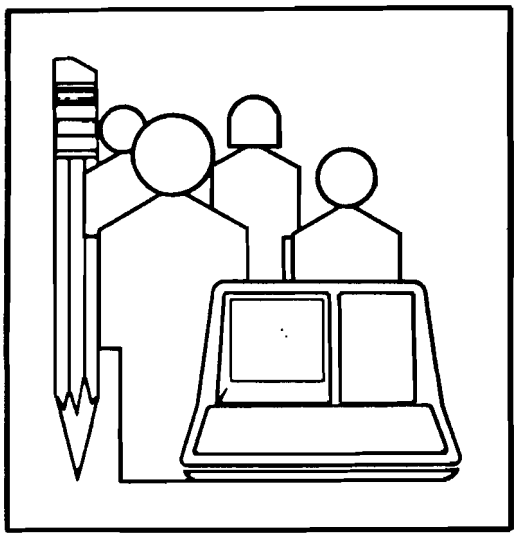

TECHNICAL REPORT

THE IMPLICATIONS OF WINDOW MANAGEMENT FOR COMPUTER-BASED TRAINING

Technical Report No. 10

August 1981

DIGITAL EQUIPMENT CORPORATION
EDUCATIONAL SERVICES
12 Crosby Drive
Bedford, Massachusetts 01730

digital

THE IMPLICATIONS OF WINDOW MANAGEMENT FOR COMPUTER-BASED INSTRUCTION

Michael J. Zimmerman
Jesse M. Heines

ABSTRACT

This paper represents the final report of a Computer-Based Course Development research project known as the Interactive Video Communication Network. It discusses the application of windowing and split processing techniques to the development of CBI courseware.

THE NEED FOR WINDOW MANAGEMENT IN CBI

The Development of EDTCAI

Early in the summer of 1980, Educational Services' Computer-Based Course Development Group in Bedford produced EDTCAI, an introductory, computer-assisted instruction (CAI) course that teaches how to use EDT, the DEC standard editor. The course serves to orient users to the basic features of EDT, including line mode, where text is entered line by line, and the new keypad mode, in which users see the document on their terminal screens and can modify it in much the same way that one would modify a paper document, e.g., deleting words, sentences, inserting new lines, and so forth. In EDT keypad mode, the keypad keys are associated with editing functions similar to those that had been previously developed for the KED editor on RT11 systems. EDTCAI not only provides an explanation of the editing function, but, in the context of exercises, allows users to "try" certain commands and then to watch their results as a simulated editor moves lines, deletes words, or does whatever the particular commands request.

We encountered several problems when analyzing students' input for the line mode exercises:

- First, there were usually a variety of abbreviations for the basic command itself.
- Second, there was an even wider range of possibilities for the syntax of the command with its reference to EDT line numbers.

The solution to these types of problems is generally to have a programmer construct a "parser" which would check the input answer against a table of acceptable values. This is, in fact, just what the editor does itself. The problem with this approach is that there is almost always another alternate variation which was overlooked and omitted from the table. In addition, as the EDT editor itself was developed in later versions, these syntax rules broadened, and in some cases, changed.

One consequence of this approach is that an accurate parsing table begins to approach the size of the real one used in the editor itself. Our answer to this problem was to make the exercises so specific that they began to lose credibility as a "model" of the editor. Nevertheless, the benefits of a graphic simulation resulting from an acceptable answer provided a strong conceptual basis for an understanding of the editor itself, assuming that the student was able to find an acceptable answer.

The difficulties in parsing line mode commands were further compounded when we attempted to simulate the visual keypad editor. Each key command, as it is pressed, results in a corresponding

action in the editor. Moving from the one dimensionality of line mode commands to the two dimensionality of keypad screen commands results in a dramatic multiplying of the parsing problem. Moving from one point on the screen to another can be accomplished in literally an infinite number of ways. Using the "seven league boots" of the keypad keys to move forward and backwards by words and lines provided a significant exercise in logistics. Namely, we had to keep a road map of what the text looked like and constantly calculate the position, because the action to be taken depended upon where one was when the keypad key was pressed. Now if we start to alter the text itself in any way, we have to revise and update the road map. This sort of programming begins to approach the functionality of the editor itself and indeed the editor is far better at doing that sort of thing than our CAI courses are, since that is all the editor is supposed to be doing.

Simulating the Editor

When you begin to consider the vast multitude of "undocumented features" in the real editor, subtle changes with new versions, not to mention the programmer's labor creating parsing tables and programming models, it was not a large step to put on our own "seven league boots" and try to use the EDT editor to "simulate" itself in keypad mode. The simulation centers around suitably restricting the functionality of the keypad editor keys so that dangerous and undesired keys have little or no effect. With one fell swoop we can furlough our programmer, synchronize editor version updates (the editor is always itself), and immunize ourselves against programming errors.

