WANG LABORATORIES, INC.

MVP PROGRAMMING GUIDE

May 31, 1978

The following document is intended to give the reader an overall view of the MVP Operating System with emphasis on programming considerations for the multi-user system. It assumes the reader is familiar with Wang BASIC. Compatibility with other 2200 systems (2200T and 2200VP) is described as well as techniques for converting existing software for the 2200MVP. The program and data sharing features, unique to the multi-programming environment are introduced.

ONE INDUSTRIAL AVENUE, LOWELL, MASSACHUSETTS 01851 • TEL. (617) 851-4111 • TWX 710-343-6769 • TELEX 94-7421

I. Philosophy of the MVP

Hopefully, programmers and users of past Wang 2200 systems will find the MVP to be a logical variation on the familiar 2200 theme. The MVP hardware is surprisingly similar to the 2200VP. The BASIC-2 language supported on the MVP is essentially the same as the language of 2200VP, with extentions. Users will find the machine almost as responsive as the VP, and every bit as easy to operate. Programmers will find a high degree of software compatibility with both the 2200VP and 2200T.

In addition to the microcode necessary to implement the BASIC-2 language, the 2200 MVP's control memory contains a multi-programming operating system. The primary goal of a multi-programming operating system is to allow several users to share a single computer efficiently. To accomplish this, the operating system divides the resources of the computer - memory, peripherals, and CPU time - among the users. Once each user has been allocated a share of the computer resources, the operating system acts as the traffic policeman, allowing each user to use the system in turn while preventing users from interfering with each other's computations.

On the Wang 2200MVP, the user memory is divided into fixed size memory partitions by executing the Wang supplied utility program, @GENPART, immediately following system power up (see MVP Intro Manual). Each memory partition behaves much like a single user 2200VP. From the user's point of view, each partition functions independently from the other partitions in the system. Each user may LOAD and RUN BASIC software, compose a program, or perform immediate mode operations. As in a single user environment, the user has complete control over his partition. No other operator or other partition may halt execution or change the program text of his partition.

Each terminal may control several partitions executing independent jobs. At any given time, only one of these partitions is in control of the screen and thus capable of interacting with the operator. The partition in control of the screen is said to be attached to the terminal or running in the foreground. Other partitions assigned to the terminal may continue to execute in the background, until such time as operator intervention becomes necessary. If a background job attempts to print to the CRT or get input from the keyboard, its execution is suspended until the terminal becomes available to it.

The terminal becomes available to the waiting background job when the foreground partition explicitly gives up control of the terminal. Sharing the terminal in this way means that a partition maintains control of the terminal for as long as it desires. Messages from other partitions cannot appear and mess up the CRT display at undesirable times. Foreground/background processing is discussed further in Section VI.

In addition to partitions operating independently, the MVP allows partitions to co-operate. Co-operating partitions may share program text (global subroutines) and/or data (global variables). These features allow considerable memory savings over a situation where each partition has its own copy of the same code or data table. The integrity and independence of a partition is maintained by requiring the partition to explicitly declare itself to be global (sharable). (Sharing program text will be discussed in detail in sections IV and V).

The analogy of completely independent single user machines is clouded somewhat by contention for shared peripheral devices. The situation is familiar to programmers used to working with Wang 2200 systems that share one or more disk drives via disk multiplexers. It is sometimes necessary to request exclusive control of a disk while an update is made. Likewise on the MVP it is necessary for a partition to exclusively control a printer for the duration of the printing of a report, lest one partition's print lines become unreadably intermixed with another's. The concept of disk hog mode has been extended on the MVP. The $OPEN and $CLOSE statements allow a partition to request exclusive control of any device on the system.

The programmer who wishes to take the macroscopic view of the MVP system as a whole is quite correct in thinking of all partitions as executing simultaneously. It doesn't take long to realize, however, that the 2200MVP has only one CPU. The operating system creates the illusion of concurrent execution of several programs by rapidly switching from one to the other in turn.

What follows is a simplified description of the MVP operating system, and as such is neither precise nor complete. The programmer need not generally be concerned with these details of how the operating system does its job, but this presentation may be helpful in giving users an overall feel for how a multi-programmed system attempts to maximize system utilization while maintaining good user response time. The programmer who is aware of some of the operating system duties can actually help the system to perform better with little inconvenience to his own coding techniques.

MVP partitions are serviced in a round robin fashion, with some additional priorities given for certain I/O operations. Each partition in turn is given a 30 ms. timeslice, during which it has exclusive control of the CPU. The CPU has within it a 30 ms. timer which is set at the beginning of the timeslice. At the completion of each BASIC statement and at various points in the middle of long MATRIX and I/O operations, the clock is checked to see whether the 30 ms. timeslice has been exhausted.

When the timeslice is over, the MVP operating system carefully saves the status of the partition so that it may be restored later when that partition's turn comes around again. The status of the next partition in line is then loaded and its 30 ms. timeslice begins. The process of swapping out a partition at the end of its timeslice is called a breakpoint. The programmer cannot predict in advance when a breakpoint will occur. Except for a few cases involving global variables within matrix and I/O statements, the occurrence of breakpoints is of no concern of the programmer.

