_archive" in MPM Commands and Active Functions. For most users, however, the append, replace, update, delete, extract, and go keys, with an occasional compact, meet the need for a convenient and relatively cheap way to store large amounts of data in the Multics system.

WANNA START SOMETHING?

by Michael Thornton

reprinted from the May-June 1981 Bulletin

I recently attended the First Annual Saurian Conference for Office Managers. Most of it was boring, but one presentation really bowled everybody over. When it finished, the applause was riotous; throughout the audience, the sleeves of three-piece suits shook and the tailored outfits from Bonwit's shimmered as scaly claws slapped together enthusiastically. At the front of the auditorium, the speaker tried to look appreciative but could only smile smugly as he bathed in the response. Obviously he knew his speech on "Emacs as Employee-Relations Tool" was a real corker.

At last the roar died down, and a question-and-answer period began. I managed to scribble down as much of it as I could, and what follows is a transcript of my notes. Although the discussion is biased towards concerns that not all of our Multics customers share, there's a lot here on Emacs start_ups that may be of general interest.

QUESTION [from a brontosaurus who used expensive cosmetics]:
My staff has been using Emacs for several months now and I think it's great. However, my budget is limited and I don't have the money for high-speed modems. I've supplied 300-baud couplers and my staff is always complaining how long it takes for the screen to re-arrange itself when they insert or delete material in the middle of the text they're working on.

ANSWER:
You can deal with these complaints by having your secretaries make the screen size of their Emacs enviroments smaller. Your typical 24-line display terminal, for example, is apportioned by Emacs into a text area of 20 lines and a 4-line minibuffer. If you cut the text area by one half, the time required for the redisplay will be less and your secretaries will be happier. To set the text area to be, say, ten lines, have your staff, each time they invoke Emacs, type "ESC X" and answer the prompt by giving the following command:

```
set-screen-size 10
```

QUESTION:
Well, yes I see, but I suspect my staff would cavil some at having to type that same command each time they invoked Emacs.

ANSWER:
You can placate your staff on that account by providing each of them with an Emacs start_up. It automatically issues certain commands each time you invoke Emacs.

QUESTION:
How do I do that?

ANSWER:
Create a segment that contains the commands you want executed automatically. Make sure the segment is called "start_up.emacs" and is in the home directory of the person who is to use it. A simple one, which just sets the screen-size, contains just one line:

```
(set-screen-size 10.)
```

Enclose commands in parentheses. This is because an Emacs start_up is really a small LISP program, and LISP requires parentheses around each discrete statement. The period after the "10" also manifests a LISP convention. You must include the period to have Emacs interpret "10" as a decimal rather than an octal number.

Then the next time you use Emacs, it automatically finds the start_up segment and executes the instructions it contains at the start of your editing session.

QUESTION:
Thank you, that doesn't sound too hard.

ANSWER:
That's the wonderful thing about Emacs: if you don't like a particular aspect of its behavior, you can change it--without a major investment in new software or equipment. And that's what makes it such an outstanding employee-relations tool. It's so cheap and easy to satisfy complaints from your word-processing personnel. I'm sure there are others out there whose staffs have peeves comparable to the screen-size problem--

QUESTION [from a tyranosaur, yelling]:
Goodness, yes! Is there any way to get rid of those ugly \010's that indicate backspaces in an underscored word?

QUESTION [also shouted]:
Yeah! And can't you get it to automatically insert a carriage return when you're typing a line that goes past the physical right margin of the screen instead of printing those confusing \c's?

QUESTION [in a mimicky voice]:
And can't you stop it from messing up the screen when my girlfriend sends me Multics mail?

ANSWER:
Hold on, folks. All those complaints can be dealt with by commands given in an Emacs start_up. Consider the following additions to the simple start_up we made earlier:

```
(set-screen-size 10.)
(opt 'suppress-backspace-display 'on)
(fillon)
(accept-messages)
```

All these Emacs commands--or LISP statements, if you want to think of them that way--set aspects of the Emacs environment that, once initially set when the start_up executes, persist throughout the editing session. Each line corresponds to an Emacs command that you could give interactively by pressing keys or by name after typing "ESC X". However, keep in mind the differences in how you give a command interactively and how you present it in a start_up. I explained the need for the parentheses and the period before; you need the apostrophes in the second line for similar reasons. An apostrophe tells LISP that the item immediately following is a literal name; i.e., it's a string constant rather than the name of a variable. The effect, by the way, of setting the "suppress-backspace-display" flag, is that Emacs displays overstruck characters more compactly. For example, the underscored word Hypocrisy appears as:

```
H__y_p_o_c_r_i_s_y
```

rather than:

```
H\010__\010y_\010p_\010o_\010c_\010r_\010i_\010s_\010y
```

The "fillon" command (on the third line) tells Emacs to automatically break long lines by inserting newline characters rather than wrapping the lines on the screen and printing "\c" to show that what you see is actually a single line. The "accept-messages" command activates Emacs's own message-processing system; even if you do not use it for sending or reading mail, having it in effect is nice because it prevents interactive messages from messing up your screen.

