MEMMAN

Voyager CCS Memory Management Software

Software Description Document

Revised April 27, 1990

Jet Propulsion Laboratory

Gregory F. Welch

Table of Contents

1. Statement of Purpose 1

2. Introduction 2

3. Software Description 3
3.1 Logo Screen 3
3.2 Auto Screen 3
3.2.1 Categories 4
3.2.2 User Name 4
3.2.3 Input SRF Version 4
3.2.4 Load Boundaries 5
3.2.5 Master Processor 5
3.2.6 Automatic Goal 5
3.2.7 Output File/Format 6
3.2.8 Comment Status 6
3.3 SRF Reading (Parsing) 6
3.3.1 Current and Overlap Arrays 6
3.3.2 Auto Array 7
3.3.3 Temp File Creation 8
3.4 Get Word Count 8
3.4.1 Debugging the Word Count 9
3.4.2 Check Time Order 10
3.4.3 Check Duration 10
3.4.4 Check Time Gap 11
3.4.5 Handling Checkpoints 11
3.5 Automatic Management 12
3.5.1 Attempt Move 13
3.5.1.1 How Many to Move 14
3.5.1.2 Best Move/Retract 15
3.5.2 Terminate by Key Press 16
3.5.3 Check Goal 16
3.5.4 Status Prompt 16
3.5.5 Where Now? 17
3.5.6 Show Messages 17
3.6 Manual Management 18
3.6.1 Manual Management Pointers 19
3.6.2 The Track List 20
3.6.3 Fresh Screen 20
3.6.4 Get Record Data 21
3.6.5 Find Match 22
3.6.6 Get Move Data 23
3.6.7 Pointer Movement Handling 23
3.6.7.1 Up Arrow Key 23
3.6.7.2 Down Arrow Key 24
3.6.7.3 PgUp & PgDn Keys 24

3.6.7.4 Home & End Keys 25
3.6.7.5 Go To a Line 25
3.6.8 Move a Command/Group 25
3.6.9 Retract a Command/Group 26
3.6.10 Move/Retract a Cyclic 27
3.6.11 Select a Group 27
3.6.12 Checkpoint Set 0 28
3.6.13 View Command Cost 29
3.6.14 Annotated SRF 30
3.6.15 String Search 30
3.6.16 Update the Word Count 31
3.6.17 Generate Output 31

4. Associated Files 33
4.1 Temp File 33
4.2 System Parameter File 33
4.2.1 Reading the System Parameter File 33
4.3 Category File 34
4.4 SMD File (SMDF) 35
4.4.1 Reading the SMDF 36
4.5 Configuration File 36
4.6 Force File 38
4.6.1 Reading the Force File 39
4.7 Log File 40
4.7.1 Updating the Log File 40

Appendix A: Sample ASCII Files 41
A.1 Sample System Parameter File 41
A.2 Sample Category File 42
A.3 Sample SMD File 43
A.4 Sample Force FIle 44
A.5 Sample Log FIle 45

Appendix B: Glossary 46

Appendix C: Notes 47

1. Statement of Purpose

The purpose of this document is to attempt to give the reader a basic understanding of the software structure of MEMMAN and how the program functions. Generally, the software design will be discussed at a high level, with exceptions in certain critical areas. For a much lower level detailed description, the MEMMAN source code should be studied, as it is well commented where low level functions are concerned.

It is assumed that the reader has an understanding of the functions of MEMMAN from a SEQTRAN/COMSIM analyst's perspective, and that they are familiar with the operation of the program. This document offers a basic overview of the software operation, but in no way attempts to provide instructions for setting up and using the software. See MEMMAN Functional User's Guide[1] for a more detailed description of program operation.

It is also assumed that the reader has sufficient knowledge of Pascal in general, and the Turbo Pascal (TP) environment in particular. This document will make no attempt to explain Pascal or the Turbo Pascal system. For such background, the user should refer to the Turbo Pascal Owner's Handbook[2].

2. Introduction

When the Voyager ground data systems were originally developed, it was never perceived that both CCS processors would be loaded simultaneously. Therefore the problem of dividing a sequence of commands between processors has only been addressed in just the past several years. As a result, the MEMMAN program (Voyager CCS Memory Management) was developed as a tool to be used in assisting the SEQTRAN/COMSIM analyst with the task of efficiently dividing the list of sequence events (in the SRF) between the two CCS processors.

MEMMAN reads the original SRF, parses the contents, divides the commands between the two CCS processors using various constraints, and produces either a Univac runstream or a new SRF as the output. In the case of the runstream output, the runstream would be used to modify the original SRF to in effect create a second (new) SRF. The new (or modified) SRF would then be used as the input to SEQTRAN.

By using MEMMAN, the SEQTRAN/COMSIM analyst can eliminate the unusual cycle of successive SEQTRAN runs normally needed to achieve a CCS load within the defined memory limits. In addition to reducing the number of SEQTRAN runs needed, MEMMAN goes one step further and attempts to optimize the movement of the commands so as to minimize the overhead CCS word cost normally incurred when moving commands from one CCS to the other.

Although MEMMAN is not required to produce a sequence, it is a tool which can greatly reduce the amount of time (hence work) required to successfully and efficiently translate an SRF using SEQTRAN.

3. Software Description

MEMMAN was originally developed using Borland International's Turbo Pascal version 4.0, while later updates were implemented using version 5.0 which offers many nice features from both a software implementer's and designer's standpoint.

The hierarchy of the source code files, show in Figure 1, consists of the main program file (Man_Main.PAS) at the highest level (the primary file), which uses several TPU files at the next lower level, which may each in turn use several other TPU files at yet lower levels. All of the TPU files are identical in syntax, so their hierarchy is solely determined by who uses whom.

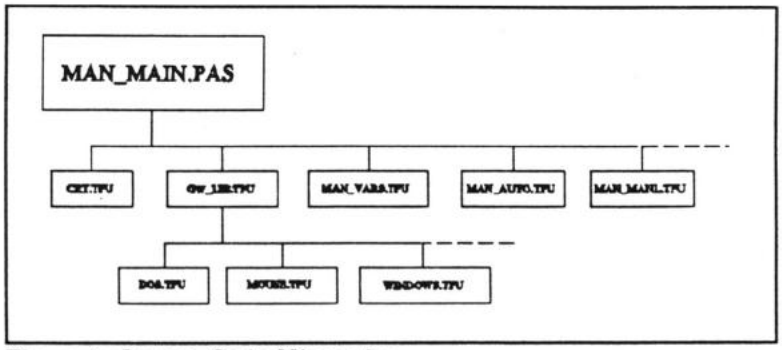

Figure 1: Source Code Hierarchy

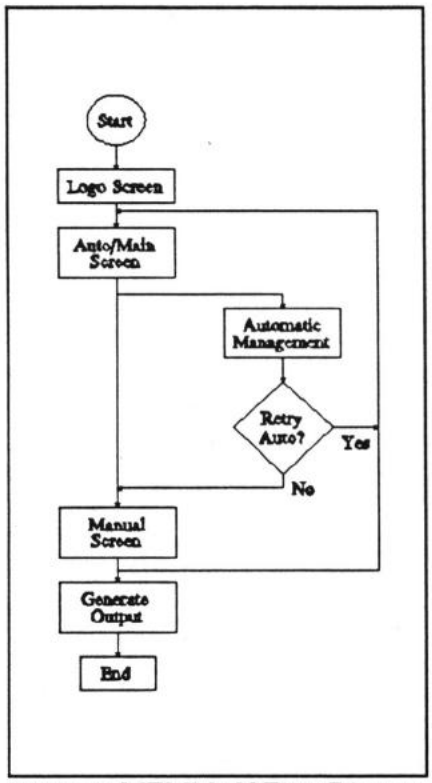

Figure 2: MEMMAN Flow Diagram

As indicated in the introduction, each possible path taken for management leads to the manual management screen. In a sense, the manual management screen is the target screen for the user, and the automatic management is simply an optional way of getting there. A general outline of the software flow can be seen in Figure 2.

3.1 Logo Screen

The very first screen the user will see after starting the program is a temporary introduction or logo screen. This screen, displayed by a call to the LogoScreen procedure found in the Man_Scr1.PAS TPU. It displays some general copyright information, along with the current MEMMAN version number. The version is stored in a typed constant called *Version*, and is converted to a string in the Man_Init.PAS TPU.

The logo screen disappears under two conditions. First the IBM PC system clock is checked to get the initial routine entry time. A loop is then entered where the keyboard is repeatedly checked for a key press, and the system clock is repeatedly read to get the current time. This is repeated until either a key is pressed, or until five seconds has elapsed since the initial routine entry time. When either of these occurs, control is passed to the Auto Screen.

3.2 Auto Screen

The main input screen for MEMMAN is referred to as the Auto Screen. This screen is displayed and processed by a call to the SelectMoveItems procedure, found in the Man_Scr1.PAS TPU. The procedure SelectMoveItems reads the category file, displays the auto screen and processes keyboard input until the user chooses to begin the automatic management, or to proceed directly to the manual management screen.

The actual printing of the auto screen is handled by calls to two separate routines, PrintSelectScreen and UpdateScreen. PrintSelectScreen simply prints the template for the screen, with all of the prompt text and instructions. UpdateScreen is called to print the list of categories, and to fill in the prompt blanks with the present data.

SelectMoveItems checks all of the prompt fields to make sure that they contain valid information before it will allow the user to proceed to either the automatic or the manual management.

3.2.1 Categories

The number of command categories is stored in a counter variable called *CatCount*, and is updated by a call to the Man_Cat.PAS TPU routine ReadCatFile. When SelectMoveItems is called, it checks the value of *CatCount* and calls ReadCatFile if it is zero. See 4.3 Category File for more information about ReadCatFile.

The UpdateScreen procedure loops through the *Category* array (a dynamic variable), changes the screen color based on the *Moveable* field (of the *Category* array record) and prints the *CatName* field between two brackets in a field which is *CatNameLen* characters wide. This loop repeats for all *CatCount* categories. Finally, a pointer (variable *Y*) is set at the first category item, and an arrow is printed on the screen which points to the first category name.