Timeslices do not always last exactly 30 ms. Unlike many operating systems, the MVP switches users (breakpoints) when it is convenient, rather than strictly by the clock. This reduces the amount of status information that must be saved, giving the MVP comparatively low "operating system overhead".

More importantly, breakpoints may occur in the middle of BASIC I/O statements. If, for instance, the disk is hogged by another partition, this condition is quickly detected and a breakpoint occurs. I/O breakpoints differ from program breakpoints in that the partition is specifically marked as "waiting for I/O". When the partition's turn comes around again, it takes only a few microseconds to decide whether processing may proceed or whether the partition is still waiting for the I/O device and thus may be bypassed. Thus if a printer runs out of paper or a partition that does not currently control the CRT attempts CRT output, processing is suspended in that partition almost as effectively as if it were removed entirely from the system, until the I/O device becomes available.

The CPU is much faster than any of its peripherals. Breakpointing during I/O operations allows the MVP to keep many I/O devices busy concurrently with program processing. To accomplish this I/O overlap requires buffering and quite often microprocessors to control the peripherals. The most sophisticated of these intelligent peripherals is the 2236MXD terminal controller. The MVP CPU does not perform INPUT or LINPUT statements. Instead it asks the microprocessor in the 2236MXD to perform the operation. Just as in the case of the printer out of paper, a partition executing an INPUT or LINPUT statement is marked as waiting for I/O and receives no CPU timeslices until the INPUT or LINPUT statement is terminated with carriage return or a special function key. The MXD also performs the line editing functions to move the cursor, and insert and delete characters.

Lastly, the MVP operating system performs some address translation. After all, every partition refers to the terminal keyboard as address /001, the CRT as address /005, and the terminal printer as address /204. The operating system makes sure that all output gets to the proper terminal and all input comes from the proper keyboard.

II. How Independent Programs can Share the CPU and Peripherals

When configured as < n > partitions and < n > terminals, the 2200MVP can be treated as if it were < n > separate 2200VP computers. The 2200MVP is generally compatible with the 2200VP BASIC-2 language. Users with existing software are encouraged to try their software on the MVP as is. The few compatibility problems that may be discovered are discussed in Section III.

As a first approximation for the required MVP partition size, use the size of the single user system the software is designed for. This may be a few hundred bytes small for programs designed for the 2200T and slightly large for 2200VP programs. Once the program is loaded and resolved, the immediate mode command PRINT SPACEK-SPACE/1024 will reveal the exact partition size necessary in K bytes.

Once the partition size necessary for each application is determined, the user will become interested in the total amount of MVP memory necessary to run several programs concurrently. The total memory requirement for an MVP system is the sum of the partition sizes plus 3K for operating system tables. The amount of user memory taken by the system on the various 2200 systems is compared below.

User Memory Taken for System Use

2200T	2200VP	2200MVP
700 bytes	3K	3K + 1K/partition

Notice that the total MVP memory overhead with 2 or more partitions is less than the memory overhead if separate VP's were used.

Note: The discussion above about determining partition size takes into account the need to reserve 1K for partition overhead.

MVP programming and operating considerations are identical with those of a system of T, VP, and workstation processors multiplexed to a disk, with the added complication that the printer can be accessed directly by any user at any time, without pushing a button on a manual multiplexer.

An operator wishing to use the printer on a manually multiplexed system must (1) determine if anyone else is about to print, (2) press the button on the multiplexer for his own station, and (3) run his program. If he runs his program when the printer is being used by someone else, his program just hangs up. In contrast, an operator wishing to use the printer on a 2200MVP must (1) determine if anyone else is about to print, and (2) run his program. However, if he runs his program when the printer is being used by someone else, the print lines from both jobs will be intermixed on the paper.

A very simple modification to the programs will eliminate the possibility of intermixing lines. The MVP allows an executing program to "hog", or reserve, a peripheral device for as long as it wants. The program must be modified to execute a "$OPEN /215" statement before using the printer, and to execute a "$CLOSE /215" statement when finished with it. A program which prints several reports might execute a "$OPEN /215" before and a "$CLOSE /215" after each separate report. This would guarantee that an entire report would be contiguous on the paper, but would possibly let some other program's output appear between the several reports.

Of course, the above discussion also applies to plotters, typewriters, or any other peripheral device which is plugged into the MVP CPU box. Note that it is not NECESSARY to include the "$OPEN" and "$CLOSE" statements in MVP programs -- this is merely a mechanism to guarantee that a program has exclusive use of a peripheral, and to allow operators to run various jobs without the possibility of spoiling someone else's output.

Another simple method to guarantee that a job has exclusive use of a printer, plotter, typewriter, etc. is to plug the device directly into the output connector on the assigned terminal. The disadvantages of this method are, (1) that it must either be considered permanent or else involve a lot of plugging and unplugging, and (2) references to the device must be modified to address /X04 (e.g., /204, /404) since this is the fixed address of the terminal output device connector. Use of a 2221M printer multiplexer on the MVP should not be completely discounted, for it is a good way for a cluster of terminals, located at some distance from the CPU, to share a common printer.