QUESTION:
But don't these longer Emacs start_ups leave you sitting around for quite a while waiting for them to finish? My staff already complains about how long it takes Emacs to get itself revved up.

ANSWER:
Well, four lines are not going to slow you down substantially. As for longer start_ups, there is a way to make them run more efficiently. To begin with, you should package the constituent commands together as a single LISP procedure (called a "function") and then execute the function. Recast in this way, the start_up given above looks like this:

```
(defun start-things-up ()
       (set-screen-size 10.)
       (setq suppress-backspace-display t)
       (fillon)
       (accept-messages))

(start-things-up)
```

You recognize four of the lines from before. I've inserted them as part a LISP function definition, which is introduced by a "defun" statement. The function defined here consists of the four operations that we want performed by the start_up. What follows the "defun" is the name to be given to the function, in this case "start-things-up". The "()" indicates that the function takes no arguments. The lines that follow are the set of things to do; "accept-messages" is the last of the set because the extra right parenthesis on that line closes the statement begun by "(defun". (A statement of a LISP program, remember, is a sequence of things enclosed by parentheses, including, perhaps, other sets of things in parentheses.) The final line in the start_up tells Emacs to execute the function that was just defined.

Now the start_up above, if stored as the segment "start_up.emacs", will work just fine. But to attain faster execution, you should compile your start_up. To compile it, you run a program named "lisp_compiler", which takes your start_up in the form you typed it (i.e., source code) and translates it into machine language (object code). If you have been using a non-compiled start_up, switching over to use of a compiled one can be

tricky because the object segment, rather than the source segment, must be named "start_up.emacs". Therefore you should rename your source file "start_up.emacs.lisp". When you run "lisp_compiler", the object segment it generates is given the name of the source segment with the ".lisp" suffix stripped off--which is exactly what you want. And how do you run the compiler? Give the command "lisp_compiler" (or "lcp"):

```
lisp_compiler start_up.emacs.lisp
```

If you get messages like "(accept-messages fillon set-screen-size) - functions referenced but not defined" or "Warning: suppress-backspace-display undeclared - henceforward assumed to be special", don't worry--these are normal. The compiler produces a segment named "start_up.emacs", which Emacs will pick up as it does an uncompiled start_up.

QUESTION:
My secretary has changed Multics so that his delete character is "<" instead of "#". Now Emacs picks this up automatically so that "<" invokes the "rubout-char" command, but he wants to accordingly have the "rubout-word" command invoked by "ESC <" instead of "ESC #". He's learned to use the "ESC X set-permanent-key" command to do this, but he frets constantly about having to give that command each time he uses Emacs. Isn't this the kind of thing you could do in a Emacs start_up?

ANSWER:
Of course it is! Start_ups are the perfect place to customize key bindings. Tell him to make a start_up as we've discussed and include the line:

```
(set-permanent-key 'esc-< 'rubout-word)
```

Again, notice the differences from giving the equivalent interactive command: you do not introduce the command with "ESC X" and you put apostrophes in front of any of the command's arguments that are character-string constants.

QUESTION:
All this talk about key bindings has got me wondering about the arrow keys on the VT100s we have. My staff is always saying how much easier it would be to move the cursor with the arrow keys than with the standard ^P, ^N, ^B, and ^F. Currently, if you press an arrow key, Emacs goes to the beginning of a paragraph and deposits a single letter there. Couldn't you write a start_up that would make these keys work?

ANSWER:
Yes, but we're getting into complicated territory. Basically, you have to do two things: write a new Emacs command (or "extension") and hook it to a key (as discussed in the previous question). You must design the command and choose the key binding so that they properly interpret and respond to the signals transmitted by the arrow keys.

When you press an arrow key on a VT100 terminal, three characters are transmitted to Emacs: an ESCAPE character, a "[", and a letter ("A", "B", "C", or "D", depending on the direction of the arrow). What happens now is that Emacs takes the "ESC [" as a signal to execute a command (and accordingly runs "beginning-of-paragraph", the standard command associated with that key binding) and then takes the letters as something you want to put in the current file. If you're willing to give up (or reassign) the "beginning-of-paragraph" command, you can instead arrange things so that "ESC [" invokes a custom-made Emacs command that reads the third letter and then, depending on which letter it is, executes one of the existing Emacs cursor-moving commands.