If the user presses the up or down arrows, the pointer (and arrow) is moved through the list of categories, one category at a time. If the user presses the space bar, SelectMoveItems calls the procedure ToggleLabel and the category presently pointed to is toggled from moveable to non-moveable (or non-moveable to moveable). This also causes the category's color to change on the screen.

3.2.2 User Name

When the user presses the "U" key, SelectMoveItems calls the GW_Lib.PAS TPU procedure GetInput to get the user name string. This string is then later placed in the MEMMAN log file (see 4.7 Log File). The user name string length is determined by the *UserLen* global constant.

3.2.3 Input SRF Version

When the user presses the "S" key, SelectMoveItems enters a loop which repeatedly attempts to get an SRF version string, until there are no errors in the input. The loop calls the GW_Lib.PAS TPU procedure GetInput to get the SRF input version string. The length of the string is then checked against the global constant *SrfVerLen* to make sure it is at least SrfVerLen characters long. Then the first character of the input SRF version is checked to make sure it is either an "A" or a "B". If any of these tests fail, then the loop is repeated, otherwise it is exited.

Next, SelectMoveItems calls the OpenFiles procedure, to see if a configuration file exists for this SRF version (see 4.5 Configuration File). If a configuration file exists, then it is read and any input which was previously entered at this screen is overridden with the data from the file. If one does not exist, then present input field data is left as is and a configuration file is created with the present *InSrfVer* variable.

The input file name is then assembled by appending ".SRF" to the end of the input SRF version string. The output file/format is then determined, based on the variable *OutputFmt*. *OutputFmt* has a default state in the system parameter file (see 4.2 System Parameter File), or can be toggled from the automatic screen (see

3.2.7 Output File/Format).

If the *OutputFmt* variable is set to a value of FmtRS, then the variables *OutSrfVer* and *OutSrfFilename* are cleared as they are not needed to generate an output runstream.

If the *OutputFmt* variable is set to a value of FmtSRF, then the Man_Misc.PAS TPU function BumpID is called (with the input SRF version as its parameter) to get the output SRF version string in the variable *OutSrfVer*. The function BumpID takes the last character in the parameter passed with it and in a sense increments it to the next letter (character). If the character is at "Z" then it is bumped to "A" again. Note that this routine will also bump numbers or any character for that manner. *OutSrfFilename* is then created by appending ".SRF" to the OutSrfVer string.

3.2.4 Load Boundaries

When the user presses the "S" key, SelectMoveItems calls GetInput four times in order to get the memory load boundaries. The four calls get the lower & upper boundaries for CCSA, and then the lower & upper boundaries for CCSB. Because the GetInput procedure allows *Lo* and *Hi* limits for the input, the call for each boundary can be constrained to be within up to the minute load constraints.

When the system parameter file is read (see 4.2.1 Reading the System Parameter File) the variables *LowBoundA*, *UpBoundA*, *LowBoundB* and *UpBoundB* are initialized with default values. These values are then used to assist in constraining the load boundaries for the particular *InSrfVer*.

When getting the input for the lower CCSA memory limit *LwLimA*, GetInput is called with *LowBoundA* and *UpBoundA* (from the system parameter file) as the *Lo* and *Hi* input limits. If the value the user inputs for *LwLimA* is not between these values, then the user will get a message and will have to re-enter the *LwLimA* value. Once the value has been obtained and is between the limits, *LwLimA* is then used as the *Lo* input limit for the *UpLimA* input. In other words, GetInput is called with *LwLimA* and *UpBoundA* as the Lo and Hi input limits. This process assures that the user doesn't inadvertently place the upper limit (*UpLimA*) below the lower limit which they just entered (*LwLimA*). The same process is followed for CCSB.

3.2.5 Master Processor

When the user presses the "M" key, SelectMoveItems attempts to toggle the global master (and slave) processor variables *PriProc* (and *SecProc*) between values of ProcA or ProcB.

To begin with, the variable *PriProc* is checked to make sure it contains a valid value. This is because if the user has not entered an input SRF version, then there will have been no master processor determined yet. If *PriProc* does not contain a valid value, then the user will get a message telling them to enter an input SRF version.

A case statement is then used to toggle the values of *PriProc* and *SecProc* to their opposite values, and to update the screen. Finally, the variables *ConfigStat* and *OldStat* are updated to be the same as *PriProc*. The variable *ConfigStat* and *OldStat* are used while reading the SRF into memory (see 3.3 SRF Reading).

3.2.6 Automatic Goal

When the user presses the "A" key, SelectMoveItems attempts to toggle the global variable *AutoGoal* between

the values JustUnder and Balanced. A case statement is used to check the current value and switch to the new one. The variable *AutoGoal* is used later during automatic management.

3.2.7 Output File/Format

When the user presses the "A" key, SelectMoveItems attempts to toggle the global variable *OutputFmt* between the values FmtRS and FmtSRF. A case statement is used to check the current value and switch to the new one. If the state is toggled to FmtRS then the variables *OutSrfVer* and *OutSrfFilename* are cleared, as they are not needed for runstream output. If however the state is toggled to FmtSRF, then BumpID is called to get the variable *OutSrfVer*, and ".SRF" is appended to that to get the *OutSrfFilename*, (see 3.2.3 Input SRF Version).

3.2.8 Comment Status

When the user presses the "C" key, SelectMoveItems attempts to toggle the global boolean variable *ShowComments* between the values "true" and "false". If *ShowComments* is false, then the Man_Load.PAS TPU procedure LoadArray will not write any records with an apostrophe in column one (any comments) to the Temp file (see 4.1 Temp File). This controls whether or not any comments are seen on the screen during manual management.

If, however, this is not the first time at the auto screen for this MEMMAN run, then the *TempWritten* flag would have a value of "true", signaling that the Temp File has already been written. In such a case, the Temp File must be recreated with (or without) the comments. Remember for example, the user could go to the manual screen, (the Temp File would have already been created), return to the auto screen, toggle the comment status, and then wish to return to the manual screen.

3.3 SRF Reading (Parsing)

After the user has entered necessary information at the Auto/Main screen, they would press the "G" key to proceed (or "Go") with automatic management, or <F3> to skip the automatic management and proceed directly to the manual management.

At this point, if this is the first attempt to manage (either automatically or manually) for this particular MEMMAN run, a call to the Man_Load.PAS TPU LoadArray procedure reads the SRF from the disk file, into memory. As the file is read, each record is parsed, and then needed information is placed in one or more of several arrays. Of those arrays, the *Current* and *Overlap* arrays are the largest, and in a way the most important. These arrays are dynamic arrays, allocated in the memory heap at run time. Together they contain a complete list of all of the commands in the SRF as they would be assembled in the current or overlap region of CCS memory (by SEQTRAN).

If a command read from the SRF has no processor specification, then the LoadArray procedure uses the *ConfigStat* variable to determine which processor to assign the command to. If a CONFIG macro is encountered in the SRF, then *ConfigStat* is updated per the CONFIG macro specifications, (the variable *OldStat* is used to return *ConfigStat* to its initial value after a CONFIG 5 was encountered in the SRF).

3.3.1 Current and Overlap Arrays

In general, the *Current* and *Overlap* arrays are arrays of command descriptions. The descriptions contain

information which is relevant to managing the CCS memory, such as the command's cost, the processor (if stipulated), the auto move status (is the command moveable automatically?), the manual move status, and various other needed parameters. The contents of the *Current* and *Overlap* arrays can be more completely view by looking in the TPU called Man_Vars.PAS, and specifically at the record type called *Item_Info.*

The parsing of the commands is accompanied by several general constraint checks, and possibly specific checks for a particular command. For example, if a force file exists for the SRF being managed, then the items in that file will be checked to see if the command presently being parsed should be forced to either processor. Also, the command must be checked to see if it is included in any of the categories the user toggled at the Auto/Main screen. If so, then the mobility of that command may be constrained by that category's status. Generally, the force file takes precedent over any decisions made by the MEMMAN program, so it is the last place checked for any constraints.

When a command is parsed in the SRF, a decision must be made as to which array (*Current* or *Overlap*) the command should be placed in. Here MEMMAN follows virtually the same rules as SEQTRAN would when assembling in either region of memory. If a command's time tag is after the "next window" time (always stripped from the WINDOW macro in the SRF) then it will be placed in the sequence overlap. If a command occurs just prior to the "next window" but the command duration causes the end of the command to occur after the "next window" time, then the command will be placed in overlap. Finally, if even one cyclic call appears after the "next window" time, then the whole cyclic definition and all of the calls will be placed in the overlap region of CCS memory. Here, in order to keep all of the definitions together in one place, the definition will remain in the *Current* array, but the cyclic calls will be placed in the *Overlap* array.

Thus it is possible to have a cyclic call which is surrounded by current region events, which may actually be an overlap event. This occurrence is really only significant when determining the word count (see 3.4 Get Word Count), and when displaying the manual management screen (see 3.6 Manual Management and 3.6.3 Fresh Screen).

3.3.2 Auto Array

In addition to the *Current* and *Overlap* arrays, while reading the SRF a special array for automatic management is filled. This array, called *Auto,* is also a dynamic array. It contains information which is used in the process of looking for commands to move back and forth between processors during an automatic management attempt.

The *Auto* array contains entries for groups of specific commands which have like characteristics with respect to their auto characteristics. It contains such information as the command's cost (appears to be redundant with the *Current* and *Overlap* arrays), the number of times in a row this command appears in the SRF with common attributes.

For example, if a command such as a DC2P was encountered (for the first time) while reading the SRF, and it was determined that it was moveable automatically, then an entry in the *Auto* array would be added for that command. This entry would be marked one of two types, either *Use* or *Skip* type. If the command can be moved automatically, then it would be considered type *Use*, otherwise type *Skip*. When the next DC2P was encountered, it would be checked for the same conditions. If its move status was the same as the last command of the same name in the table, then instead of adding a new entry, a counter of the available DC2Ps (available to move) would be incremented. If its move status was different than the last command of the same name, then a new entry would have to be added, and its *Avail* count would be set to one.

The result of this process is that a single command may result in multiple entries in the *Auto* array, entries which are gathered into groups of Use and Skip types. Figure 3 depicts a short sequence, and how the example of DC2Ps might appear in the *Auto* array.