It is possible to reserve a peripheral permanently at partition generation time. This is often done with TC boards, since it is rarely logical for two programs to share a TC board.

III. Compatibility with earlier Wang 2200 Systems

This section deals with the few software imcompatibilities that exist in the 2200MVP. They are listed in the order of frequency with which they are usually encountered. While it is desirable to get existing software up and running quickly, the MVP offers several means of improving performance and reducing memory requirements. Quite often it is possible to reduce the memory requirements of 2200T programs by recoding them using the more powerful BASIC-2 statements. The greatest memory savings are to be realized when program text is shared as discussed in Sections IV and V.

Programs may choose to ignore the potential existence of other programs in the system, but a small amount of "good citizenship" can go a long way in improving overall system performance. Use of the INPUT and LINPUT statements better utilizes the 2236MXD and thus frees a lot of CPU time for other partitions to use in comparison with doing input with the KEYIN statement. If a program must use KEYIN, it is better to use the "VP form" of the statement: KEYIN A$ rather than KEYIN A$, 10, 20.

If a program must test a condition over and over in a tight polling loop, it should perform the test once and then give up the remainer of its timeslice with the $BREAK statement. For programmer convenience, this feature has been built into the polling form of KEYIN. The KEYIN statement is treated as if it actually were KEYIN A$, 10, 20 : $BREAK.

Time delays should be created using SELECT P rather than with FOR/NEXT loops. SELECT P delays are timed by the 2236D terminal, while FOR/NEXT delay loops waste CPU time that could be more productively used by other partitions.

Features of earlier 2200 systems not supported.

1) Users converting 2200T programs for use on the MVP should refer to Appendix C of the BASIC-2 Language Reference Manual for a list of differences between Wang BASIC and BASIC-2.

2) The MVP does not permit $GIO to a disk. This will affect programs that send a CBS strobe to the disk to activate hog mode. The MVP uses the $OPEN and $CLOSE statements to activate disk hog mode, as well as to request exclusive control of any other peripheral. The address form of disk hog, SELECT DISK 390, is supported on the MVP.

3) Some programs use KEYIN to input atom codes from the text atom keys. This works properly on the MVP, but the 2236D terminal does not have all of the atom keys found on a 2226 console. The absence of a PRINT key seems to create the most software transportability problems.

4) Many programs test for the existence of printers and disks before attempting to access them. This is a problem on the MVP because $GIO is not permitted to disks and because several lines worth of buffering separate the terminal printer from the I/O bus. A partial solution is to use the $OPEN and ON ERROR GOTO statements to see if the device address was declared in the master device table when the system was configured using the @GENPART program.

5) Some programs use $GIO with timeout to the keyboard to insist that an operator respond in a fixed period of time. $GIO with timeout is not supported at address /001 on the MVP.

6) Character insert mode is not supported on the MVP. Characters must be inserted into text lines using the insert edit key.

7) The MVP does not permit CI, CO, or INPUT to be selected to any device other than the 2236D terminal. The only exception is that the output of TRACE may be selected to a printer with SELECT CO. The width of the Console Output device may not be redefined to be other than 80 bytes for the purpose of INPUT or LINPUT.

MVP Differences

1) The most obvious difference is that CRT output is somewhat slower on the MVP, depending on the data rate set for the serial line connecting the 2236D terminal to the 2236MXD. This may alter programming strategies that frequently update the entire screen.

2) The 2236MXD allows a maximum of 480 bytes to be entered into a single line request. This places some restriction on the maximum length field that may be entered with a single INPUT or LINPUT statement. This restriction also limits the length of a multiple statement program line that may be entered or edited on the MVP.

3) There are several differences in the LINPUT statement.

   a) The most obvious difference is that the cursor blinks to indicate edit mode. The 2200VP displays an asterisk to the left of the field being edited.

   b) The second difference is that the non-edit mode form of LINPUT places the cursor at the first position of the field instead of following the last non-blank prefill character as on the 2200VP.

   c) The maximum length field is limited to 480 bytes.

4) When an MVP is master initialized, the default disk (SELECT DISK or #0) is the platter the system microcode is loaded from. This differs from the 2200T and 2200VP, which always set the default disk to /310 when master initialized.

5) Time delays using FOR/NEXT loops or $GIO (75xx) not only waste CPU time, but vary in duration as a function of CPU loading. Every 30 ms. a program breakpoint occurs. When the partition's turn comes around again, it is allowed to waste another 30 ms. Time delay loops thus become minimum delay loops on the MVP. It is recommended that SELECT P be used for delays.

SELECT P delays are implemented by the 2236D terminal. Special characters are sent to the terminal to cause the terminal to delay. Since the terminal is buffered, the program does not wait at the PRINT statement causing the delay. It is not unusual for a program to be several PRINT statements ahead of the CRT display while using SELECT P.