This jump is not without its consequences. In the real editor, there is little opportunity for feedback to the student other than directly showing the consequences of the editing action. Furthermore, if the student destroys the text beyond recognition, there must be ways to get help and begin again or return to the CAI instruction for more coaching. In addition, there is little way to know whether or not the student was successful, much less where and what problems arose with his or her editing session. This problem of passing back information is inherent in the VAX/VMS design.

A CAI course and an editor are both run in something called a "process" which has a process context called an "image". When VAX/VMS switches images (as it must in going from the course to the editor), it necessarily loses image context. It can only do one thing at one time and it has a very short term memory for what it just did. We can keep track of where in the course the student is working and file this information before moving into the editor. However, once in the editor there is limited ability to "branch based on feedback" or to do the thing that CAI does best. Students are, so to speak, on their own. This particular

approach has been implemented in the VMSCAI course due to be released with a future update of VAX/VMS.

One solution would be to control the visual output on the terminal so that the "context" remains at least in part on the screen, and the editor is restricted to, say, the lower right-hand quarter. Thus, the instruction, advice, and any helpful hints from the CAI course remain in the top and left portion of the screen with perhaps a picture of the keypad functionality for students to view as they work with the editor. This sort of solution is still on our wish list. It places special and specific demands on being able to do two separate things and merge the output. The implications of this are enormous. But for the moment, let us back up and consider a reverse situation, namely, moving from the utility image to its on-line documentation (HELP).

The HELP Analogy

Most software on VAX/VMS has on-line documentation (HELP) available which will coach users through a problem. In most cases, the information sought is presented in such a way that the user's previous context is not disturbed. In almost all cases, software utility HELP functions in an encyclopedic manner. The highest level lists the HELP syntax and its keywords. This is accessed by simply typing "HELP". The next level generally returns up to a full viewing page of information, with a possible second level of keywords, and so on. In most cases this suffices. (However, if the user is totally naive to the software and the system, he or she may not even know to type "HELP".) In EDT line mode, one must type "HELP CHANGE KEYPAD" to access the only reference (three levels deep) to the keypad HELP key (see Figure 1). Unfortunately, typing the more obvious "HELP CHANGE SUBCOMMANDS HELP" does not tell you which key to press (see Figure 2).

However, to enter the word "HELP" in change mode, you must be in nokeypad mode, and the diagram that HELP produces in nokeypad mode refers to keys that do not function in this mode. In keypad mode one must already know which key to press to access the diagram which tells you that that key is the one for HELP. The tacit assumption is that one has a hard copy of the keypad diagram at hand. Indeed, this is a fundamental problem in the design of keypads for on-line documentation and instruction, and indeed, this turned out to be a classic error in EDTCAI. The fact is that almost all EDTCAI users did not have the hard copy User's Guide available containing a keypad diagram. It is possible to generate the hard copy diagram in EDT, but one must be something of a DCL and EDT wizard to do so. Perhaps some thought is due to having on-line documentation (HELP) and instruction (CAI) generate their own hard copy, at least for things like keypad diagrams.

```
CHANGE

KEYPAD

You enter the keypad submode of change mode when your terminal is a
VT52 or VT100 and the KEYPAD option is on.  (This option is on by
default.)  In this submode, the terminal screen is used as a window
into the text buffer.  Characters typed on the main keyboard are
inserted into the buffer at the cursor position.  You enter editing
commands by using keys on the auxiliary keypad, or control keys on the
main keyboard.

For more help on keypad mode, type CHANGE to enter that mode.  Use the
keypad HELP facility as follows:

    1.  If your terminal is a VT100, press the keypad key marked
        "PF2".

    2.  If your terminal is a VT52, press the red keypad key.
```

Figure 1

THE HELP MESSAGE GENERATED BY THE COMMAND
"HELP CHANGE KEYPAD"

```
CHANGE

SUBCOMMANDS

HELP

The HELP command causes a diagram of keypad functions and CONTROL key
descriptions to appear on the screen.  If executed in keypad change
mode, additional information can be obtained by pressing keypad or
control keys; in nokeypad mode, pressing any subsequent key returns
to editing mode.
```

Figure 2

THE HELP MESSAGE GENERATED BY THE COMMAND
"HELP CHANGE SUBCOMMANDS HELP"

The moral here is that you can never assume the existence of available hard copy with CAI instruction. Yet the naive user desperately needs to have a bridge between the instruction and the software tool. CAI can provide that bridge explicitly if it can window the software tool into the terminal screen. Further more, if the two processes can function simultaneously, the input can be piped to both processes and, in the case in question, the editor would do its thing in the lower right quadrant, while the CAI course would provide the necessary feedback in the remainder of the screen.