In this example, the first CONFIG sets the master processor as CCSA (as noted). Therefore, any commands which are not in CCSA are not "moveable" under any circumstances. Notice that in this example, the first two DC2Ps are moveable because they are in the master. However, the next two are "configged" to CCSB and are therefore not moveable. Note that for this element in the array, there are zero available and two occurrences of a Skip type of DC2P. Any commands which must be skipped are entered in the array with a positive number of occurrences, and an available value of zero. This is so that during the automatic management attempt, these two DC2Ps will be skipped (not used).

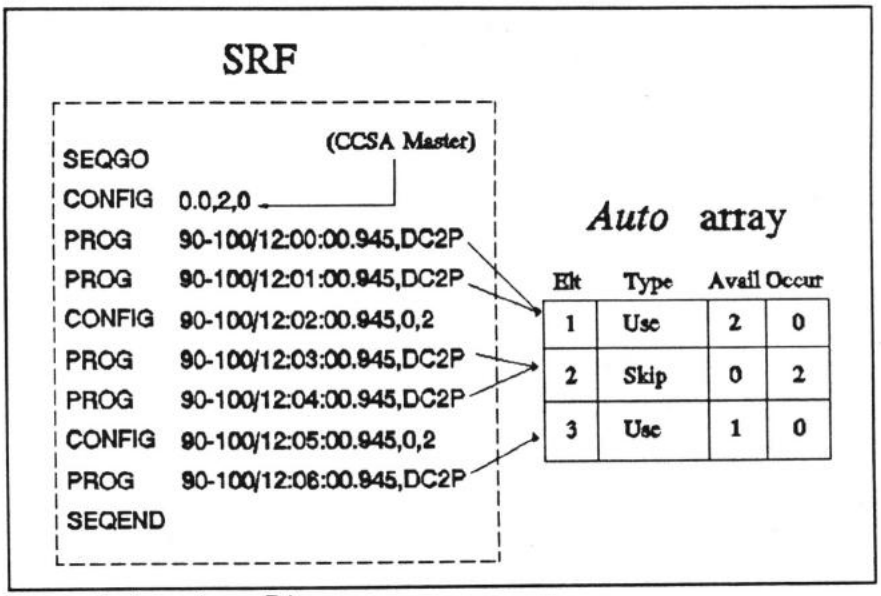

Figure 3: Auto Array Diagram

When the final DC2P was encountered, it just so happens that the last DC2P placed in the array (table) was a Skip type, and because this particular one is moveable, a new Use entry was added to the array.

3.3.3 Temp File Creation

The last major function performed during the reading (parsing) of the SRF is the creation of the Temp file, (see also 4.1 Temp File). Almost every record read from the SRF is placed in the Temp file. The exceptions are all of the header records in the SRF, and any comment records, if the Comment option is toggled off at the Auto/Main screen.

The Temp file is created only to facilitate the screen handling during manual management. It is stepped through during manual management, rather than the actual SRF, in order to remove much of the decision making (program logic) that would otherwise be required.

3.4 Get Word Count

Once the SRF has been read into memory (see 3.3 SRF Reading) and a few times later on in the program execution, the routine GetCurrentCount (Man_Gcnt.PAS TPU) is called to determine the sequence word cost, in the present memory configuration. This routine has a partial effect of running SEQTRAN, since the result is the word count for each processor.

To determine the word count, GetCurrentCount simply steps through the *Current* and *Overlap* arrays, accumulating the command costs into a total for each processor. The word count totals are accumulated in the variables *MemA* and *MemB*. These variables are initialized to reflect the minimum word cost to load a sequence (6 words), as well as any special loading routines (such as CCSTIM, etc...) which for one reason or another are not treated like normal commands. In such a case, there will be no entry in the *Current* array for the macro, but the macro cost will be reflected in the variables *StartA* and *StartB*.

As one of the final steps in GetCurrentCount, the present command's duration is stored in the variable *LastDuratA* (or *LastDuratB*), the command's name in the variable *LastCmdA* (or *LastCmdB*) and the command's time tag in the variable *LastA* (or *LastB*). These variables are to be used in checking several constraints with the next command (in the same processor).

In the process of accumulating the various command costs into a single CCSA and CCSB word count, several checks for CCS overhead costs must be made. Such costs can be incurred when one of several conditions occur. Such conditions could include macro commands which are out of time order, a macro command which due to its duration overlaps the next macro command (in time), or a macro command which occurs more than some maximum amount of seconds after the last command in the same processor. Explanations for each check can be seen in the following sections.

3.4.1 Debugging the Word Count

To assist with word count errors in MEMMAN, the Man_Gcnt.PAS TPU contains several helpful routines (with logic) which can be activated if needed. This capability will allow the programmer to track the cost accumulation as it is being done.

When the logic is enabled, each time the routine GetCurrentCount is called, a special prompt window will ask the user (the programmer in this instance) if they wish to debug either the CCSA or CCSB. If the user responds "N" (No) then the word count will proceed normally. If the user responds "Y", then during the word counting, every time any words (including CCS overhead) are added to a processor's word count for a known reason (listed above), a special debugging window will pop-up with a message about the word cost. If the reason for the added word cost is not known by the routine, it will display a pop-up window to that effect also. The debugging logic can be enabled for just one processor, or both processors.

Because automatic management involves the repetitive process of moving commands and then determining the CCS word cost (see 3.5 Automatic Management), the final call to GetCurrentCount may be the only relevant call to monitor. If this is the case, simply run the automatic management once without debugging to see how many attempts at management it takes to achieve the goal. Then enable the logic, rerun and answer "N" to the debugging prompt until the last call of GetCurrentCount.

Use the following steps to enable the debugging logic:

(1) Edit the file Man_Gcnt.PAS (a TPU)
(2) Search for the string {greg} (a comment). Press Ctrl-Q-F, enter {greg} as the text to find, enter gun for the search options. The line of text found should read something like {greg} {Ch := promptuser('Debug A?'.....
(3) Delete the comment brackets (left and right) both before the Ch and after the ; of the first two commented-out lines to debug CCSA, and the second two commented-out lines for CCSB.
(4) Re-compile (make) the program.

Now when the program is run, the special debugging prompt window will appear each time GetCurrentCount is called. Remember to restore the comment brackets when the debugging is complete (before delivering).

3.4.2 Check Time Order

One of the various checks for CCS overhead costs involves monitoring the time tags of events in the order they appear in the SRF. This is done with a call to the procedure CheckTimeOrder. Almost without exception, if an item appears physically in the SRF after an item which has a later time tag, then the two commands are considered "out of time order". In this instance, SEQTRAN will assemble some additional CCS words to start a new time event region in the CCS, and so MEMMAN must cost these additional words.

The main exception to this check is when either the command being checked or the prior command in the same processor was a BRKPNT macro call. For this macro, the time tag has no effect. It is the physical placement in the file that determines the location of the CCS breakpoint (the new time event region). For this reason, the time tags in BRKPNT macros calls are the exception to the time order check.

If a command is found to be out of time order, then the *Overhead* field of the command's *Current* or *Overlap* array element is assigned a value of one. This field is then checked when the manual screen is printed. If the field has a value of one, a special mark is placed on the screen next to the command to signify the fact that it is the first command in a new time event region. See also 3.6.3 Fresh Screen.

3.4.3 Check Duration

In addition to looking for out of time order commands, MEMMAN also checks the duration of time required to complete many of the commands when they actually execute on the spacecraft. If the execution of a command will not be complete until after the start time of the next command, then SEQTRAN will assemble some additional CCS words to start a new time event region in the CCS.

During the reading of the SRF (see 3.3 SRF Reading) the duration (execution time) of each command is gathered for use while determining the word count in the procedure GetCurrentCount. Most commands essentially take no (zero) time to execute, but those that take a specified amount of time to execute must be included in the procedure ProcessCommand (Man_Pcmd.PAS TPU), as opposed to the SMD file, (see 4.4 SMD File). Here the variable *Duration* is updated with the appropriate value in seconds.

Every time a command is processed in GetCurrentCount (see 3.4 Get Current Count) the procedure CheckDuration is called to compare the time tag of the present command with the sum of the last command's time (in the same processor) and the last command's duration (in seconds). It is important to grasp the concept that the time & duration of the previous command is always checked with respect to the time of the present command.

If the comparison shows that the last command will finish executing after the present one is called, then the variable *MemA* (or *MemB*) will be incremented to account for the fact that SEQTRAN will assemble some additional CCS words to start a new time event region in the CCS. In addition, the *Overhead* field of the command's *Current* or *Overlap* array element is assigned a value of one. This field is then checked when the manual screen is printed. If the field has a value of one, a special mark is placed on the screen next to the command to signify the fact that it is the first command in a new time event region, (See 3.6.3 Fresh Screen).

As is the case with out of time order events, special checks must be made with respect to checkpoints. See

also 3.4.5 Handling Checkpoints.

3.4.4 Check Time Gap

As a final check during the procedure GetCurrentCount, MEMMAN calls CheckForTE to compare the time of the present command with that of the previous one (in the same processor) to see how much time has elapsed. If the time (delta) is greater than the maximum time delay possibly held by a CCS word, then extra words must be added to account for the fact that SEQTRAN will assemble some extra overhead words.

The added words may reflect two possible scenarios. Under normal circumstances SEQTRAN would simply start a new time event region in the CCS, costing an additional three CCS words.

However, if the time gap occurs either in a checkpoint-breakpoint region, or any time after a CHKPNT macro call has been made in the present sequence, SEQTRAN will assemble commands to switch the CCS to hours mode, delay a certain number of hours (instead of seconds), switch back to seconds mode, and then delay any remaining seconds. All of this costs a total of four words instead of the three in the first scenario. SEQTRAN handles the second scenario as it does so that a checkpoint-breakpoint region will not be broken up by a new time event region occurring in the middle of it.

If an out of range time duration is found, then the *Overhead* field of the present command's *Current* or *Overlap* array element is assigned a value of two. This field is then checked when the manual screen is printed. If the field has a value of two, a special mark is placed on the screen next to the command to signify the fact that a switch to hours mode (and back) will be inserted by SEQTRAN just prior to the marked command. See also 3.6.3 Fresh Screen.