6) The 2236 MXD buffers up to 36 keystrokes that have not yet been requested by a program. This helps smooth out peaks in operator typing speed in data entry applications, but it also allows the operator to anticipate program prompts. Sometimes it is necessary to flush this keystroke buffer, for instance, to minimize the effect of an operator beating on the return key while a program overlay loads from floppy disk. It is easy to flush the keyboard buffer with:

```
10 KEYIN A$, 10, 10
```

IV. How to Share Program Text (with No Overlays)

Consider the situation where < n > operators are to sit at < n > terminals attached to < n > partitions, and all run the same job. If the partitions are treated as separate computers, then each partition will have a complete copy of the program, including text and variables. A considerable savings in memory could be realized if only one copy of the program was kept in the MVP, and all the users could share it. This can be done. The text of the program, which does not change as the program runs, can be shared; while the variables, which must change as the program runs, cannot be shared (each user must still have his own copy of all the variables).

In order to share program text, it must reside in a partition which has declared itself to be "global", that is, accessable to other partitions. The simplest way to do this is to create a separate partition, that will not interact directly with an operator, to contain the text of the program (but not the variables), and to let each operator's partition contain only (1) a complete set of variables for the program and (2) a call upon the global partition.

Any partition in an MVP may declare itself to be global and give itself a name by executing a "DEFFN @PART name " statement. Any partition may access a marked subroutine ("DEFFN ' number ") in a global partition by first executing a "SELECT @PART name " statement, and then executing a standard call to the marked subroutine ("GOSUB ' number "). If the name s match and the marked subroutine does not exist in the calling partition, the correct marked subroutine in the global partition will be entered. (See the 2200VP BASIC-2 manual, chapter 16 "The 2200MVP", for a complete description of global and calling partitions). Arguments may be passed to global subroutines in the same manner as if the marked subroutines resided in the calling partition.

Since the calling partition is resolved separately from the global partition, it is the responsibility of the calling partition to contain the necessary DIM or COM statements to define all variables that will be referenced during the execution of the global text. Failure to do so will result in execution time errors.

In order to change a VP program into a shared MVP program, the program must first be modified, and then be properly loaded and run. The following instructions pertain to the modifications.

1. Only marked subroutines can be shared. Change all desired entry points to global text to marked (DEFFN') subroutines. Quite often it is possible to share an entire program, in which case a DEFFN' statement is added to provide a marked subroutine entry to the main line routine.

2. Separate the program into two program files: one, which we will call MAIN, containing all the existing DIM and COM statements and any non-sharable code; and the other, which we will call SUBS, containing all the marked subroutines to be shared.

3. Create a list of all the variables used by all the marked subs. A LIST V operation on the file SUBS is the most convenient method here.

4. Ensure that no DIM or COM statements remain in the file SUBS, otherwise unnecessary storage will be allocated in the global partition.

5. Add to the beginning of the file SUBS the following code:

```
10 DEFFN @PART "jobsubs"
20 $RELEASE TERMINAL
30 PRINT "Partition"; #PART; "background job 'jobsubs'"
40 GOTO 20
```

Line 10 makes the text in the partition accessable to the other partitions and gives it the name "jobsubs".

Line 20 makes it convenient to resolve this partition under operator control, if that becomes necessary. Global partitions are most conveniently loaded by the "automatic program load" feature of system configuration.

Line 30 provides some information if a terminal ever gets attached to this partition, which normally should not happen. Execution is suspended when line 30 is executed until such time as the terminal becomes attached.

Line 40 is needed because it is not logical to execute the global text directly, because the global text is all subroutines and because no variable storage is allocated within the global partition itself. After the branch back to line 20, which releases the terminal, execution again becomes suspended in line 30.

6. Using the existing DIM and COM statements in the file MAIN and the list of variables obtained in step 3, add to the file MAIN a DIM statement containing all the variables used by the text in SUBS which are not already mentioned in a DIM or COM. The purpose of this is to ensure that all of the necessary variables are allocated in the calling partition; any combination of DIM and COM statements which accomplishes this is sufficient.

7. Add to the beginning of the file MAIN the following code:

```
5 SELECT @PART "jobsubs" : ERROR $BREAK : GOTO 5
```

The execution of this statement will allow all marked subroutine calls in this partition to refer to marked subroutines in the global partition named "jobsubs".

The process of global partition definition and reference is accomplished by the execution (not the resolution) of "DEFFN @PART" and "SELECT @PART" statements. If line 5 in this calling partition is executed before line 10 in the global partition has been executed, the system will not know that we intend to have a global partition "jobsubs", and a run-time error will result. This is likely to happen even if MAIN and SUBS are both started using the "automatic program load" feature of system configuration. In order to cope with this possibility, the part of line 5, above, that follows the ":ERROR" will be executed if the global partition is not yet ready, causing the calling partition to "wait a while, then try again".

Once the program has been modified for sharing, it must be properly loaded and run. The most convenient method to do this is to construct a configuration to run the programs, and to use the automatic program load feature to start them up. The following instructions pertain to determining the size of each partition and setting up a memory configuration:

1. Working in a comfortably large program-development partition, load the file SUBS, but do not RUN it. (If you RUN it, you will be attached to another partition by the "$RELEASE TEMINAL" statement. However, nothing else dangerous will happen, and the program will be resolved when you get back to it). Next, execute in immediate mode "PRINT SPACEK-SPACE/1024". The number printed will be the required size of the global partition.