These considerations led us to investigate the problems and practicality of windowing in a multiprocess environment.

THE DEVELOPMENT OF PROTOTYPE WINDOWING SOFTWARE FOR CBI

Window Management at Brown University

The Computer Science Department at Brown University had done extensive work with window management and split processes on a VAX 11/780 running under the UNIX operating system [1]. After tracing down various internal developmental projects on windowing and multiprocesses, we contacted Greg O'Brien who was working on a windowing problem with a RAND editor and who was also familiar with UNIX. We all made the trip to Brown University where Dr. van Dam and his students demonstrated the VAX/UNIX software on a RamTech display.

The question was whether this was possible on VAX/VMS with a VT100 terminal. UNIX is Bell Telephone's version of what an operating system should be like. VMS is, of course, DEC's system, specifically designed for the VAX 11/780. Although there are considerable differences between UNIX and VMS, it seemed likely that if windowing could be done on a VAX using UNIX, it should also be able to be done using VMS. We obtained many conflicting opinions on this question, and decided to pursue our involvement with Brown based on our belief that we could learn about windowing techniques from them while trying to resolve the VMS implementation problems within DEC.

---

[1] The bulk of this work was done by Steve Feiner, Norman Meyrowitz, and Margaret Moser under the direction of Dr. Andries van Dam.

Window Management on VAX/VMS

The problem really splits into two separate but related problems:

(1) how to split a process to effectively do two things at once and yet be in control of both, and

(2) how to control the process output or, in our terms, how to window.

To this effect, we wrote multiprocess software that enabled the user to window and control one or more additional processes or subprocesses from the user's terminal. Appendix A presents a DCL command procedure which controls the subprocess, causing (among other things) the subprocess to prompt with its name. Appendix B presents a DCL command procedure which extends the previous approach and allows the user to window the output at the terminal. If run from the main process, this DCL command procedure allows the main process to summon or sleep the various subprocesses. If run from a subprocess, it controls the output format from that subprocess by using the scrolling region feature of the VT100 terminal and windowing the first subprocess to the top half and the second subprocess to the bottom half of the terminal. A third or fourth subprocess would be presented in the bottom third or fourth of the screen with the other two outputting to their respective thirds or fourths.

Actually, both of these DCL command procedures are simplifications of a third pair of interconnected command procedures which used DECNET from a subprocess to login to a completely new process. This was the only way to create and control multiple interactive main processes. These two interrelated DCL command procedures allowed two or more entirely different users and user accounts running in main processes to share the same terminal each with its own horizontal strip similar to the output from the command procedure in Appendix B. However, recent changes with Phase III DECNET have restricted the variety of DCL commands that the user is able to use through the network from a subprocess without extending certain quotas or adding certain process privileges.

Although we will not present this last pair of command procedures, we will discuss certain aspects in their development, and copies are available from Michael Zimmerman upon request. The other software is presented in the Appendices and discussed below.

The VMS RUN command allows the user to create a subprocess to execute a particular image. (An image is an executable program.) In this creation process, the user can specify where the input data for that image comes from and where the output data stream and error messages are to go. These data streams are called SYS$INPUT, SYS$OUTPUT, and SYS$ERROR. They are "logical names"

and each refers specifically to a file.

When you first login to the system, the system creates a process and then runs an image called LOGINOUT which "sets up house" for your session. Process creation will assign the logical names (all those shown in Table] except SYS$LOGIN and SYS$COMMAND), but LOGINOUT makes the data streams "process permanent", that is, permanently open for the life of your process. LOGINOUT also adds one more logical name, SYS$COMMAND, which will have the same assignment as the original SYS$INPUT assignment.

LOGINOUT

It is LOGINOUT that actually prompts you for your Username and Password, then welcomes you to the system with a message, and modifies the created process with your username, process name, UIC, privileges, quotas, default directory (SYS$LOGIN), and a few other things. But most important, LOGINOUT maps a command language interpreter into your process space. This interpreter enables you to use a command language to talk to the system. Furthermore, your process does not terminate when the image exits (as it does with any other image). That is, LOGINOUT creates and assigns these logical names to process permanent files (TTA1 in the case above) so that the system can keep continuously track of where to get the input, where to send the output and error messages, and finally since overall monitoring and control is impor-

Table 1

CONTENTS OF PROCESS LOGICAL NAME TABLE

Logical Name	Actual Device
SYS$INPUT	_TTA1:
SYS$OUTPUT	_TTA1:
TT	_TTA1:
SYS$DISK	_DBA1:
SYS$COMMAND	_TTA1:
SYS$ERROR	_TTA1:
SYS$LOGIN	_DBA1:[USER]

TTA1 is a particular designation for the user's terminal, DBA1 is the system name for the user's disk, and [USER] represents the directory where the user is upon logging in.

tant in a continuous interactive process, where to go for help (SYS$COMMAND) in case something goes awry.