3.4.5 Handling Checkpoints

To manage the special checkpoint conditions related to checking out of time order situations, there are two logical variables used called *ActiveChkPnt* and *Checkpoint*. If a CHKPNT macro is encountered while stepping through the *Current* and *Overlap* arrays to determine the word count, then both of the checkpoint flags mentioned above are set to a value of "true". As soon as a BRKPNT macro is encountered (or an effective breakpoint is caused by a command duration or an out of time order occurrence) then the *ActiveChkPnt* flag is cleared, but not the *Checkpoint* flag.

The variable *Checkpoint* is set to a value of "true" whenever the first CHKPNT macro call is encountered in the SRF, and is usually never cleared. This flag is supposed to function not unlike the CHKPNT symbol in SEQTRAN. This symbol is set true when SEQTRAN encounters a CHKPNT macro. When this flag is "true" in SEQTRAN, it will no longer create a new time event region for large time gaps. Instead it will follow the steps mentioned above in 3.4.4 Check Time Gap which costs an extra one CCS word every time it occurs.

To save CCS words, a common memory management practice is to place the command CHKPNT SET 0 after every BRKPNT macro call in the SRF. This will clear the CHKPNT symbol in SEQTRAN, allowing SEQTRAN to once again start a new time event region at any large time gaps, thus saving one word for each occurrence. This practice can be implemented from within MEMMAN by toggling the CHKPNT SET 0 option off or on, (see 3.6.12 Checkpoint Set 0).

The variable *ActiveChkPnt* is only set to a value of "true" from the time a CHKPNT macro call is encountered until the time a BRKPNT macro call is encountered or a combination of commands causes an effective

breakpoint. In other words, it is true only while a checkpoint region is active. If either an out of time order or a long duration situation (both described above) causes an effective breakpoint while the *ActiveChkPnt* flag is set, this is a critical occurrence because it has the same effect as placing a breakpoint (BRKPNT) at that location in the SRF. Therefore any commands which appear after the problem events (before the actual BRKPNT macro call) will no longer be governed by the preceding checkpoint, and the user should be warned as such.

3.5 Automatic Management

The MEMMAN automatic management is essentially an iterative process of getting a word count, moving or retracting needed commands, and then seeing if the goal specified by the variable *AutoGoal* has been met. Figure 4 shows how the process works. When the user presses "G" (Go) at the auto management screen to begin automatic management, the SRF is read into memory and the automatic management process begins by a call by MemMan (Man_Main.PAS) to the AutoManageMem procedure.

The first thing AutoManageMem does is to make an initial call to GetCurrentCount (see 3.4 Get Word Count) to see what the CCSA and CCSB word counts will be after translating the input SRF in its current state (with no intelligent memory management - force file not withstanding). Keep in mind that if there is no force file, then generally the word count will be well over the limit of one processor, and well under the limit of the other. If however there is a force file, and it is so extensive as to almost completely manage the memory without any intelligent attempt by MEMMAN, it is possible that the CCS word counts for both processors will prove to be very close to the target, thus requiring little or no intelligent management by MEMMAN.

After the initial word count is done, the main automatic management loop is entered. First, an attempt is made to move any possible commands so that the word count will be adjusted to the necessary levels. It is important to understand the terms move and retract as they apply to MEMMAN. To move a command implies moving from the master to the slave processor. Any movement from the slave back to the master is considered a retraction. In other words, command movement should be thought of in terms of the master processor as the default.

The initial word count establishes the base count to be used during each attempt to move commands around in the AttemptMove procedure. Every time a command is moved (from the master to the slave) its cost is subtracted from the master processor total and added to the slave total. Every time a command is retracted (from the slave to the master) its cost is subtracted from the slave processor and added to the master. This AttemptMove procedure continues as sort of a fast estimate of the effects of command movement.

Once the attempted moves (and retracts) appear to have adjusted the base word count sufficiently enough to meet the automatic goal specified by the *AutoGoal* variable, then GetCurrentCount is called again to get an accurate word count (more accurate than the fast estimate in AttemptMove). Remember, GetCurrentCount checks for time event regions, time gaps, and other situations that will effect the word count.

The new, more accurate word count is checked to see if the attempted moves (and retracts) have truly adjusted the base word count sufficiently enough to meet the automatic goal specified by the *AutoGoal* variable, or if the estimate made by AttemptMove was off, and more movement (or retraction) must be done.

If the goal was achieved, then the user is prompted to see if they are happy with the results. The user has several options available to them at this point (see 3.5.4 Status Prompt), including (if the *AutoGoal* was just

under) further auto management of remaining unmoved commands.

If the user is content with the results, then AutoManageMem calls ShowMessages to display any error or warning messages that accumulated during GetCurrentCount. Finally, under normal circumstances MemMan (Man_Main.PAS) calls the MoveManual procedure to begin the manual management.

Because the automatic management is not perfect, there is a chance that circumstances will arise where the process will in effect enter an endless loop. In this case the user must be able to interrupt the automatic management loop. For instance, it may move two commands, call GetCurrentCount, and then retract the same two commands, over and over again. The routine checks for a keyboard press each pass through the loop. If a press is detected, then the user is prompted to respond yes or no to whether they wish to terminate the automatic process (see Figure 4). If they answer no, then the process will continue. If the user responds yes, then the routine will call GetCurrentCount one last time and the exit (the loop).

3.5.1 Attempt Moves

As mentioned above in 3.5 Automatic Management, the procedure AutoManageMem begins with a very accurate word count, obtained by an initial call to GetCurrentCount. However, as it proceeds to loop through the *Auto* array (list), moving and retracting commands, the word count becomes less accurate. This is because in the AttemptMove procedure's main loop, commands are moved and retracted without regard for any overhead associated with (or created as a result of) the movement or retraction.

The chief concern of the AttemptMove procedure is to find and move (or retract) the commands which make the most sense as far as the command is concerned. It is the main concern of the AutoManageMem procedure to then check the word count from AttemptMove. This process in a sense mimics that used previously by the SEQTRAN/COMSIM analyst.

Each time through the AttemptMove main loop, (see Figure 4), the present word count for each processor is checked to see if the goal specified by the *AutoGoal* variable has been met, or just how far away we are (how many we need to move or retract). The number of words to be moved or retracted, stored in the variable *HowMany* (see 3.5.1.1 How Many to Move), is then checked to see whether it is less than or equal to the value of the

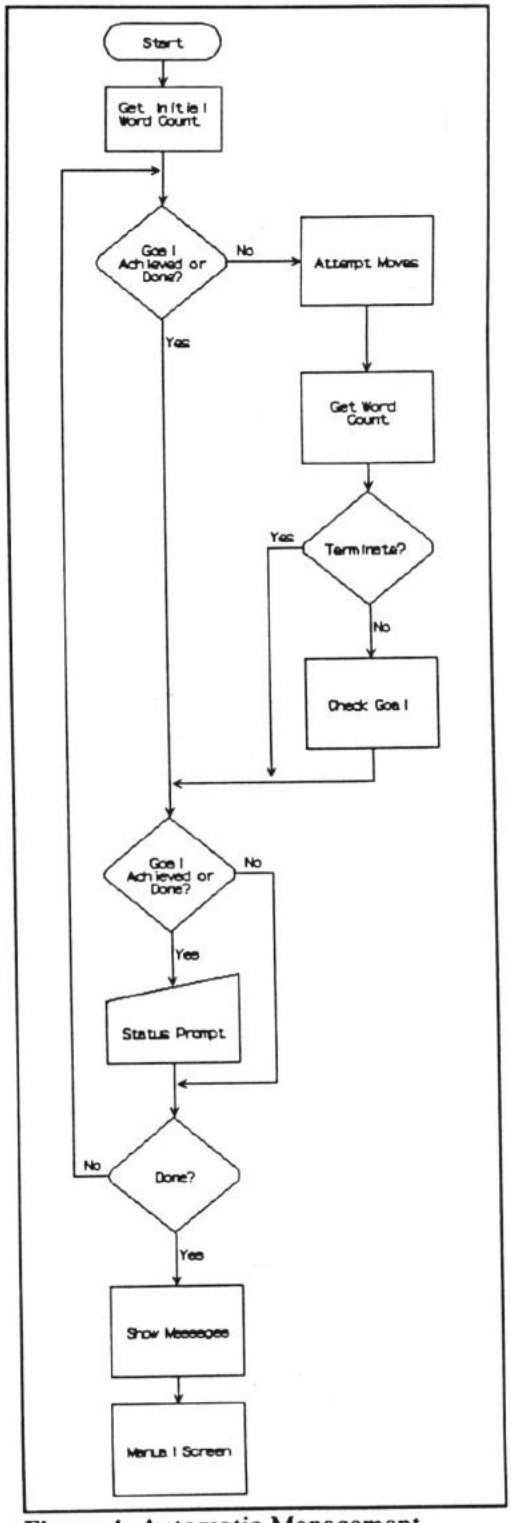

Figure 4: Automatic Management

variable *Tolerance*, to see if the attempt is done.

If the value of *HowMany* implies that the management is done, AttemptMove is ended, and an accurate word count is done (GetCurrentCount) in AutoManageMem. If the value of *HowMany* implies that there is more management to be done, then *HowMany* is checked more closely to see whether there are too many words in the master or the slave processor.

If the word count is over in the master, then commands must be moved to the slave. Likewise, if the count is over in the slave, then commands must be retracted back to the master. If over in the master processor, BestMove is called to search the *Auto* array for the best candidate for movement at the present time. The same process is used for a word count which is over in the slave processor, however the procedure BestRetract is called instead.

As a command is selected to be moved (or retracted), the *Auto* array is updated to show that one less (or more) of that type of command is available to move (or retract), and the CCSA and CCSB word counts are updated to reflect the transfer of words from one CCS to the other.

Generally, this process is repeated until either the goal of just under or balanced memory has been achieved, or until the list of moveable commands (the *Auto* array) has been exhausted.