2. Next, LOAD and RUN file MAIN. MAIN must be RUN in order to allocate memory for the variables. (RUNning MAIN should result in an infinitely long wait at statement 5, since "jobsubs" is not defined if a CLEAR was executed after step 1. Use the HALT key). Determine the memory requirements of MAIN with the same immediate mode PRINT command used in step 1.

3. Power down and up again. When the system generation program, @GENPART, is running, construct a configuration with a partition for each terminal, sized with the number from step 2 above, with "program to load" set to "MAIN". Add one partition (any terminal), sized with the number from step 1 above, with "program to load" set to "SUBS".

The following is an example of what the @GENPART program should display when the configuration has been completely specified, assuming (1) the required size of the global partition is 28 KB, (2) the required size of the calling partition is 10 KB, and (3) three terminals are to run the same job:

PARTITION	SIZE(K)	TERMINAL	PROGRAMMABLE	PROGRAM
1	10.00	1	Y	MAIN
2	10.00	2	Y	MAIN
3	10.00	3	Y	MAIN
4	28.00	1	Y	SUBS

Partition #4 has been assigned to terminal #1. This assignment will not be used, but each partition must be assigned to exactly one terminal.

Programming has been enabled for all partitions in this example, but you may want to disable programming if the operators are to be forced to run only the one program, in MAIN.

4. This configuration may be stored on disk. Whenever the system is loaded using this configuration, the calling and global programs will automatically be loaded and started up, and the operators need only to sit down and begin to operate.

V. How to Share Program Text When Overlays are Involved

Many software packages use program overlays to some extent. Since it is generally not appropriate to overlay in global areas this means that it is not as simple a matter to share program text as the last section implies. Several possible strategies for sharing portions of applications that involve overlays are discussed in this section.

First, look for subroutines that are used in several overlays. Disk access methods, such as KFAM, and screen formatting subroutines come to mind. It is easy to apply the methods of the last section to extract a set of subroutines and put them in a global partition. The two basic rules of text sharing still apply: 1) Only marked subroutines can be shared, 2) The calling partition must define all variables that will be encountered during execution of global text.

Many packages are structured with several frequently used overlays and a number of infrequently used overlays. For example, in a data entry package such as Easyform, a number of overlays support forms generation, listing, etc. while one or two support real time data entry activities. It may be efficient to divide such a package into:

(a) The frequently used real time overlays which might be restructured into global programs and made available to a number of operations simultaneously with small calling partitions large enough only to contain the variables.

(b) The infrequently used overlays which might be retained as a single user job, possibly requiring quite a bit more memory than the above, with the intention that only 1 operator will perform these functions.

Some jobs have a number of sequential overlays with little common text, (similar to job steps in a batch run and/or operator prompt and initialization jobs executed prior to a main job). Often these type of jobs cannot benefit significantly from global text storage.

Some things to consider when sharing text from a program which uses overlays:

The LOAD statement clears (resets) the global pointer. That is, when a program, which has established a link with a global partition, loads an overlay, the link with the global partition is broken and must be reestablished. Usually the simplest way to do this is to have the overlay contain a SELECT @PART statement near its beginning. Note, however, that the MVP allows transfer to a specific line number after executing a LOAD, which could be a line in the resident part of the program containing the SELECT @PART and a GOTO to the beginning of the overlay.

If a LOAD statement is encountered while executing global text, the overlay is performed in the calling partition. The usual rules of program overlays apply: 1) The subroutine stack is flushed and 2) execution of the overlay begins in the calling partition. In effect; the system pretends the LOAD statement was encountered within the text of the calling partition.

The steps pertaining to modification of a program using overlays are almost identical to those described for non-overlay programs. The only differences are:

1. The variables used in the extracted global subroutines are inserted via COM or DIM statements in each valid overlay configuration. This is in each overlay, if the entire program is overlayed each time, or it may be once, if inserted in a main section which is never overlayed.

2. The variables should keep the same context as in the original program. That is, common variables if not to be initialized at each overlay, non-common if to be reinitialized at each overlay.

3. It is possible that not all the overlays will reference every marked subroutine in the shared text, and therefore not all the variables mentioned in SUBS will be referenced. Those not referenced need not be defined in the calling partition. The loading of an overlay provides an opportunity to redefine the variables in the calling partition.

VI. How Programs Can Share an Operator (Foreground/Background Processing)

Many CPU bound or disk bound jobs run for a relatively long time without operator intervention (relatively long, meaning long enough for the operator to do something useful elsewhere). In such a case, it would be beneficial if the operator could be running another, perhaps interactive, job in the meantime. Of course, any operator can do this, merely by getting up and sitting down at another terminal.