You can run LOGINOUT to map the command interpreter into the process space, specifying a DCL command procedure for the input file and a name for the output file:

```
RUN SYS$SYSTEM:LOGINOUT/INPUT=infile.spc/OUTPUT=outfile.spc
```

where "infile.spc" represents a DCL command procedure file, and "outfile.spc" is the name of an output file to be created.

A DCL or "command procedure file" is something like a program because the system will execute the lines in the file sequentially as if they had been typed at the user's terminal. However, each line in the DCL command file can execute its own image. When the last line of the command procedure is reached, the process logs itself out and stops. Although the subprocess is now gone, this still gives the user a way to execute a series of images in a subprocess.

Now suppose that the input command procedure discussed above would prompt the user for a DCL command, then execute that command, and then return again to prompt for another command. This would constitute an interactive looping command procedure as shown in Figure 3. If the user-supplied command itself requires user input, an extra command line must be added as shown in Appendix A.

Running LOGINOUT with the input assigned to an interactive looping command procedure file which requests input from the user, then executes that input and loops back to request input again, seemed to be a safe bet. However, the DCL INQUIRE command looks toward the original SYS$COMMAND assignment for its input. Certain DCL commands (like the INQUIRE command) need this original command orientation to be the user's terminal in order to function. In a subprocess running LOGINOUT and inputting an interactive command procedure file to LOGINOUT, the process logical names are as shown in Table 2.

SYS$COMMAND (like SYS$INPUT) is originally assigned to the disk command procedure file. A reassignment of SYS$COMMAND within a command procedure has no effect. It was set originally when LOGINOUT was run and it is this original setting that determines where INQUIRE will prompt. The system does not bother to look and see where you have reassigned it in the meantime. We were trapped within the confines of the structured levels of DCL command procedures. There was one exception, although it involved an obscure use of DECNET.

```
$ ! This interactive looping command procedure file
$ ! called COMMAND.COM will only work if SYS$COMMAND
$ ! was originally assigned to the user's terminal.
$
$ LOOP: INQUIRE COMMAND "$"  ! inquires for DCL command
$                            !   from SYS$COMMAND
$       'COMMAND             ! substitutes & executes
$                            !   what you gave it
$       GOTO LOOP            ! returns to the INQUIRE
$                            !   for the next command
```

Figure 3

AN INTERACTIVE LOOPING COMMAND PROCEDURE

Table 2

PROCESS PERMANENT LOGICAL NAMES FOR A SUBPROCESS RUNNING LOGINOUT WITH A COMMAND PROCEDURE INPUT (COMMAND.COM)

Logical Name	Actual Device
SYS$INPUT	_DBA1:
SYS$OUTPUT	_DBA1:
TT	_DBA1:
SYS$DISK	_DBA1:[USER]COMMAND.COM
SYS$COMMAND	_DBA1:
SYS$ERROR	_DBA1:

Splitting Processes through DECNET

If you are connected to a network, VAX/VMS DECNET allows you to SET HOST to a different computer and operate there exactly as if you are on that system. (You must, of course, have an accessible account on that system to do this.) The new system prompts for a "Username" and "Password" and then welcomes you to that system. In the meantime, the originating process is kept "on ice" in a process called RTPAD, which passes the keyboard input out to the other system and then dutifully writes the output from that other system on your terminal screen.

The network software then may be used to log in again on your original system. Why not just log out and log in again? The answer is that your original process in RTPAD preserves part of your process context, including directory default, logical names, symbols, etc., and even your place in a command procedure. Using this technique you may, so to speak, teleport yourself and return to check out a command procedure from a different point of view.

The RTPAD image allows you to reenter the system in a second process, and as it does, it checks the logical assignment of SYS$COMMAND. Thus, the desired effect can be produced by creating a subprocess through DECNET with LOGINOUT input assigned to a command procedure containing a SYS$COMMAND assignment to the user's terminal and a DCL SET HOST command. You cannot gain command of the subprocess, but you can use it to run RTPAD and log back into your own system. You can even supply a username and password in the command procedure file. All of a sudden, you get two processes making alternate input requests, each with a "$" prompt.

The process of controlling this input prompt and the subsequent process output was achieved for VT100 terminals with yet another command procedure (somewhat like that in Appendix B) which caused one of the processes to become inactive while it waits for a cue from the other process. That cue was the existence of an empty file with a special name and type and version number which the new main process recognized. The existence of that file in a designated directory enables the process to prompt for input in the desired portion of the VT100 terminal screen using that terminal's scrolling feature. The command procedure printed an identifying header with relevant information at the top of its region, prompted for input, redefined the scrolling region, executed it, and finally checked for the existence of that enabling file and, if it existed, looped back for more input. Otherwise, it kept checking every second for that file to materialize. The procedure was generalizable to more than two processes. However, an error in the RTPAD software would cause a system failure under certain circumstances.