Now I don't want to get too involved in the issue of writing your own Emacs commands; if you want to get into that, see the Multics Emacs Extension Writers' Guide (CJ52). Let me just show you an Emacs start_up that defines a new cursor-moving command and hooks it up to the "ESC [" keys:

```
(%include e-macros)
(defcom move-cursor-via-vt100-arrow
        (setq x (get-char))
        (cond ((= x 101)(prev-line-command))
              ((= x 102)(next-line-command))
              ((= x 104)(backward-char))
              ((= x 103)(forward-char))
              (t (display-error "Undefined esc-[ command" (printable x)))))

(defun start-things-up ()
       (set-permanent-key 'esc-[ 'move-cursor-via-vt100-arrow))

(start-things-up)
```

The first block defines a new command called "move-cursor-via-vt100-arrow". The second defines the start_up function, which comprises one operation, the association of the new command with the "ESC [" keys. The last line, of course, executes the start_up function.

One last thing about Emacs extensions and start_ups: eventually you may write lots of extensions; rather than defining them all in your start_up source segment, you'll find it easier to keep them around as separate (possibly separately-compiled) segments. Your start_up can then use the Emacs "loadfile" command to make your extensions available to you.

QUESTION:
My terminal is not a VT100, but it has arrow keys. Will the strategy you just described work for other terminals?

ANSWER:
It depends. Check the user's guide for your terminal to check what signals its arrow keys do in fact generate. The up-arrow key on the terminal I use, for example, transmits a signal equivalent to the standard Emacs command "kill-lines" (^K). I could simply hook up "prev-line-command" with CONTROL-K to make the up-arrow key move the cursor up, but if I did I would disable the "kill-lines" command. I could then bind "kill-lines" to some other key, but then I'd have to get used to the new arrangement.

QUESTION [from a stegosaurus with stylishly-polished spikes]:
I've heard that an Emacs start_up must always have the "default-emacs-start-up" request as the last line so that Emacs does its usual starting-up procedures as well as particular ones you've added.

ANSWER:
That was true in the past, but is no longer. Emacs goes through its usual routine whether you have such a line or not. The Honeywell documentation, however, has not yet been updated to reflect the change.

QUESTION:
Documentation? There's documentation on all this?

ANSWER:
Oh, yes. See Appendix G of the Emacs Text Editor Users' Guide (CH27).

EXCERPTS FROM "OUR FAVORITE MULTICS QUESTIONS"

The following are composites of questions we have heard a number of times via the "online_consultant" (olc) command, together with the replies to them. They are excerpted from the Bulletin column "Our Favorite Multics Questions".

How to Copy a Segment from Someone Else

Q. I want to copy a program from my friend's account. How can I do it?

A. First of all, you should know that it may be unnecessary to copy the program. But whether it is or not, you will first need to know the pathname of the program, which is its address in the Multics storage system hierarchy. Most people keep things in their home directories, i.e., the initial working directory at login.<1> If your friend's Multics Person_id is JRipper and he is registered on the SURGERY project, his home directory is most likely:

```
>udd>SURGERY>JRipper
```

Anyway, you should ask him if the program is in his home directory; and if he says "Yes," or "What's a home directory?", you can be pretty sure it is. Otherwise, he'll tell you what subdirectory it's in. Suppose, though, that the program is in his home directory and its name is "find_victim". Then its pathname is:

```
>udd>SURGERY>JRipper>find_victim
```

Next, you need to get him to set you access to the program. Normally, you cannot read or write another person's files on Multics without explicit permission from him. This is a good thing; you don't want to come back after vacation to find that some fool has erased three months' worth of your work or just turned in a thesis based on your as yet unpublished data. However, this means that you need to go to a little extra trouble to share data. In order for you to use his program, JRipper has to put you on the Access Control List (ACL) for the segment "find_victim", with "read" ("r") and "execute" ("e") access. If your Person_id is BStrangler and you are registered on the VISITOR project, he would do this with the "set_acl" ("sa") command like this:

```
sa  find_victim  re  BStrangler.VISITOR
```

Or, if he doesn't mind who uses his program, he could just type:

```
sa  find_victim  re  *
```

OK, so now you know the address of the program and you have access to it. Why copy it? If it's a main program, you can use it just by typing its pathname. Or if it's a subroutine, you can use it by telling the Multics system its location with the "initiate" ("in") command:

```
in  >udd>SURGERY>JRipper>find_victim
```

If you intend to use the program a lot and don't want to keep typing its pathname, you can use the "link" ("lk") command to make a permanent link to the program in your working directory:

```
lk  >udd>SURGERY>JRipper>find_victim
```

This link will make it seem just as though the program were in your working directory. But if you are really insistent on having a copy of your own (and paying the storage charges for it), you can use the "copy" ("cp") command:

```
cp  >udd>SURGERY>JRipper>find_victim
```

This will make an actual segment in your working directory. In this case, you probably will want the source too, so make sure that JRipper sets you access to that, too; and then:

```
cp  >udd>SURGERY>JRipper>find_victim.fortran
```

<1> In a sense, this is unfortunate, because they could organize their work much more efficiently if they would create and use subdirectories. This is easy to do and allows you to separate groups of segments of different kinds or related to different tasks.