On the 2200 MVP, it is possible for one terminal to control more than one partition, and therefore more than one job. The $RELEASE TERMINAL statement causes the terminal to be detached from the current partition and attached to another partition (to which it has been assigned at system configuration time).

A single operator can control any number of "background" jobs, plus one "foreground" job. A background job is defined as one which can be expected to run to completion, or at least for a usefully long time, before requiring any attention from the operator. In the commercial environment, there are many programs, usually with the term "update" as part of their title, which operate on a data file or files according to what they find in a command or transaction file. Such programs are ideal candidates for operation as background jobs. On the other hand, there are many programs, often with the term "entry" in their title, which are designed to accept data from an operator (this is where the command or transaction files come from). These obviously must be run as foreground jobs.

Using the most direct method of running two jobs, the operator merely starts a job in one partition, and when it is going well and needs no more intervention from him, he detaches from that partition in order to go start a job in another partition. Unfortunately, if the detached job gets into trouble, there is no terminal available for it to use to call for help. Furthermore, many programs periodically report their current status via the CRT in order to reassure the operator that the job is still alive and running. If a terminal is not available, the job will be suspended at its first status report. A few simple software modifications make the running of multiple jobs much safer and more productive.

The background program should be modified so that after the initial dialog, the program releases the terminal to the foreground partition. Thereafter, whenever the program has something to report which does not require a response from the operator, it must test to determine if the terminal is attached to the partition with $IF ON /005. If it is, the report may be made; if it is not, the report may be discarded. When the program requires a response from the operator, it can simply communicate to the terminal without testing, and the operating system will automatically suspend the partition until the terminal becomes attached. In addition, the background job should provide a means for allowing the terminal to be released back to the foreground job when the operator has read the status report and is satisfied.

The foreground job should be modified so that it periodically releases the terminal to the background job, or at least provides the operator with an option of checking on the background job. Since the method of checking the background job involves releasing control of the CRT, checking the background job must be done when the foreground job is willing to allow the screen display to be cleared or at least is in a position to reconstruct the display when it regains control of the screen.

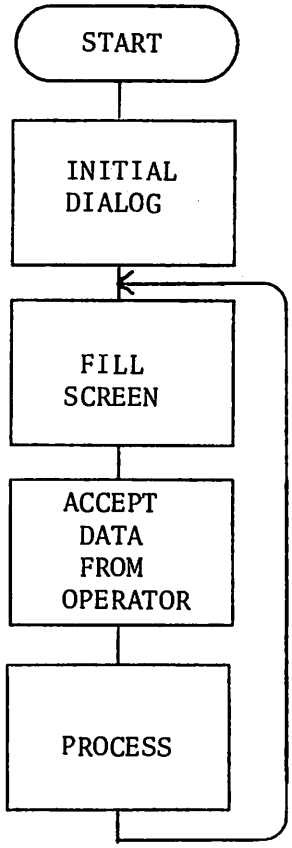
The distinction between two of the forms of the $RELEASE TERMINAL statement is important here. $RELEASE TERMINAL polls the background job for distress messages or operator requests. The background job will respond only if its execution has become suspended due to an attempt to print to a CRT it did not control. $RELEASE TERMINAL TO partition number or partition name attaches the terminal to the background job whether it has asked for it or not. When the background job tests to see if it controls the terminal, it will find that it does and thus may proceed to print its status information.

The method of testing the status of a background job suggested in this section requires the background job to be constantly testing to see if it controls the CRT. Another method of producing background job status reports using global variables is discussed in the next section.

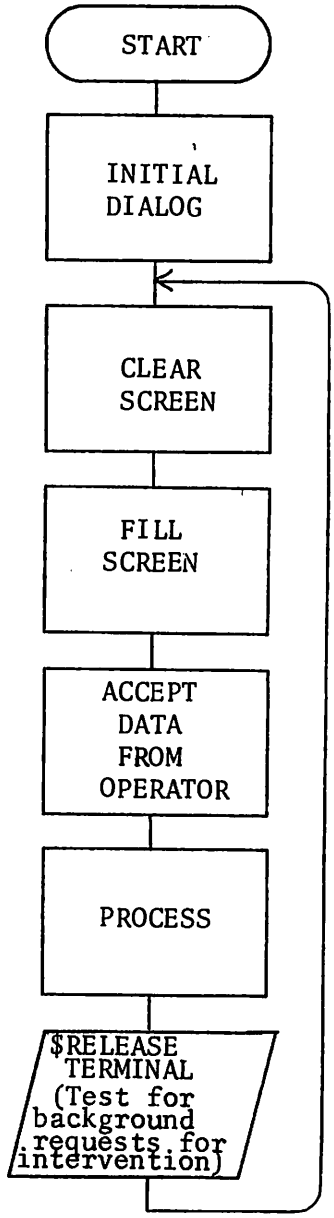
The following examples assume that an analyst has selected a suitable program to be run as a background job, and that the current structure of that program is: first, it conducts an initial dialog with the operator (to find out what files to use, etc.), then begins to execute a loop, which contains some code that reports the results of each trip through the loop, using the CRT. It is further assumed that the analyst has selected a program to be run as a foreground job, and that this program executes a loop containing a section (probably beginning the loop) which updates the entire CRT screen. Your programs may not have this exact structure, but the elementary principles in this example can be adapted or extended to fit many circumstances.