It was the interplay between the two command procedures that kept things going and the existence of the directory and file that

enabled the various processes to "communicate" with each other. The existence of the file with the proper name, extension, and version number enabled that process to prompt on the terminal. The first command procedure controlled the windowing and defined the DCL symbols referring to the second command procedure which did the process creating and interprocess controlling by way of file creation and deletion.

Splitting Processes in Other Ways

One of the drawbacks with the previous arrangement was the high overhead and demand on system resources. Each extra user-commandable process requires the creation of two more processes: the first to SET HOST to the second, a DECNET main process. In addition, DECNET itself tends to require a lot of system resources.

The CONTROL/Y key (CONTROL key and Y key pressed simultaneously) is commonly known functionally as an abort key. It acts as an interrupt of the process, and a return to command level prompt. The EDTCAI course disengaged the functionality of this key so that students could not accidently abort the course. This was a mistake. Some students found themselves helplessly lost in the maze of modules with no clear idea of how to return to monitor level. Standard across-the-board functionality, such as a universal HELP key colored red or a universal ABORT/EXIT procedure, is highly recommended. It is less well known that the user can return to that image with a "CONTINUE" DCL command, provided that no new command is executed that runs an image. There are a limited number of DCL commands that do not run an image, and these are listed in Table 3.

Table 3

DCL COMMANDS THAT DO NOT RUN AN IMAGE

ALLOCATE	ASSIGN	CLOSE
CONTINUE	DEALLOCATE	DEASSIGN
DEFINE	DELETE/SYMBOL	DEPOSIT
EXAMINE	GOTO	IF
INQUIRE	OPEN	READ
SET DEFAULT	SET PROTECTION/DEFAULT	SET VERIFY
SET UIC	SHOW DAYTIME	SHOW DEFAULT
SHOW QUOTA	SHOW PROTECTION	SHOW STATUS
SHOW SYMBOL	SHOWTIME	SHOW TRANSLATION
WAIT	WRITE	

VMS development promises to add another: a SPAWN command to enable the user to "clone" his or her processes context, so to speak, thus creating a subprocess in which another image or a series of images could be run without affecting the parent process. The user could return at any time to the parent and continue the process image there.

This would be very useful from our point of view. The current CBCD procedure is to:

(1) create a temporary file containing relevant status data before leaving the CAI course,

(2) exit the course via a LIB$DOCOMMAND system service call to a command procedure which runs the utility, and

(3) then run the CAI course again but starting the user approximately where he or she left off based on the status information in the temporary file.

This procedure works well for our new VMSCAI course and should also serve well in the revised EDTCAI course. However, if we can have our way, we would like both processes to exist, interpreting student input. The utility would show what actually happens and the CAI course would give the feedback. Although SPAWN would not actually be able to do this, a system service call to a LIB$SPAWN might work with the right programming interface.

Further Software Modifications

In some respects we have the SPAWN command now with the command procedure in Appendix A. It is actually not necessary to SET HOST in the subprocess. Subprocesses can be created directly with the command:

```
RUN SYS$SYSTEM:LOGINOUT/INPUT=TTA1:/OUTPUT=TTA1:
```

Such subprocesses are fully interactive. The process permanent logical names shown in Table 4 attest to that. The problem now lies only with the windowing. The interacting pair of command procedures were rewritten into one command procedure to take advantage of this new feature.

This revised command procedure is shown in Appendix B. The visible result is almost identical to that for the original pair. Each subprocess prompts in its window after checking to see if it is enabled to inquire. One difference is that the users are able to name the subprocesses themselves and that each subprocess then prompts with its name [2].

Another difference is that the main process acts only to switch among the subprocesses. CONTROL/Y is used to wake the main

process which then deletes and/or creates the files necessary for the subprocess to prompt. In this manner, as many subprocesses as is physically possible could be formatted on the VT100 terminal screen.

If we forego the windowing effect (which works only for VT100 terminals), we may settle for the command procedure in Appendix A which, instead of the existence of an enabling file, uses the DCL WAIT command. This command will put a process to sleep for the specified amount of time. The process can, however, be awakened with a CONTROL/Y. Furthermore, if that process is executing a command procedure, the user can specify a response to the CONTROL/Y.

If we make the subprocess prompt with its name, execute the inputted command, and then loop back and continue to prompt and execute as long as there is real input (it goes to sleep with just a <RETURN>), this subprocess can serve to perform tasks that the main process is unable to do if it is running an image.