For more information on directories, segments, and access control, see the New Users' Introduction to Multics (CH24) and IPS Memo MS-1: Multics at IPS.

****

Likely Causes of Common Error Messages

Q. While running my program, I got the message: "Error: storage condition by bombout|3210 (line 99) (>user_dir_dir>BASEMENT>Bomber>bombout) Attempt to reference beyond end of stack. Stack has been extended." What could have caused this?

A. A number of things could have caused this error message, the most likely of which are using too much "automatic" storage, and referencing beyond the bounds of an "automatic" array.

Automatic storage is the default kind of storage in both PL/I and FORTRAN. In FORTRAN, automatic storage is used for variables that do not appear in COMMON or SAVE statements.<1> Each time your program is run, the space for these variables is re-allocated in a segment called the "stack" in a special temporary directory, the "process directory".

If you have too many or too large arrays of automatic variables in your program, the space needed to allocate them will exceed the maximum length of the stack segment and you will get the "storage" condition. If this caused your storage condition, you can type "start" and your program may run, because the maximum length of the stack has been increased. However, if the maximum length has not been increased enough, you may get the storage condition again. And if you do this several times, you will finally get a "Fatal process error"; i.e., the process directory and all your temporary storage will be thrown away, and you will be given a new process directory, just as if you had logged out and logged in again. A better solution than typing "start", then, is to remove your large arrays from automatic storage and put them into static storage (PL/I) or named COMMON blocks (FORTRAN).

Another major cause of the storage condition is a reference beyond the bounds of an array (or a string in PL/I). If the automatic storage reference is far enough out of bounds, it may be beyond the maximum length of the stack, and the result is the storage condition. Here, typing "start" will not solve the problem. You must look at your program and find out where the bad subscript reference is. If you compiled the program with the "-table" option, the error message will give you the line number (such as 99 in the example) where the bad reference is being made. Then, with a listing of your program (obtained by compiling with the "-map" option) and the "probe" command, you can examine the values of variable subscripts when the error occurs and find out its cause.

A third major cause of the storage condition is mismatched argument lists. This occurs when, for example, you call a subroutine that has four parameters with a "call" statement that has only three arguments. As a result, the subroutine finds garbage when it looks for the fourth parameter--and anything can happen. You should suspect this problem when you get a storage condition error from a Multics system subroutine or an IMSL routine. And, of course, the solution is to examine the documentation for the subroutine being called and the listing of your program, make sure that the argument and parameter lists match, and fix your program if they don't.

These three problems probably account for ninety-nine percent of all storage condition errors. Let us know if you find some new and creative ways of producing them.

Q. What does the message "record_quota_overflow" mean?

A. It means that you've run out of storage space. When your Multics project was set up, the User Accounts Office assigned it a maximum amount of space to use for directories and segments (25 records unless your project administrator requested more). This is the "storage quota" for the project, and is shared by all project members. (A Multics record can hold 4096 characters, or 1024 single precision binary numbers. This should not be confused with the term "record" used in FORTRAN or PL/I, which has a different, though related, meaning.) If, while you are using an editor or a program that writes into a segment, you create or increase the size of a segment so that the total storage space occupied by all the directories and segments of all the users on your project exceeds the project storage quota, you receive a message like:

```
Error: record_quota_overflow condition by fillerup|20
(>user_dir_dir>CONSUMERS>Spacehog>fillerup)
```

<1> This assumes that the program does not contain a SAVE statement with no variable names, a "%global static" statement. or a "%options static" statement.

```
referencing >user_dir_dir>CONSUMERS>Spacehog>more_space|0
```

In this message, the part after "Error...by" (in this case "fillerup") is the name of the program which was trying to do the writing when space ran out. The number after it is the octal location, within that program, of the machine instruction being executed. If you compiled the program with the "-table" option, you get a source line number after that. The next line gives the absolute pathname of the segment containing the program. On the last line, after "referencing", is the absolute pathname of the segment or multisegment file into which the program was trying to write. And the number following is the octal location within the segment at which it was trying to write.

How can you fix this? The easiest way is to use the "delete" command to get rid of segments you no longer need. You can find out how much space these segments are using with the "list" command, whose output looks like this:

```
Segments = 19, Lengths = 80.

r w    3  poisoned_apple
       .
       .
       .
```

Here the number "3" next to the segment name "poisoned_apple" means that the segment is using three records of storage. If you delete it, those three records become available to your program. Then you can just type "start" and the original program will resume. If you don't want to delete any of your segments, your project administrator will have to get the User Accounts Office (x3-4118) to increase your project's storage quota.