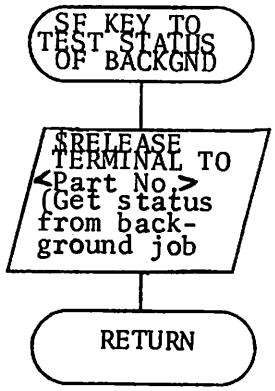
MODIFICATION OF A PROGRAM FOR FOREGROUND OPERATIONS

ORIGINAL PROGRAM



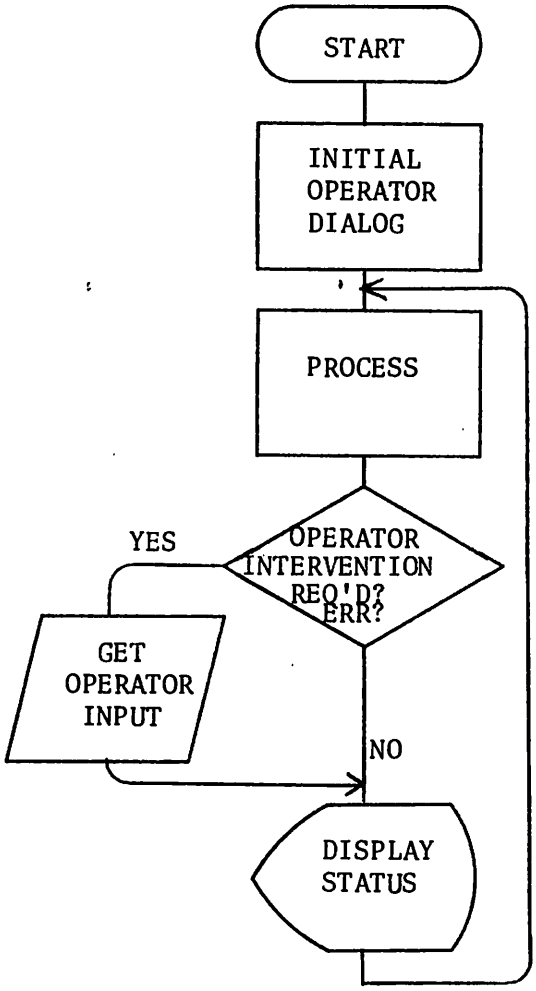
MODIFIED PROGRAM



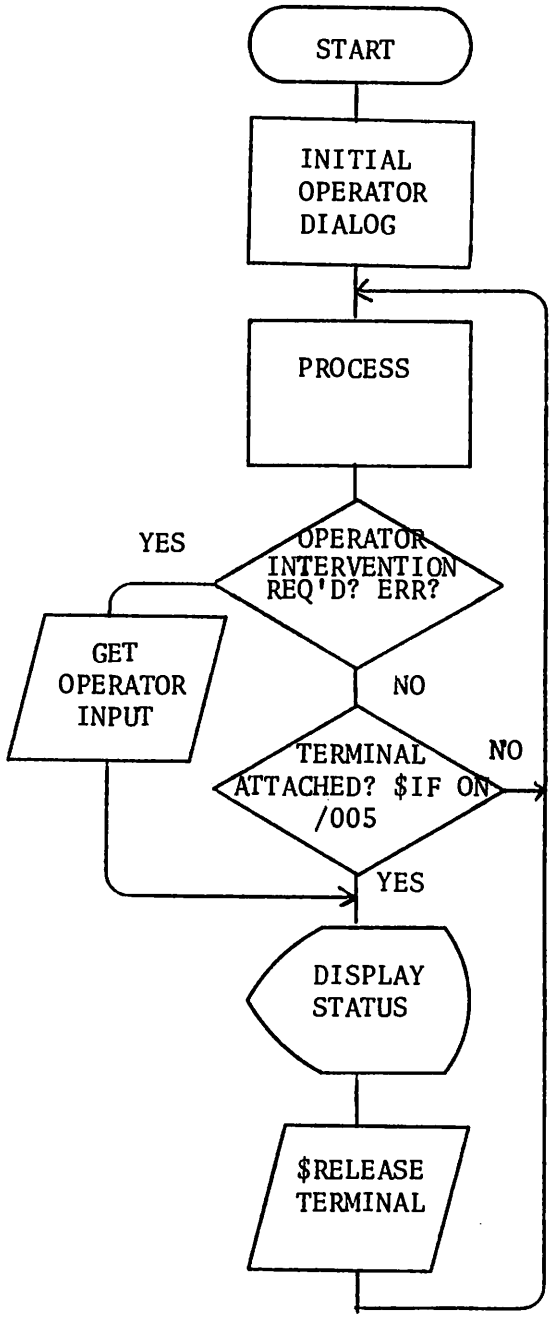


MODIFICATION OF A PROGRAM FOR BACKGROUND OPERATIONS

ORIGINAL PROGRAM



MODIFIED PROGRAM



VII. How Programs Can Share Data

In addition to the sharing of program text discussed in sections IV and V, the MVP allows data to be shared between partitions. Perhaps the most common use for these global variables is to exchange timing and status signals between cooperating programs. It is also possible for jobs to be logically divided into concurrent tasks. For instance, one partition may handle all disk I/O and pass the data to another partition via global variables for computations to be performed. In all cases mentioned so far the advantage of global variables is the speed advantage of communication within user memory instead of the alternative approach of sharing disk files. Use of global variables may also result in memory savings. Table driven systems need have only one copy of the tables contained within the same partition as the shared program text.