Table 4

PROCESS PERMANENT LOGICAL NAMES FOR A SUBPROCESS WITH INPUT AND OUTPUT ASSIGNED DIRECTLY TO THE USER'S TERMINAL

Logical Name	Actual Device
SYS$INPUT	_TTA1:
SYS$OUTPUT	_TTA1:
TT	_TTA1:
SYS$DISK	DBA1:
SYS$COMMAND	_TTA1:
SYS$ERROR	_TTA1:

---

[2] Process names should be be unique within groups on VAX/VMS. Users on the system belong to a group with a group number identifying that group. The user identification control (UIC) is a number consisting of two parts: a group number and a member number. In the revised command procedure, the file used for communicating between the subprocesses has a name corresponding to the process name, and a version number corresponding to the user UIC member number. This helps solve the problem that previously resulted if more than one person used the software referring to the same files in the same directory.

For example, if users are annoyed by messages appearing on the screen while they are in the editor, they may summon the subprocess with a CONTROL/Y, have it SET TERMINAL/NOBROADCAST, continue when the main process prompts again, and then "<RETURN>" to sleep the subprocess. CONTROL/W refreshes the screen and the user is exactly where he or she had been before with no loss in process context.

Similarly, the user might wish to TALK to another user or reply to another user who wished to TALK to him or her. If an image is currently running, the user can:

(1) interrupt that image and wake the subprocess with a CONTROL/Y,

(2) sleep that main process with a WAIT 1 (wait one hour),

(3) TALK to the other user through the subprocess,

(4) type another CONTROL/Y when he or she is finished talking to wake up the main process, and then whichever prompts first,

(5) press <RETURN> to sleep the subprocess,

(6) type CONTINUE to return to the original image, and

(7) type CONTROL/W to refresh the original image.

Appendix A presents a command procedure for a subprocess which will do precisely this. This command procedure is independent of the type of terminal. The affect of this command procedure is related to the goal of SPAWN, except that certain parts of the process context are not cloned. Users can, however, create their own contexts in the subprocess by executing their LOGIN.COM files or using some other technique. They can keep the subprocess awake as long as it is getting input. As soon as it receives a <RETURN> for input, it goes back to sleep.

The situation is almost completely symmetrical. The main process can perform in the same manner. Either process will return to the "$" prompt with the DCL EXIT or STOP command. All sleeping processes and subprocesses will wake and prompt (in somewhat random order) wth a CONTROL/Y input. However, it is only the main process which can stop the subprocess (with STOP "subprocess name"). The difficulty of not being able to leave EDTCAI with a CONTROL/Y is solved if the user enters the course from a subprocess and wakes the sleeping main process to stop the subprocess.

Other uses involve switching between a foreground and background process (say an editor and DCL to compile link and execute a program) such that exiting and reentering the foreground process is accomplished using CONTROL/Y and WAIT. The image is saved so

that reentry is exactly at the exit point with all context preserved.

CONCLUSIONS

In some respects, we are much closer to our final goals than it may at first appear, because we can sequentially execute different images by switching between the utility and the CAI course about the utility. The technique of storing the relevant data from the CAI course in a temporary file before leaving the course has another dividend: while this procedure enables subsequent reentry into the course at the exit point, it could also be employed to place a user into a CAI course at a specific point based on the desired content. Thus, in a manner similar to on-line HELP, on-line LEARN might be made available to access the appropriate section of the CAI course for specific instruction. If this instruction was presented in a subprocess, the original process context would not be lost. These capabilities exist now. The central focus for growth now seems to lie with windowing.

Since the demonstration last fall, Brown University has been able to window on any type of terminal under their VAX/UNIX system. This includes the familiar VT100 terminal. The results of their work are published in a paper called "BRUWIN: An Adaptable Design Strategy for Window Manager/Virtual Terminal System" by Norman Meyrowitz and Margaret Moser. We visited Brown again this spring to see their recent developmental work. This work is in a general setting, but its consequences are clear: multiprocess windowed output is not only possible but also highly desirable for our CAI work.

As students make the transition from a controlled CAI environment to the actual utility, there is necessarily a qualitative discontinuity. While users may make this jump only once for each subject or utility they learn, it is nevertheless psychologically very significant. Using the "real thing" has infinitely more impact than the simulation does. We are primarily concerned with easing the problems encountered with this transition. As the distinction between on-line documentation and on-line training becomes more obscure, the gap that lies between CAI and the utility it is teaching should disappear, allowing interactive training to become an integral part of the user's entire on-line environment.

Appendix A

SUBPROCESS CONTROL

This is a command procedure for all terminals and all processes and subprocess created with the the symbol SBPC which is defined in this command procedure. It distinguishes the subprocess from the main process by making the subprocess (or main process) prompt with its name: "{subprocess name}:" verses "$".