Of course the best policy is to get enough quota from User Accounts so that you won't run into this problem. Your project doesn't pay for any it doesn't use, but we still must limit the storage assigned to each project. We have to keep track of the total storage assigned so that we don't accidentally run out of storage space altogether! If you think you need more quota, first find out how much you are now using. Type:

```
get_quota
```

If the printed result says "quota=0", then you are borrowing from the storage quota of the next directory up in the hierarchy, usually your project directory. To find out the quota used in that directory (including what you're using), type:

```
get_quota  <
```

You should be able to estimate how many more records you need. Remember, if a segment contains anything at all, it is using at least one record. And since a record can only hold 4096 characters, if the segment contains even 4097 characters, it will need two records.

Q. I tried to compile my FORTRAN program, which used to work fine on the IBM system, and got this error message:

```
FATAL ERROR 414 on line 3
Implementation restriction: stack frame has overflowed its limit of 62000 words.
new_fortran: Fatal error. Translation aborted. stack.fortran
```

What's wrong?

A. The problem is that you are using up too much local, non-static storage in your program. All variables that do not appear in a COMMON or SAVE statement<1> are considered "automatic" variables. The storage space that holds these variables is allocated and freed dynamically with each run of the program (or on each call to a separately-compiled subroutine). And the place where this space is allocated is in a special segment called the "stack". To help your programs run more efficiently, the maximum length of the stack is kept at 65,536 words. And since system programs use 3,000-4,000 words at the beginning of the stack, it is obvious that if your program needs more than 62,000 words of automatic storage (one word holds a single integer or real value, or half a double-precision value), there won't be enough space in the stack to run it. The FORTRAN compiler can detect this problem under certain conditions and warns you about it.

So what can you do? Well, you have several options. One is to put a "%global static;" statement at the beginning of your source segment:

<1> Provided there is not a global SAVE statement in the program or an "%options static;" or "%global static;" statement in the source segment.

```
%global static;
C THIS PROGRAM IS OF LITTLE USE BUT GREAT BEAUTY
      DIMENSION SPCHOG(63000)
```

or a "%options static;" statement before in any of the subroutines that have especially large arrays of local (as opposed to COMMON) storage:

```
      END
%options static;
      SUBROUTINE CLEVER(TRASH,WISDOM)
C SUBROUTINE TO CONVERT TRASH INTO MEANINGFUL DATA
      DIMENSION FUDGE(70001)
```

These have the advantage of being easily identified for removal if you wish to export your code to a non-Multics system. Another way around the problem is to include a SAVE statement with no variable list in any of the subroutines where a "%options static;" statement would be appropriate. Or you can list variables in the SAVE statement to show that ONLY those variables are to be static (non-automatic):

```
dimension storage_hoard(30000),black_hole(40000)
save  storage_hoard,black_hole
```

If no variables are listed, as in:

```
dimension repository(50000),fort_knox(20000)
save
gold_brick=fort_knox(30)
```

then ALL of the local variables in the subroutine are considered static (including "gold_brick", for example). Or, of course, you could just put the big arrays in COMMON storage, preferably named COMMON.

One advantage of all these types of static storage is that they are allocated only once per process which saves some processing time. What could be a disadvantage is that any zeroings of variables or initializations in DATA statements are done only once, during the first run in a process. This possible disadvantage can be overcome by use of the "run" command (See the MPM Commands and Active Functions for further information), which causes static variables to be reallocated and reinitialized with each program run. Of course, the efficiency of a single allocation goes out the window in this case.

****

Hold Everything; It's an Interrupt!

Q. What does the "level 2" in my ready message mean?

A. It means that some program has been interrupted in the middle of execution and is waiting to be restarted. If you don't want to restart the program, type "release". If you are in an editor, like "qedx", and want to continue editing, type "pi" (program_interrupt). Or if you just hit the Break or Attn key by mistake, type "start" to continue.

"Level 2" refers to a Multics operating system subroutine called the "listener", which (through subroutine calls) prints the ready message on your terminal and then waits for you to type a command line. It reads each line you type, and as soon as you type a non-blank line, it passes the line to another subroutine called the "command processor". The command processor translates the line into calls either to system programs or to your programs. If, during execution, your program or a system program encounters some problem that it can't handle (such as the record quota overflow discussed earlier) or if you signal Multics to interrupt the program's execution with the "BREAK" or "ATTN" key, Multics does several things:

- stops executing the program;
- saves information about the interrupted program in a segment called the "stack" (in a special directory, the "process directory", which you normally don't need to worry about);
- prints out an error message (or the word "QUIT", if you did the interruption yourself);
- calls the listener again, to read a command line from your terminal.

The stack, which contains information about all programs either running or interrupted, now holds the following. (The program at the top of the stack is the one running; the rest are waiting.)