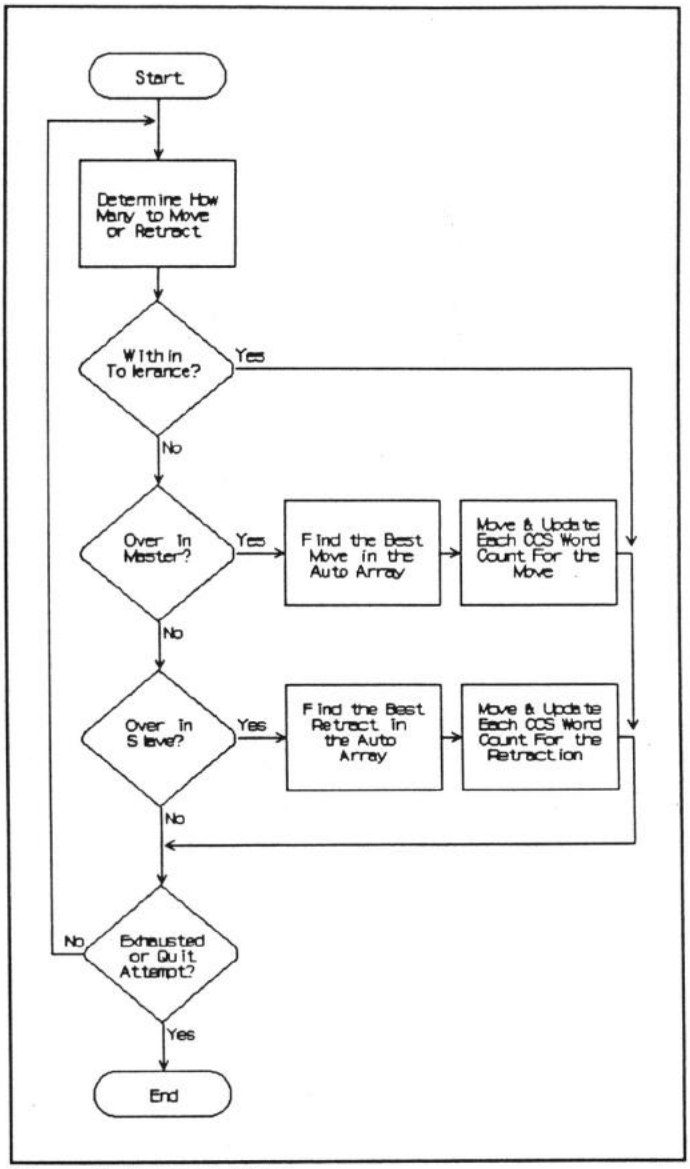

Figure 5: Attempt Move

In either case, AttemptMove is exited, and AutoManageMem calls GetCurrentCount to get an accurate word count, and then the new word count is checked against the goals again. If this second check reveals that the goal has in fact been reached, or that it can not be reached because all possible moves have been exhausted, then the user is prompted for further action by a call to the StatusPrompt routine.

3.5.1.1 How Many to Move

For the AttemptMove routine to begin moving or retracting commands, it must have determined how many words need to be moved.

When the automatic goal is to get the word count just under the master limit, the process is fairly straight forward. The difference between the limit for the master and the actual word count is how many need to be moved. This value is stored in the variable *PriDiff*. Therefore, the variable *HowMany* is simply assign the value of *PriDiff* in this case. If the result is positive, commands must be moved. If it is negative, they must be retracted. For example, if the master processor is ten words over, AttemptMove will attempt to move approximately ten words (possibly ten plus the *Tolerance* value) from the master to the slave.

However, if the automatic goal is to balance the amount of free memory in the two CCS processors, then the calculation is slightly more involved. Here the limit-minus-actual differences must be known for both processors. *PriDiff* holds the difference between the primary (master) limit and the actual word count, and *SecDiff* holds the difference for the secondary (slave). Then, the difference between these two variables is basically divided by two, and then stored in the variable *HowMany*.

Whether the goal is to balance the free memory in the two CCS processors, or to end up with the word count just under the master, the variable *HowMany* will contain the number of commands to move (if positive) or retract (if negative).

3.5.1.2 Best Move/Retract

After the value of the variable *HowMany* has been determined, AttemptMove will actually attempt a move or a retraction. Rather than simply moving (or retracting) the next available item in the *Auto* array, AttemptMove calls either BestMove or BestRetract which search the *Auto* array looking for the best possible candidate. Both functions return a pointer which is assign to the variable *MListPtr*, a pointer to the prospective entry in the move list (the *Auto* array).

When looking for the best possible commands to move or retract, a rule of thumb used is that the more words that can be transferred by less movement the better. It is better to move one event of tremendous cost than to move many events of lesser cost. The reason for this is that the more commands that are moved (or retracted) the larger the possible overhead as a result of that moving or retracting.

But of course, to move the command with the largest word cost might be to move too many words. Therefore, an attempt is made to find the command with the largest cost which does not exceed the value of *HowMany*. Each time BestMove or BestRetract is called, they peruse the *Auto* list to see which entries are of Use type, and whose *Avail* field is still greater than zero. Among these, they look for the entries with costs closest to that needed in order to match the value of *HowMany*.

Within each routine (BestMove and BestRetract) is another routine called FindEvent. This procedure is called once an entry in the *Auto* array has been settled on as being the best choice. FindEvent will move forwards or backwards through the *Current* or *Overlap* arrays looking for the command which matches the particular entry in the *Auto* array, (see 3.3.2 Auto Array).

In the calls to BestMove and BestRetract, the *Point* parameter contains the returned pointer to the particular command in the *Current* or *Overlap* array. The *EType* variable contains a value of Cur or Ovr, signifying which array (*Current* or *Overlap*) the command is located in. The variable *MoveCost* gives the total cost of moving the proposed command (or cyclic & calls, or platform events).

If the BestMove or BestRetract procedures return a cyclic as a candidate for transfer, then the cyclic definition along with all of the calls will be moved. If the candidate is a platform command, then all of the platform

related commands will be transferred. These are two special cases of command movement, which are specially handled within BestMove and BestRetract.

Once the candidate has been found, the variables *PriMem* and *SecMem* are updated to reflect the transfer of words from one CCS to the other, and the process returns to determine now *HowMany* must now be moved or retracted.

3.5.2 Terminate by Key Press

Just in case AttemptMove should enter an endless series of moves and retracts (see 3.5 Automatic Management), the procedure's loop has a check for a key press included. If the loop detects that a key has been pressed, then it will stop and prompt the user (with a call to PromptUser) to see if they would like to "Abandon the AUTO attempt?". If they respond yes, then the variables *Terminate* and *Done* are set to "true" so that the loop process of both AttemptMove and AutoManageMem will end.

3.5.3 Check Goal

As seen in Figure 4, the call to AttempMove is followed by a call to GetCurrentCount, which is eventually followed by a call to the CheckGoal procedure. As the name implies, this procedure checks to see if after doing the second (accurate) word count, the goal appears to have been achieved.

CheckGoal follows steps similar to those described in 3.5.1.1 How Many to Move. However, a unique important function it performs is to update several flags such as *GoalAchieved*, *Done*, *AutoOK*, *NeedToMove* and *NeedToRetract*. These various flags control the entry and exit to and from several loops in the automatic management routines. For example, note that in AutoManageMem, if either the flag *GoalAchieved* or *Done* is set, then the procedure StatusPrompt will be called to see what the user would like to do next.

As these flags are updated here, they are also updated by other procedures such as AttemptMove. Keep in mind, the automatic process is one where AutoManageMem constantly goes back and forth between GetCurrentCount and AttemptMove to try to reach the specified goal. In the process, AttemptMove may end with the impression that the goal was achieved. However, after GetCurrentCount is called again, CheckGoal may clear some flags that were just set in AttemptMove. This is because the accurate word count provided by GetCurrentCount is used to double check the work done in AttemptMove.

3.5.4 Status Prompt

As seen in Figure 4, following the call to CheckGoal, if it was determined that either the goal was achieved or that we are done (no more commands left to move, even though we need some), then the procedure StatusPrompt is called. StatusPrompt simply presents the user with the current word count once it is apparent that either the goal was reached, or it never will be.

StatusPrompt presents the count in the form of how many words over or under the limit we are in each processor. It gets these values by placing the difference between the available memory in each processor (variables *AvA* and *AvB*) and the actual memory (variables *MemA* and *MemB*) in the variables *PriDiff* and *SecDiff*. The variables *PriDiff* and *SecDiff* are then presented to the user.

Here, if the *AutoGoal* variable is JustUnder, and the *GoalAchieved* flag was set in the CheckGoal procedure, and the Partial flag is not set, then StatusPrompt will offer the user a choice of what to do next by calling the

procedure WhereNow (see 3.5.5 Where Now? for an explanation of the *Partial* flag and the WhereNow procedure).

If this is not the case, then if either of the two flags GoalAchieved or Partial are set, the procedure will simply state that the goal has been achieved, and it will prompt the user to press a key to continue on to the manual management.

If this however is not the case either, then the goal was never achieved (as determined by CheckGoal), and so all attempts to move commands must have been exhausted. Here the user will be notified of the problem, and offered a chance to repeat the automatic management. If the user chooses to repeat the management, then the flag *AutoOK* is set to "false" so that when the main loop in the MemMan procedure (Man_Main.PAS) takes control again, it will revert back to the automatic management screen (SelectMoveItems) rather than moving on to the manual management procedure MoveManual (Man_Manl.PAS).

3.5.5 Where Now?

The procedure WhereNow is only called when the variable *AutoGoal* is equal to JustUnder, (automatic goal to manage a CCS load which resulted in a word count that was just under the limit for the master), and the variable *GoalAchieved* is equal to "true" (goal was achieved). In this case, assuming that there is still some free memory in the secondary (slave) processor, the user has three options. They can either accept the management as it is, pursue a balanced load, or continue to move any specified number of commands from the primary CCS to the secondary (up to the number of free words in the secondary)

If the user chooses to accept the management as it is, the variable *Done* is set to "true" to cause the main loop in AutoManageMem to be terminated. If the user chooses to pursue a balanced memory load, the variable *Done* is set to "false", the variable *GoalAchieved* is set to "false", and the variable *AutoGoal* is changed from JustUnder to Balanced. This will assure that the main loop in AutoManageMem will not be terminated, in fact it will continue to loop until balanced memory is achieved, or until all of the possibilities have been exhausted.