This command procedure is an interactive loop for a subprocess created from the main process with the symbol SBPC. A subprocess running in this interactive loop can help run errands for you when you are otherwise occupied in executing an image, the editor for example. Normally the subprocess sleeps. It prompts with the subprocess name that you give it. You wake it with a CONTROL/Y and you sleep again it with just a <RETURN>. As long as you give it a series of commands it will execute images like TALK or MAIL that do not interfere with the main process image. In the mean time the main process will prompt on the terminal screen with a "$". Tell your main process to sleep with the DCL WAIT 1 command. This puts it main process to sleep while you work with the subprocess, and it doesn't destroy the main process image. When you are finished, a CONTROL/Y will wake the main process which will prompt with a "$". Use the DCL "CONTINUE" to get back your original image in the main process. When the subprocess prompts again with its subprocess name, just press <RETURN> and it returns to its waiting state.

You must first create the subprocess from the main process with the symbol SBPC. Normally you won't have this symbol defined. When you logon you have only your main process prompting with its "$". If you run this command procedure from your main process, it will prompt you with your main process name. This will also define the symbol SBPC so that you can use it to create a new subprocesses. When the main process prompts the first time with its name, type SBPC to create the subprocess. When it prompts a second time, then just press <RETURN>.

It will be out of your way while you work just with the subprocess, giving it the context that you wish. A new subprocess has only permanent symbols and logical names: $STATUS, $SEVERITY, SYS$INPUT, SYS$OUTPUT, SYS$COMMAND, SYS$ERROR, SYS$DISK, and TT, but not SYS$LOGIN. You must explicitly give it the context you want it to have. Run it through your LOGIN.COM or the LOGALL.COM in the SYS$MANAGER directory. Then run it through this command procedure. You will be prompted to name the subprocess. The name you pick must be unique within your UIC group. If it already exists, you will be prompted again. Then the subprocess prompts you with the name that you gave it. <RETURN> sleeps it. CTRL/Y wakes it and all other sleeping processes up, including the main process. You can return either process or subprocess to

its normal "$" prompt the DCL EXIT or STOP command. You can stop a subprocess from within with the DCL LOGOUT command in the subprocess or from above (the creator process) with STOP "subprocess" (the process name). (Watch the case of the letters in subprocess name it makes a difference.)

```
$ver$:='f$ver(0)
$START:
$ SBPC :== run sys$system:loginout /input='f$log("TT")' -
    /output='f$log("TT")' ! the subprocess will have no name
$ if "''f$proc()'".nes."" then goto LOOP       ! now name it
$ RETRY: on error then goto RETRY     ! try again to name it
$ write sys$output "Please name your subprocess"
$ assign /user sys$command sys$input  ! so image can prompt
$ run sys$system:setname              ! changes process name
$
$ LOOP: p8=p8+1                                ! Loop deletes
$       delete /symbol p'p8                    ! symbols p1-p8
$       if "''p8'".nes."" then goto LOOP       ! Check til done
$       on contY then goto $                   ! Wakes you up
$ ! The last line will make the (sub)process prompt with its
$ ! name but will also unfortunately destroy a process image
$
$ $: inquire $ "''f$proc()'"     ! prompts with process name
$ if $ .eqs. "" then wait 21     ! sleeps process if no input
$ on severe_error then goto $    ! keeps it prompting
$
$ if "''f$logical("SYS$INPUT")'".nes."''f$logical("TT")'" -
    then ass/user 'f$logical("TT") sys$input -
$                                ! for interactive image
$     '$                         ! 'f$ver(ver$)
$     ver$ := 'f$ver(0)          ! shuts verify off
$ goto $                         ! the interactive loop
```

Appendix B

SUBPROCESS WINDOWING

Modify the directory specification "[]" for MPC in the second line after START: so that this command procedure file resides there and you can write in that directory. (If you name this file ".COM" then "@MPC" will run it.)

1. Run the main process through this command procedure.

2. Use the command "SUBP" to create a subprocess.

3. Run the subprocess through this command procedure, but specify the window size and location as follows:

a. Use CONTROL/Y to get the attention of the main process, e.g., "@MPC: 2 3" for the second third of the screen.

b. Use "GET", "LET", or "STOP" with the subprocess name to:

- GET to that subprocess and suppress the others,
- LET that subprocess prompt and not suppress the others, or
- STOP that subprocess (or use "LOGOUT" in the subprocess).