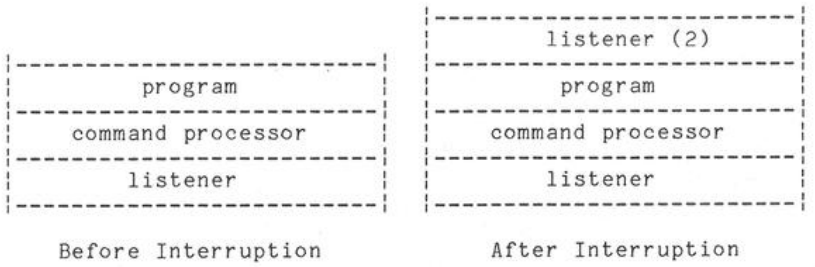

Figure 1. Stack Before and After Program Interruption

This is called a new "listener level", because the listener appears twice in the stack. Since the second execution of the listener knows about the first one (through shared static variables), it prints out a ready message with "level 2" to let you know that you have a program suspended and waiting to be restarted.

This second listener level is an excellent Multics feature not available on most systems. It allows you to issue commands to diagnose the cause of the error and sometimes fix it. Then you can return to the execution of the interrupted program without starting it all over again from the beginning. For example, if you had received the record_quota_overflow error, you could use the "delete" command to delete some segments:

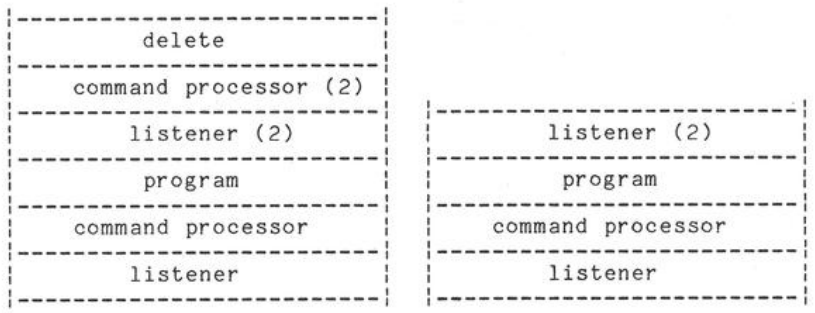

Figure 2. Fixing a Problem

and then use the "start" command to restart your program at the point where it was interrupted:

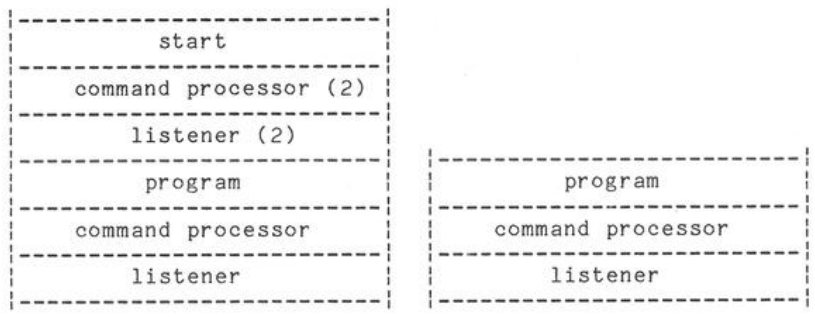

Figure 3. Restarting a Program

If you don't want to restart the program, you can throw away the information being saved in the stack by giving the "release" command:

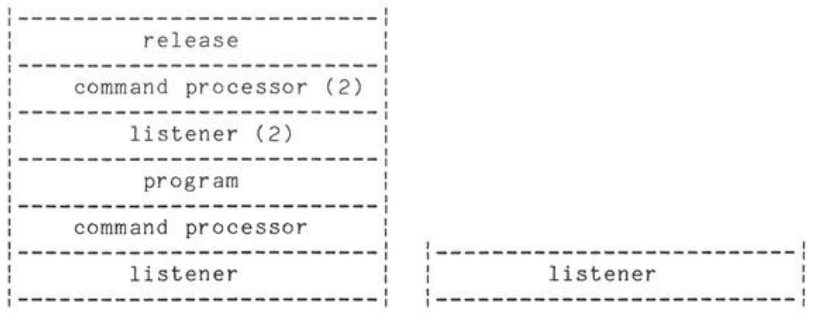

Figure 4. Releasing the Stack

And now you're back to the original listener level with no "level 2" in your ready message. Of course if you interrupt a program run from listener level 2, you get a ready message with "level 3", etc., and you may have to use "release" several times (or "release -all") to get back to level 1.

It's a good idea to release the stack if you don't want to restart the interrupted program. Sometimes (usually in your programs but also in system programs, such as "qedx") a second execution of the same program or a related program while a previous execution is suspended in the stack will cause a problem because of data shared between the two programs.

If programs such as editors (e.g., "qedx") and debuggers (e.g., "probe") that accept subcommands, or "requests", from the terminal are interrupted, it is often better to use the command "program_interrupt" (or "pi") instead of "start" to restart the program. This aborts the request being executed and puts these programs in a condition ready to receive the next request.