The third option manipulates the variables *AvailA* and *AvailB* to cause further attempts at automatic management to in effect move a specified number of words from one processor to the other. For example, if the automatic goal is just under, and that goal had been achieved, one could then reduce the number of words available in the master and restart the automatic attempt. Because MEMMAN thinks that there are fewer words available in the primary CCS, it will attempt to move that many words to the secondary.

If the "M" (Move) option is chosen in the procedure WhereNow, GetInput (GW_Input.PAS TPU) is called to get a value in the variable *Move*, in order to find out how many words the user would like to transfer to the secondary. GetInput is passed a zero value and the variable *FreeW* (given a value in CheckGoal) as the input value limits (constraints). In other words, the user's input must be between zero and the number of free words in the secondary.

3.5.6 Show Messages

The last step taken in the automatic management is to display any messages gathered during the final GetCurrentCount by calling ShowMessages (Man_Misc.PAS TPU). All messages gathered during execution of the GetCurrentCount procedure are stored in the *MsgTrack* array so that they can be displayed at an appropriate time (not always immediately after GetCurrentCount).

3.6 Manual Management

As mentioned earlier in the introduction, the manual management screen is the final screen the user will visit if everything is going normally. At this screen the user has an opportunity to scroll through the entire sequence, viewing various characteristics of commands and memory management, and making final "manual" adjustments to the memory management.

The user can press several keys to perform specific actions including the arrow keys, the home key (top of SRF), the end key (end of SRF), the tab key (view a command cost), and many others. The keys which carry the most importance for this screen however are the <F2> "Select" key, the <F7> "Move Manually" key, the <F8> "Retract Manually" key, the <F9> "Get Word Count" key and the <F10> "Exit & Generate Output" key.

Figure 6 depicts the basic logic of the manual management routine MoveManual (Man_Manl.PAS TPU). Although the flow chart only shows two decisions being made after a key press, there are actually many decisions (keys) as described above, Figure 6 simply shows two examples of key press checks made.

As MoveManual allows the user to travel through the SRF, the Temp file (see 4.1 Temp File) is really the file which is being stepped through, as opposed to the input SRF. Each event (command) in the Temp file is checked for a match in the *Current* or *Overlap* arrays. If a match is found, then the screen will reflect any known characteristics of the command, otherwise the command will be darkened (grey) in appearance.

The screen has a left marker character (◂) on the right side of the screen which is used to point to a record (command) in the SRF. By using the up & down arrows, page-up & page-down, and the home & end keys the user can move through the sequence, pointing to

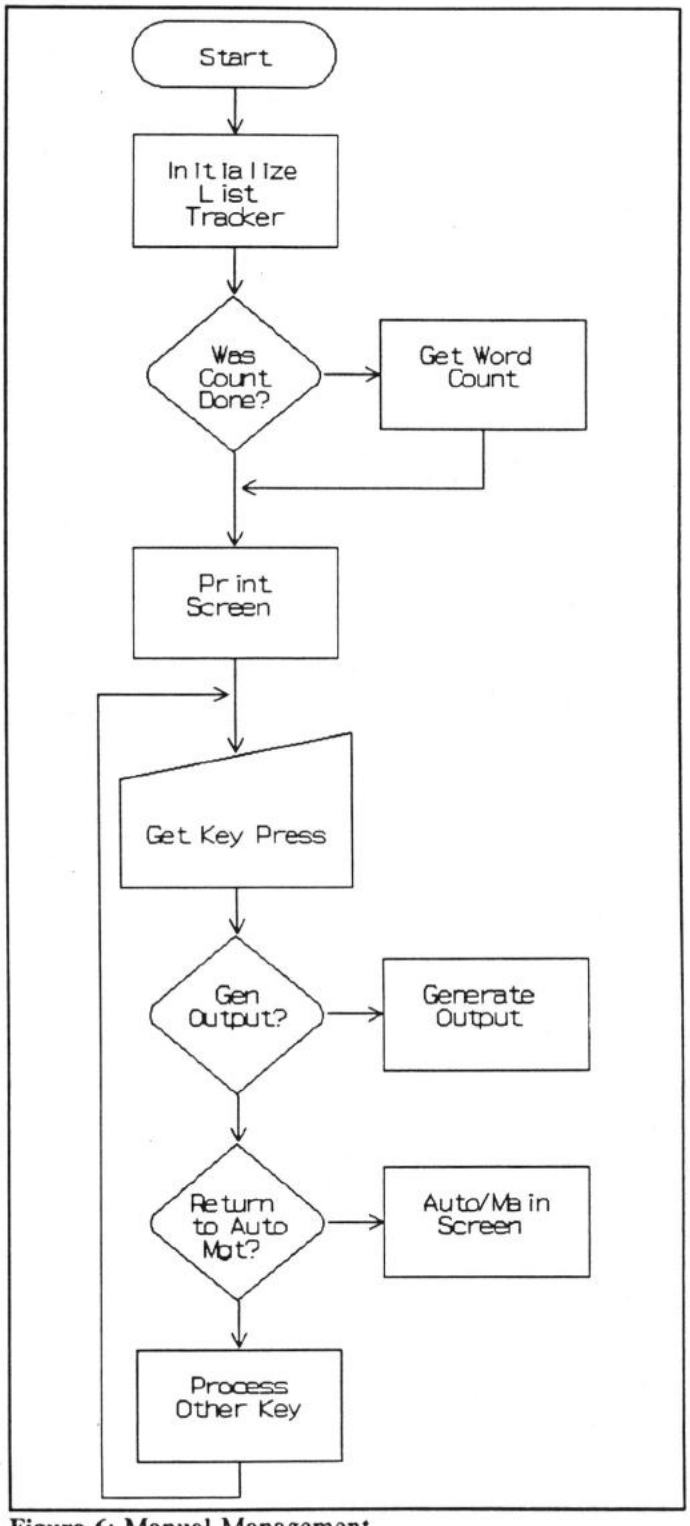

Figure 6: Manual Management

individual commands.

The user can point to a single command, or select a group of contiguous commands, and press a key to move them to the secondary CCS. They can then also press another key to retract commands (which have been manually moved) back to the primary. Once commands have been moved or retracted, the user can press a single key to get an updated word count (an accurate count done by GetCurrentCount).

The following sections describe several of the manual management functions, and attempts to give basic descriptions of how they work.

3.6.1 Manual Management Pointers

A very important concept to understand is how the pointers to the Temp file and the *Current & Overlap* arrays are positioned and maintained for movement throughout the SRF. In the manual management system, movement through the SRF is categorized as either forwards or backwards. In fact, all movement must be sequential command movement in either the forward or reverse direction, as opposed to random access to any command in the SRF.

The reason for only allowing sequential forward or reverse movement is so that pointers to the *Current, Overlap & Auto* arrays and the Temp file will be kept aligned. For every command in the Temp file which is relevant to the memory management, there is a matching *Current* or *Overlap* array entry. The Temp file contains the text for every needed SRF record, while the *Current* and *Overlap* arrays contain descriptions of relevant memory management characteristics for the matching Temp file records. For many of the records in the Temp file, there are also matching elements in the *Auto* array (see 3.3.2 Auto Array). As an added chore, the pointing to elements in the *Auto* array must also be kept aligned with the movement through the Temp file.

It is possible to have records in the Temp file which do not match an element in either the *Current* or *Overlap* array. Examples would include comment records (irrelevant to memory management), continuation records for a command and $$MESSAGE records. By the same token, there are elements in the *Current* array which will not have counterparts in the Temp file. The best examples are cyclic definitions. These definitions each occupy a single element in the *Current* array, but are never actually seen in the manual management screen, and are therefore not stored in the Temp file.

Since there is no random access to SRF records displayed in the manual management screen, large jumps over records (such as page-up, page-down, home, end & go to a line) must step sequentially through and commands that are essentially being skipped. For instance, if the user is at the top line of the screen, and they press the page-down key, then manual management will [transparent to the user] step through twenty-one commands to get to the desired one, the top of the next page.

The only exception to this rule is the ability to initialize array and Temp file pointers at the "top" or "bottom" of the SRF in order to speed access to commands which are far away (in the SRF). If the user is pointing to (or somewhere near) the start of the SRF and they press a key to go to (or somewhere near) the end of the SRF, before stepping through the whole SRF (time consuming) a check will be done to see if it would be closer to start from the end of the SRF and step backwards. If it's closer, the pointers are initialized to the end of the file, and are then stepped backwards until the target is reached. The same check (and process) is used for moving from near the end of the SRF to near the start.

3.6.2 The Track List

Another important manual management pointer maintenance function is to track where the user is presently pointing in the Auto array while traversing the Temp file (SRF). The track list is needed because although generally a certain command might be moveable, a certain instance of that command may not be moveable due to special CONFIGs in the SRF, or other constraining factors. We must correlate movement through the Temp file with the *Auto* array so that we will know if we are pointing at that one certain instance of a command which is not moveable.

For example, when looking at Figure 3, we can see that generally the DC2P commands are allowed to move. However, the third and fourth occurrences in the SRF have been explicitly configured to be in CCSB, hence they are not moveable. If this SRF were being viewed at the manual management screen, the user would be able to move the pointer (arrow on the screen) back and forth through the list of commands. Because not all of the DC2Ps are moveable, we must track the pointer (arrow) movement through the SRF so that when pointing to the third DC2P, we will have a match with the correct entry in the *Auto* array.

As can be seen in the Man_Vars.PAS TPU, the array *Track* is of type *ListTracker*. This record type has three fields; the command name (*Name*), the element of the Temp file currently being pointed to (*Element*), and the occurrence of a command in the SRF (*OccurNo*). As the user steps through (traverses) the SRF, each command is checked for an entry in the *Track* array (one entry in *Track* for every entry in *Auto*). If an entry is found, then the *Element* and *OccurNo* fields are updated to reflect the new position in the file.

A possible future enhancement could possibly be to combine the *Auto* and *Track* arrays into one array which serves both purposes. The tracking logic was added late in development, thus not originally designed to work as closely as might be hoped with the Auto array.

3.6.3 Fresh Screen

The FreshScreen procedure (Man_Misc.PAS TPU) is used to print a complete screen, starting with one particular record in the SRF (the Temp file). The procedure has two parameters; the start pointer (to the Temp file) and the direction (a *direction* type variable). The start pointer points to the desired first record on the new screen, where the direction flag tells which direction we need to go to get to that start record. These two local variable have the names *Start* and *Direc* (respectively) in the FreshScreen procedure.