Global variables are identified by the character @. Global variables are completely distinct from non-global or local variables. (i.e., A$ and @A$ are separate variables). Both A$ and @A$ may be used in the same program with no ambiguity. Global variables may be defined as numeric or alphanumeric, scalars or arrays according to the same rules as local variables. The difference in defining global variables is that they may not be implicitly defined, but must be defined in DIM or COM statements. Since the purpose of global variables is to allow other partitions to look at or modify their contents it makes sense that global variables should be defined within a global partition.

An intriguing use for global variables is to report the status of a background job. If a background job declares itself global, the contents of its global variables become accessable to other partitions. The background job may now post its status to its global variables. Foreground jobs, and even partitions assigned to other terminals, may learn the status of background job by examining background job's global variables. The flow of the background job is never interrupted with status requests.

When the background job declares itself global to allow other partitions access to its global variables, its marked subroutines become available to other partitions as well. A marked subroutine may thus be included in the text of the background job for the purpose of neatly formatting the status information on the CRT when a foreground job calls it.

The following is an example of global variables and global subroutines used to display the status of a background job.

```
10 DIM @I,@N
.
.
.
60 REM INITIAL DIALOG
.
.
.
90 GOSUB 9000
100 $RELEASE TERMINAL
200 REM PROCESS LOOP
210 FOR @I = 1 TO @N
.
.
.
500 NEXT @I
600 END
.
.
.
1000 REM STATUS SUBROUTINE
1010 DEFFN '16
1020 PRINT "NOW PROCESSING RECORD"; @I; "OF"; @N
1030 RETURN
.
.
9000 DEFFN @PART "BACKGND":RETURN
```

To learn the status of the sample program, a foreground partition executes the following two statements:

```
100 SELECT @PART "BACKGND"
110 GOSUB '16
```

It is possible to call the status subroutine in immediate mode by entering the SELECT @PART statement and pressing special function key '16.

Note that the DEFFN @PART statement is at the end of the sample program. This is because implicitly defined variables are not allocated storage if they are encountered after a DEFFN @PART statement. Rather than include DIM or COM statements for all variables, it is easier to put the DEFFN @PART statement at the end of the program and call it as a subroutine. Remember that a partition does not become global until the DEFFN @PART statement is executed. The sample program chooses to wait until it has finished its dialog with the operator before declaring itself global and making its status available to other partitions.

The discerning reader can probably see that global variables can potentially suffer from the same problem as shared disk files or shared printers if two partitions access the same global variable simultaneously. To some extent, the MVP operating system automatically resolves such conflicts. Sometimes, however, it is necessary for a program to gain exclusive access to a critical region of code that performs updates on global variables.

Most BASIC statements are non-interruptable; that is, the MVP will complete the statement before allowing a breakpoint to occur. The exceptions are I/O and most MAT statements. Chapter 16 of the BASIC-2 Reference Manual contains the complete list of non-interruptable statements. Programmers may be tempted to identify the potential breakpoints in a program by the occurrence of colons and line numbers in the program text, but there are two exceptions to this rule. In the statement

```
100 @A = @A/N:ERROR @A = 9E99:GOTO 500
```

if an error occurs during the division, the first statement of the :ERROR recovery routine will be executed before the 30ms clock is checked. In other words, @A = @A/N:ERROR @A = 9E99 is treated as if it were a single statement.

The other exception is the IF-THEN-ELSE statement. Because IF-THEN-ELSE is non-interruptable, it may be used as a semaphore to prevent other partitions from entering a section of code while a critical update is in progress.

```
50 $BREAK:IF @S = 0 THEN @S = #PART:ELSE GOTO 50
.
.
.
70 MAT SORT @A$() TO W$(), L$()
.
.
.
100 @S = 0
```

If, in the example, a partition executing line 50 finds global variable @S = 0, it sets @S equal to a non-zero value and proceeds into the critical section of code. When another partition comes along, it will find that @S is not zero and will branch back to try again. The $BREAK statement is included so that waiting partitions only test the semaphore once per timeslice. The fastest way for a waiting partition gain access to the critical region is to yield its time to the partition inside the critical region½ When the partition using the critical region is finished, it sets @S = 0 to allow waiting partitions to enter.

Several variations on the IF-THEN-ELSE semaphore are possible. The semaphore variable @S was set to #PART in the example mainly as a debug aid. If the program hangs up, printing out the value of @S will reveal which partition entered but never left the critical region.