****

How to Join Segments Together

Q. How do I join together several segments into one large segment?

A. First of all, it should be clear that we are talking about ASCII segments, i.e., segments containing documents or source programs.<1> And we are also talking about joining them together and then leaving them together.<2> The most common way of doing this is to use the editor "qedx". For example, suppose you have three sections of the chapter of a book written by three different people and want to combine them into the finished chapter. You can do so like this:

```
qedx
r >udd>Writers>JBrown>joes_section.runoff
r >udd>Writers>MSmith>marys_section.runoff
r >udd>Writers>CChan>charlies_section.runoff
w chapter1.runoff
q
```

The first line gets you into the qedx editor. The next three read in the three sections of the manual. The second is appended to the end of the first, and the third is appended to the end of the second.<3> The next line writes the combined contents of the three segments into a fourth segment, "chapter1.runoff", in your working directory.

<1> There is a way of joining together object segments too: by use of the "archive" and "bind" commands.

<2> The "archive" command is used to combine several segments into one in such a way that any of the original segments can later be extracted easily. This is usually done for the purpose of organization and saving storage space.

<3> Of course, if all three sections were in your working directory, you could dispense with the ">udd...>" part.

Another method would be to use the AML "concatenate_segs" ("ccs") command.<1> You could achieve the same result as above with one command:

```
ccs chapter1.runoff        >udd>Writers>JBrown>joes_section.runoff
     >udd>Writers>MSmith>marys_section.runoff
     >udd>Writers>CChan>charlies_section.runoff
```

(all typed on one line). The only difference between these two methods is that if chapter1.runoff already exists, qedx erases its contents before writing in the contents of the other three segments whereas "concatenate_segs" appends to the current contents of chapter1.runoff. Also, "concatenate_segs" has control arguments that allow you to automatically delete the original segments after concatenation and do other useful things (See IPS Memo MS-52, The Author-Maintained Library, or type "help ccs").

****

Help! I'm Trapped in a Question to the OLC!

Q. q
end
\f
logout
How do I get out of this thing?

A. Although it is fairly easy to start transmitting questions to the on-line consultant (just type "olc", followed by a carriage return), people sometimes forget to find out how to stop transmitting; and the consultant sees something like the above on his terminal. To stop transmitting, just type a period all by itself at the beginning of a line, followed by a carriage return, like this:

.

If you don't, anything you type will continue to be sent to the on-line consultant until he or she is able to get a word in edgewise and tell you what to do, or until you hit the BREAK key or hang up the phone. While we're on this subject, notice the correct (and very similar) usage of the "send_message" (sm) command when you want to type the message on a different line(s) from the command itself:

```
sm Lumberperson.TREES
  <---[Multics responds "Input:"]
Please don't cut down any more trees on my account.
I'm converting to a CRT.
.
```

****

Using "Tape archive"

Q. How do I tell "tape_archive" what tape to use?

A. When you use "tape_archive" to create a new archive, for example, by using the "append" (a) key:

```
tape_archive  a  warehouse  kazonga.fortran
```

it creates a segment, in this case "warehouse.ta", in your working directory. This segment is used to control the tape archive and is called a "control segment".

You may continue requesting that segments be appended, e.g.,

```
ta  a  warehouse  use_kazonga.runoff
```

<1> The AML (Author-Maintained Library) is a collection of software written and maintained by our customers (as opposed to staff or vendors). It is documented in IPS memos MS-52, The Multics Author-Maintained Library, and AP-52, Author-Maintained Library: Application Software. Its support level is FOUR (See IPS memo RT-3, Hardware and Software Available at IPS, for an explanation of support levels, which affect availability of consulting, credit refunds, etc.).

and so on.

At this point you must have purchased or rented a tape and registered it at IPS (either by seeing the Tape Librarian on the second floor of Building 39, or by sending Multics mail to Tapes.Operator). When the tape is registered, you will be given a six-digit slot number (a "volume name" to Multics), like "071234", to be used in identifying the tape. Then, when you are ready to have the segments copied to tape, you can type, for example:

```
ta go warehouse
```

and "tape_archive" will ask you for the slot number:

```
Enter volume name of new first volume:
```

Type the slot number; "tape_archive" will then execute the requests you made earlier ("tape_archive a ..."), as they appear in the control segment, and copy the segments onto the tape. When it has finished and released the tape, Multics will print a ready message.

It's a good idea to try extracting a file from the archive at this time, just to make sure that the tape is good, e.g.,

```
ta x warehouse use_kazonga.runoff; ta go warehouse
```

You can then delete the segments, if you wish.

The next time you type "ta go warehouse", "tape_archive" will not ask you for a "volume name"; it has been stored in the control segment.

While you are using a tape for a tape archive, you cannot use it for anything else, including another archive (i.e., one with a different control segment) unless you are willing to destroy the contents of the archive.

****

Retrieving a Lost Segment

Q. I just made a terrible mistake and deleted a data segment that it took me three years of hard labor to construct. Is there any way to get it back?