The global variable *LastTempPtr* holds a pointer to the Temp file record which is presently being pointed to. Whenever the routine is called, *LastTempPtr* is compared with *Start* (pointer to the first command of the new screen) to see if there are any records which must be traversed to get to *Start*. For example, if the user was presently pointing at the last item on the screen, and they pressed the PgDn key to get to the next screen, there would be no need to traverse any records to get to the top of the next screen. However, if the user was pointing to the top (or somewhere in the middle) of a screen, and pressed the PgDn key, there would be a need to sequentially traverse the remaining records on the screen, in order to move all of the pointers to the first command on the next screen.

The first step in the FreshScreen procedure is a call to the FreshHeader procedure (Man_Misc.PAS TPU) to update all of the screen header information (such as unused words, overlap words, etc...). Next, the depending on the *Direc* flag, *LastTempPtr* and *Start* are compared to see if there is a need to traverse some commands to get to the start of the new screen. If so, then a short loop repeatedly calls the procedure GetRecordData (Man_Misc.PAS TPU) in order to advance (or retreat) the pointers to the record one before the start of the

new screen if traversing forwards, or one after the end of the new screen if traversing backwards, (see also 3.6.4 Get Record Data).

Once the pointers are positioned correctly, a second loop proceeds to print the displayed records on the screen. If the direction was in reverse, then this would proceed backwards from one past the last item on the screen, backward to the first item. If the direction was forward, then this would proceed from one before the first item on the screen, forward to the last item. In the forward moving case, the pointers would now be at the last item on the screen, so a third loop would then traverse backwards to the first item on the screen. Once back at the first item on the screen (for forward or reverse movement) a call to PrintMarker (Man_Misc.PAS TPU) is made to print the arrow symbol next to the command presently pointed to (the first command on the screen in this case).

For each item, the letter A or B is printed in the processor column of the manual screen, unless the event has some command overhead cost associated with it. If the command is the first in a new time-event region, then the *Overhead* field of the *Current* or *Overlap* array element will have a value of one. In this case, a paragraph character (¶) is printed in the appropriate processor column, rather than an A or a B. If the command is preceded by a switch to hours and back [to seconds], then the *Overhead* field of the *Current* or *Overlap* array element will have a value of two. In this case, a switched arrow character (↕) is printed in the appropriate processor column.

In addition, if a command is contained in the *Manual* array (if it is tagged to be moved manually) then a special character will be printed next to the processor column. For a manually moved single command, a right bracket character (]) will be printed. For a manually moved group of commands, an extended right bracket will be formed by using the ┐, | and ┘ characters. This extended right bracket will bracket all of the commands in the manually tagged group.

3.6.4 Get Record Data

One of the main manual management procedures is the GetRecordData procedure (Man_Misc.PAS TPU). The name of this procedure is slightly misleading, because the procedure is used to not only get information, but to adjust some array pointers as it finds that information. The procedure is called any time there is a need to move forward or backward from one command to the next (or previous) sequentially.

To begin with, GetRecordData will seek the file record specified by the *TempPtr* parameter to the procedure, and then read the record from the file. Note that this pointer should (in actuality) only be one record away from the last Temp file pointer (*LastTempPtr*). Next, a check is made to see if the direction is forward and if the matching command is in the tracking list (checked by a call to the InTrackList function, Man_Misc.PAS TPU). If the command is found in the tracking list (array *Track*), then the *OccurNo* field of the *Track* element is updated show that we are moving forward one event. Next, the procedure calls the function FindMatch to see if there is a match for this Temp file record in either the *Current* or the *Overlap* array (see also 3.6.5 Find Match).

If a match for this record was found, after checking which array the match was found in, several things happen. First, a check is made to see if the direction is reverse and if the matching command is in the tracking list. If the command is found in the tracking list (array *Track*), then the *OccurNo* field of the *Track* element is updated show that we are moving backwards by one event.

Next, the procedure GetMoveData (Man_Misc.PAS TPU) is called to determine whether the command is

moveable or not, (see 3.6.6 Get Move Data). The result is stored in the boolean variable *Move*. This is done for screen color choice and assistance in checking whether it is allowed to be moved or not). The status of the *Move* variable is logically AND'ed with the *ManMove* field of the matching *Current* (or *Overlap*) array element, and the last status variables *LastMove*, *LastCost* and *LastName* are updated to reflect the new command now pointed to.

If no match for this Temp file record was found, then a check is done to see if this current Temp file record is simply a continuation of another command. If this is a continuation record and the direction is forward, then the last known matching information (if any) is used to describe this command (continuation). If this is a continuation record and the direction is in reverse, then a special algorithm searches backwards until the command (for the continuation) is found, and the necessary information can be gathered.

If no match was found, and the new Temp file record is not a continuation record, then is simply marked as unmovable, and will later (in another procedure) be printed in a darkened (grey) color. This case usually applies to comments or TXFER macro commands, neither of which cost any CCS words.

3.6.5 Find Match

The FindMatch function (Man_Misc.PAS TPU) uses several very important variables including *CurPtr*, *OvrPtr*, *MasterArray* and *LocalType*. The variables *CurPtr* and *OvrPtr* are maintained throughout the manual management as pointers to the element which matches the present Temp file record (if there is a match). Both the *MasterArray* and *LocalType* variables are of the *EventType* type (Cur, Ovr, Neither or Force). *MasterArray* holds the type of the main array being pointed to presently. Remember that overlap events can be mixed within current events if looking at the SRF in time order. In other words, it is possible to be "temporarily" pointing to an overlap event which is surrounded by current events. In this case, the *MasterArray* would be Cur, but the *LocalType* would be Ovr. This usually occurs when a cyclic is forced into the overlap region, due to a single call after the next window open time. The calls to that cyclic would remain in time order in the SRF, possibly surrounded by current events, but they are really in the overlap region.

Although the logic of the FindMatch function can best be understood by studying the source code, a few comments will help in understanding the algorithm. Generally, while stepping forward through the Temp file (hence the *Current* and/or *Overlap* arrays) if the end of the *Current* array is reached, then the *Overlap* array would be searched next, as it generally follows in time order after the current commands. In the same sense, while stepping backwards through the Temp file (hence the *Current* and/or *Overlap* arrays) if the beginning of the *Overlap* array is reached, then the *Current* array would be searched next, as it generally precedes the *Overlap* array in time order.

Because commands can be time tagged to occur during what would normally be called current sequence time, but yet placed in the overlap area of memory, it is possible (as mentioned above) to have overlap commands interspersed throughout current events while viewing the SRF (Temp file) in the manual management screen. In this case, FindMatch will sometimes have to "jump" back and forth between the *Current* and *Overlap* arrays looking for a command that has been forced into the overlap region for one reason or another.

Generally, if the FindMatch routine notices (by checking time tags) that it has searched beyond where the matching events should have occurred in one array, then it will back-track to where it was and search the other array. Only after looking in both arrays will a search be given up. Incidently, among many other possible MEMMAN modifications, the Temp file could be modified to hold a pointer to either the of the two arrays, thus eliminating the needed for this searching algorithm.

3.6.6 Get Move Data

The procedure GetMoveData (Man_Misc.PAS TPU) is an integral part of the manual management code. GetMoveData is primarily called by the GetRecordData routine, in order to determine whether a command is moveable (in the manual management) or not. The GetMoveData returns a "true" move status based on three tests which must be passed; the command must not be constrained by previous automatic management actions, the command must not be marked unmovable in the manual management system, and the command must not be a cyclic call for a parallel cyclic (whether the particular call is in parallel or not).

The routine begins by initializing the *Move* flag (variable) to be "true" as a default. Any time a test fails, the flag is cleared (set to "false"). Next, three separate checks are made to ensure that the command is moveable. First, the boolean function InTrackList is called to see if the command is being tracked (see 3.6.2 The Track List). If the command is being tracked, then by using the present value of *CurPtr* or *OvrPtr* (depending on the which region of memory the command the user is presently pointing to is in - stored in the *LocalType* variable), the present *Current* or *Overlap* array element's *Proc* field is checked to make sure it is not set to SecProc (the slave processor). Such a setting would [of course] preclude any movement, since the command is already in the secondary processor.

Second, the *Element* field of the *Track* array record is used to point to the matching *Auto* array record. The *AType* field of the *Auto* array record is then checked to make sure the command presently pointed to is of type Use and not type Skip. If the command is of type Skip, then the *Move* flag is set to "false". Third, if the command is of type Use, the *OccurNo* field of the *Track* array record is compared with the *Occur* field of the corresponding Auto array record to make sure the command hasn't already been moved.

3.6.7 Pointer Movement Handling

The manual management system has in general two main pointers which must be maintained (moved) together as the user traverses the SRF. The two pointers are *TempPtr* and either *CurPtr* or *OvrPtr*. Each time the user chooses to move in either direction through the Temp file (the SRF), both the Temp file pointer and the pointer to the present assembly array (*Current* or *Overlap*) must be kept aligned. The following sections will describe (in general) how the pointers are maintained for all movement through the Temp file.

3.6.7.1 Up Arrow Key

The up arrow key press is facilitated by a call to the UpArrow procedure (Man_Curs.PAS TPU). When the up arrow is pressed, the first pointer action in the routine must be to make sure the user is not choosing to move beyond the start of the Temp file (the SRF). If the user is pointing to the first event in the sequence, and they press the up arrow, an error message will be issued.

Then, if the Select flag (see 3.6.11 Select a Group) is set and the user is presently pointing at the first command in the selected group, an error message is issued (only allowed to select forward). Otherwise, the Temp file pointer (*TempPtr*) is compared with the top of the page pointer (*PageTopPtr*) to see if they are the same. If they are the same, this indicates that the user is presently pointing to the first command on the screen. In this case, rather than just moving to the previous item on the same screen, we must move to the last item of the previous screen. This requires setting the *DesiredPtr* variable to the previous command (one less than *TempPtr*), calling the PageUp procedure (Man_Curs.PAS TPU) to position the marker (and pointers) at the first command of the previous screen, and finally calling the MoveMarker procedure (Man_Misc.PAS TPU) forward enough times to position the marker at the last item on the screen.