You may also use DCL commands in the main or subprocess. X or EXIT will return any subprocess to the "$" prompt. CLEAR will clear the screen.

```
$ver$:='f$ver(0)  ! MPC:.COM for VT100 subprocesses
$START:           ! SYMBOLS AND LOGICALS FOR SUBPROCESS CONTROL
$ X := "exit !'f$ver(ver$)"
$ if "''f$log("MPC")'" .nes. "[]" -
    then assign [] MPC
$ if p1 .eq. "" then goto MAIN       ! for the main process
$ if "''f$proc()'" .nes. "" then goto NEXT !already has name
$
$ RENAME: on error then goto RENAME  ! you must rename it
$ write sys$output "Pick any name for your subprocess."
$ assign/user sys$command sys$input  ! so image can prompt
$ run sys$system:setname             ! changes process name
$
$ NEXT:  clr:="<ESC>[H<ESC>[J<ESC>[1;24r" 'f$ver(0)
$      CLEAR:="write sys$output clr" ! will clear the screen
$
$!  WINDOW FORMATS FOR MULTIPLE SUBPROCESSES   'f$ver(ver$)
$ header := -
   "'f$pro()' = 'f$use()' ""in"" 'f$dir()' ""at"" 'f$tim()'"
$ if "''p2'" .eqs. "" then p2 = 1       ! want whole screen
$ bot=24*p1/p2                          ! bottom of window
$ rng=24/p2                             ! size of window
$ top=bot-rng+2                         ! top of window
$ hdr=top-1                             ! header location
$!  top='top  bot='bot  rng='rng  hdr='hdr -THIS sums is up
$
$! USER UIC INDIVIDUALIZES THE ENABLING FILES FOR THE USER
$ uic := 'f$user()                      ! get the user's
$ comma = 'f$locate(",",uic)+1          ! UIC and extract
$ bracket = 'f$locate("]",uic)-comma    ! the user number.
$ uic := 'f$ext(comma,bracket,uic)      ! (the second part)
$
```

```
$ LOOP: p8=p8+1                          ! cleans up the
$       delete /symbol p'p8              ! symbols p1-p8
$       if "''p8'".nes."" then goto LOOP!
$
$       v := ""
$! CREATES ENABLING FILE AND HEADER INFORMATION
$DCL:create/protection=(w:rwed) MPC:'f$process().'uic';1
$    currentheader := 'header            !'f$ver(0)
$    wr sys$output "<ESC>[''top';''bot'r<ESC>['''hdr';1f"
$    wr sys$output "<ESC>[0;K"
$    wr sys$output "<ESC>[1;7m<ESC>#3 Welcome to ", -
                   "subprocess ''f$proc()' <ESC>[0;m"
$    wr sys$output "<ESC>[1;7m<ESC>#4 Welcome to ", -
                   "subprocess ''f$proc()' <ESC>[0;m"
$    wr sys$output "<ESC>[0;K"
$
$! level-1 DCL $ LOOP for output  window control 'f$ver(0)
$
$ $: currentheader := 'header        !'f$ver(ver$)
$    on control_y then goto $
$    if "''f$logical("SYS$INPUT")'".nes."''f$logica("TT")'"-
       then ass 'f$logical("TT") sys$input !'f$ver(0)
$    wr sys$output "<ESC>['''hdr';1f<ESC>[1;7m", -
       "<ESC>[2K ''currentheader'<ESC>[0;m", -
       "<ESC>['''top';''bot'r<ESC>['''bot';1f<ESC>[A
$       inquire  $ "''f$process()'"
$       if $ .eqs. "" then goto $
$       on error then goto $
$       '$                          !'f$ver(temp_verify_state)
$       temp_verify_state := 'f$ver(0)
$       wr sys$output ""
$
$CHECK: open /read /error=wait file MPC:'f$process().'uic';1
$       close file                  ! the enabling file exists
$       goto $                      ! so goto the process prompt
$
$WAIT:  wait 0:00:01                ! the enabling file does not
$       goto CHECK                  ! exist so check again ...

****************************************************************
****************************************************************

$! The following part is for the main process control only
$
$ MAIN: on control_y then goto MAIN
$ H*ELP:= goto HELP
$ SUBP := RUN SYS$SYSTEM:LOGINOUT /INPUT='F$LOG("TT")' -
       /OUTPUT='F$LOG("TT")'        !
$       show process /sub           ! lists the subprocesses
$       directory MPC:*.'uic'       ! shows which can prompt
$ INQM: inquire $ "What do you want to do? (HELP available)"
$       if "''f$extract(0,3,$)'" .eqs. "GET" then goto GET
$       if "''f$extract(0,3,$)'" .eqs. "LET" then goto LET
```

```
$       '$
$       wait 21
$ GET:  delete MPC:*.'uic';*
$ LET:  create/protection=(w:rwed) -
          MPC:'f$extract(3,9,q).'uic'
$       goto CHECK
```