A. Yes! If it had only taken you half an hour of hard labor to construct, you might be out of luck--but in this case, the Multics automatic backup system will save you. Every weekend the whole Multics storage system (all the directories and segments) are dumped to magnetic tape and the tapes are saved for a period of time (some for months; others, longer). And every evening all the directories and segments modified that day are dumped to tape. The weekend dump is called a "complete" dump and the daily one, an "incremental" dump. If you are near Building 39, you can fill out a Retrieval Request form at the dispatch counter (Rm. 39-260). Call 253-4121 first to be sure the dispatch area is open. Otherwise, type "help rr" to find out how to submit the request online with the "retrieval_request" ("rr") command. You will be charged $10.00 for each tape mounted in the retrieval (assuming the loss was your fault--otherwise, there's no charge). Retrievals are usually done within 24 hours.

If you have managed to create a very expensive segment and accidentally delete it the same day, you may be able to retrieve a good copy of it from the Volume Dumper tapes. The Volume Dumper once an hour scans the storage system and dumps to tape segments modifed since its last scan. However, since this dumper is somewhat less useful for retrievals than the one which creates the dumps mentioned above, you should avoid using it if possible. If you do need it, type "help err" to find out how to submit a volume retrieval request with the "enter_retrieval_request" ("err") command. More information on retrieval can be found in the article "Multics Backup", elsewhere in this issue.

Of course, if your data segment was so valuable it would have been wise for you to keep up your own backup tape by using the "tape_archive" ("ta") command (See the MPM Commands and Active Functions or type "help ta"--also see "Favorite Multics Questions" in the January-February 1980 issue of the Bulletin). In that case you would be able to retrieve a copy of the segment almost any time, usually for less than $10.00.

****

What Happens if You Just Hang Up

Q. I always get out of Multics by just hanging up the phone. Recently I noticed I'm being charged for additional connect time. Why?

A. The save_on_disconnect attribute was given to all MIT users in November, so Multics now saves all disconnected processes. This means that users who are accidentally disconnected can log back in and reconnect to their old process and not lose any of their work. Unfortunately, if you just hang up the phone, Multics thinks that you've been accidentally disconnected, and so it saves your process. It does not (as it used to) automatically log you out. Therefore, you should always leave Multics politely, by means of the logout command. Wait for Multics to reply before hanging up the phone or you might disconnect before the logout processing is complete and accrue additional charges. If you accidentally disconnect when you mean to log out, you should log in again, giving the "-destroy" argument on your login line. This destroys the disconnected process. If you're not sure that you have a disconnected process, you can login in the usual way. If you have one, Multics tells you so, and then asks you for instructions (see below). You can logout a disconnected process by answering "destroy".

Q. I lost two hours of editing yesterday when my kitten knocked over the phone and disconnected me. Today someone told me I could get the editing back by reconnecting to my old process. How do I do this?

A. Your disconnected process is saved only for about an hour, so the editing you did is lost forever. The next time your kitten knocks over the phone, you should login on the same project within an hour. If your process was successfully saved you receive the messages:

```
You have 1 disconnected process.
```

and:

```
Please give instruction regarding your disconnected process(es).
Please type list, create, connect, new_proc, destroy, logout, or help.
```

To reconnect to your disconnected process type "connect". (If you're sure you have a disconnected process, and you want to reconnect to it, you can do that most efficiently by specifying the control argument "-connect" when you login.)

Once you issue the "connect" command, Multics advises you to "Wait for QUIT." When QUIT is printed at your terminal, your process is in the same state as if you had issued the QUIT yourself by pressing the "BREAK" or "ATTN" key. You can type "program_interrupt" ("pi" for short) to retrieve your qedx or Emacs editing. Alternatively, you can type "start" or "release", depending on whether you want to restart or abort whatever you were doing at the time you were disconnected. Or you can just give a Multics command if you want to do something else from the new command level. For example, you might want to reset your terminal modes (see below).

However, the best way to protect yourself from editing losses is to save your editing frequently during your edit session.

Q. After I reconnected to Multics, my terminal was very slow and the lines were being truncated to 79 characters. What happened?

A. When you reconnect to a disconnected process, not everything about your process is saved. In particular, Multics does not remember any terminal modes you might have specified, such as line length, editing characters, and so on. Instead, it resets the terminal modes to the default for the terminal on which you are reconnecting. This is because users sometimes log back in on a different type of terminal than the one they were originally logged in on. When you reconnect, you should reset the terminal modes from the new command level and then restart or abort whatever you were doing at the time you were disconnected.

Emacs users beware: Emacs remembers terminal types. If you're disconnected in the middle of an Emacs session, you must always reconnect to the same type of terminal. Also, Emacs remembers terminal types between invocations during the same process (unless you informed it of the terminal type with the "stty" command). So, if you invoke Emacs again after reconnecting from a different type of terminal, you must inform Emacs of the change by giving the "-reset", "-query", or "-ttp" control argument with the "emacs" command.