If the marker is not presently at the first command on the screen (*TempPtr* does not equal *PageTopPtr*), then a call is simply made to the MoveMarker procedure in order to move the marker (and pointers) backwards to the preceding item on the same screen.

Finally, if the *Select* flag is set, then the *Ptr* and *Line* fields of the *EndSel* record must be updated to reflect the fact that we moved backwards one command (un-selected a command from the presently selected group). If the *Select* flag was set and we were pointing to the first command in the group, this would have no effect.

3.6.7.2 Down Arrow Key

The processing of a down arrow key press is much the same as the up arrow processing, except that it is slightly more involved as it requires the possible handling of an extension of a presently selected group on to the next screen.

The first constraint check made by the DownArrow procedure (Man_Curs.PAS TPU) is to make sure the user isn't presently pointing at the last command of the sequence. This is obviously an error, and must be flagged as such.

Next, if the user is pointing to the last command on the screen, (and not the last command of the sequence), then generally the *DesiredPtr* variable will be set to the present value of *TempPtr* plus one (the next command), PageDown will be called (Man_Curs.PAS TPU) to print the next screen and position the marker & pointers at the first command on that screen, and finally MoveMarker will be called to position the marker on the screen.

If however the key press occurs while the user is presently pointed at the last item on a screen, and the *Select* flag is set, then several other steps must be taken to handle the transition to the next screen. First, GetRecordData (Man_Misc.PAS TPU) is called to get the move data for the next command. If it is not moveable, then it can not be selected (see 3.6.11 Select a Group). Here, GetRecordData will have to be called a second time in the opposite direction (reverse) to correct the pointer positions.

If the first command on the next screen is moveable, then GetRecordData will be called again to move the pointers back to the last item of the previous screen. Remember, it [GetRecordData] was called once to get data for the first item of the next screen, and must be called again to set the pointers at the correct location for a call to PageDown. After the "correcting" call to GetRecordData, PageDown is called (Man_Curs.PAS TPU) to position the marker & pointers at the first command of the next screen. Finally, the *Line* and *Ptr* fields of the *EndSel* record will be updated to extend the range of the selection.

3.6.7.3 PgUp & PgDn Keys

Both the PageUp and PageDown procedures (Man_Curs.PAS TPU) operate in much the same fashion. In each case, *PageTopPtr* (the pointer to the first Temp record at the top of a page) is adjusted by the number of lines per screen (constant *PageLen*). Once *PageTopPtr* has been determined, its value is compared with the first (variable *StartTemp*) and last (variable *TempCount*) valid record in the Temp file.

In the case of the PgUp key, if the new *PageTopPtr* is less than the *StartTemp* value, then a call is made to GoTopSRF, otherwise a call is made to FreshScreen (with *PageTopPtr*, in the reverse mode). In the case of the PgDn key, if the new *PageTopPtr* is more than the *TempCount* value, then a call is made to GoBottomSRF, otherwise a call is made to FreshScreen (with *PageTopPtr*, in the forward mode).

3.6.7.4 Home & End Keys

Both the Home and End keys can be used to move quickly to the start or end of the SRF (Temp file). After checking the *Select* flag to make sure the user is not in the middle of selecting items to move, each key press would execute a call to the GoTopSRF or GoBottomSRF, respectively (Man_Curs.PAS TPU).

The procedures GoTopSRF and GoBottomSRF both begin by calling the InitListTracker procedure (Man_Curs.PAS TPU) with a parameter of either HighEnd or LowEnd. This procedure sets up the pointers (variables) needed to track movement through the *Auto* array.

Next, in both cases the *LastCurTempPtr* and *LastOvrTempPtr* pointers are set to negative one, to signal that they have no real meaning. Then, *CurPtr* and *OvrPtr* are set to values one less than the start or one more than the end of the respective *Current* or *Overlap* arrays, depending on whether you are looking at GoTopSRF or GoBottomSRF.

Next, the *MasterArray* variable is set to Cur in the GoTopSRF procedure, and Ovr in GoBottomSRF. Finally, *PageTopPtr* is updated with an appropriate value, and FreshScreen is called to display the new screen.

3.6.7.5 Go To a Line

By pressing the <F5> key anywhere on the manual management screen, the user can choose to move to any line (by number) in the Temp file. Because the present line number is always displayed at the top of the manual management screen, the user can easily remember the line number of a specific command. Then, they can return to that command after moving around the rest of the SRF (Temp file). This is handled by a call to the procedure GoToLine (Man_Curs.PAS TPU).

The first step taken in GoToLine is to call the TP procedure Window in order to restore the original 80 character by 25 character screen size (reduced normally during manual management). Next, minimum and maximum values for the line are determined, to be used with the GetInput procedure. It is important that the user not be allowed to specify a line number which is beyond either the beginning or the end of the file.

Next, the variable *DesiredPtr* is assigned a value which corresponds to the line presently pointed to by the user. Then, a small loop proceeds to prompt the user for input by calling the GetInput procedure (GW_Input.PAS TPU). After each prompted input, the user's response is adjusted to compensate for entries in the file which are prior to the "visual" beginning of the file. The first "visual" record in the file is marked by the variable *StartCurrent*, so *DesiredPtr* is adjusted with *StartCurrent* each pass through the loop.

If during the loop it is discovered that the user has chosen a line which is currently displayed on the screen, they will get a message as such. Otherwise, the loop repeats until either the user enters a number between the minimum and maximum values determined earlier (start and end of the Temp file), or until the user presses the escape key (which reverts *PageTopPtr* to its original value).

Finally the TP procedure Window is called again to re-size the screen (allow for undisturbed header on screen), and then the procedure TempSeek (Man_Curs.PAS TPU) is called to seek the desired line number, and then reprint the screen with the marker positioned at the desired line.

3.6.8 Move a Command/Group

If the user presses the <F7> key while pointing to a single moveable command, or a selected group of moveable commands (see 3.6.11 Select a Group), that command (or group of commands) will be "moved" from the primary processor (master) to the secondary (slave). This process involves not only adjusting arrays such as *Current* and *Overlap* to specify the new processor, but also adding a description of the command (group) to the *Manual* array. This array will then be used during the output generation (see 3.6.17 Generate Output) routines to either add CONFIG macro commands directly to the SRF, or to the end of an SRF editor runstream.

Because the command movement process is so involved, this section will cover a slightly higher level of discussion than might be desired. For a deeper explanation, see the MoveManually procedure source code (Man_Manl.PAS TPU).

To begin with, once the <F7> key is pressed, the main manual management procedure (MoveManual) calls MoveManually (also in Man_Manl.PAS TPU). Move Manually begins by setting a flag (variable) called *Ok* to "true" to begin with. Throughout a series of checks, the flag will be cleared if a problem is encountered.

Next, so that group and command movement can be handled in a similar fashion, and single commands are converted to a group with one entry. If the *Select* flag is not set, the StartSel and EndSel records are updated with information for the single command, thus converting it into a command group format.

Next, the first item in the group (only item if this is a single command) is "seeked" and read from the Temp file, in order to check several constraints. The starting record is checked to make sure that it is not a comment or a record which is a continuation of a prior command. Next, the record is checked to make sure that it hasn't already been marked to move (manually). Next, the record is checked to make sure it is not a comment (same check done earlier when move target is not a group of commands).

After some other miscellaneous constraint checks, a call is made to the procedure AddToManualList (Man_Misc.PAS TPU), in order to add a new entry to the Manual array, and bump the counter *ManualCount*. Next, if the group is that of a single command, and that single command is a cyclic call, then a call is made to the MoveCalls procedure (see 3.6.10 Move/Retract a Cyclic) in order to move every call to that cyclic in the whole sequence. While the select flag is set, cyclics are not allowed to be moved, and the user will be instructed to use <F7> alone (move) instead.

Finally, by using the MoveLine and PrintLine procedures (Man_Manl.PAS TPU) the command(s) is (are) reprinted on the screen with their new status (slave processor, non-moveable). In addition, the PrintLine procedure prints some special characters next to the command(s) to show that they have been moved manually (see also 3.6.3 Fresh Screen).

If a cyclic was just moved, then the procedure ReprintScreen is called in order to correct any other calls to the same cyclic which are on the present screen. Then a special pop-up window is displayed to inform the user that all other calls to that cyclic have also been moved.

3.6.9 Retract a Command/Group

As is the case when moving a command, the command retraction process is so involved that this section will cover a slightly higher level of discussion than might be desired. For a deeper explanation, see the RetractManually procedure source code (Man_Manl.PAS TPU).

To begin with, the RetractManually procedure will check to see if the *Select* flag is set. It is not possible to select a group of commands, and then to retract them. (Instead, because commands that were moved together stay together in a group, any member of that group can be pointed to, and then the <F8> can be pressed. This action will retract all members of the original group.) At any rate, if the *Select* flag is "true", then a message (window) will be displayed, and the routine will be exited with no further action.

Next, the routine checks the item presently pointed to, to make sure it is in the *Manual* array (list). If the item is not in the list, then a message (window) is displayed, and again the routine will be exited without any further action.

If the command pointed to does appear in the *Manual* array, the next step is to retrieve the *StartSel*, *EndSel* and *NextCount* (number of continuation records for the last command in the group) values from the Manual array element. Then the procedure RemoveFromManualList is called with the pointer to the *Manual* array element containing the information about the present command or group. This procedure removes the element and updates the *ManualCount* variable.

If the command pointed to is a cyclic call, then a call is made to the RetractCalls procedure (Man_Cycl.PAS TPU) in order to retract every call to the same cyclic in the sequence. In addition, the records *StartSel* and *EndSel* will be updated to point to this particular cyclic call (some other call to the same cyclic might have been pointed to in order to move the cyclic).

Finally, by using the MoveLine and PrintLine procedures (Man_Manl.PAS TPU) the command(s) are reprinted on the screen with their new status (master processor, moveable). In addition, the PrintLine procedure covers-up any special characters next to the command(s), characters which were printed in order to show that the command had been moved manually.

If a cyclic was just retracted, then the procedure ReprintScreen is called in order to correct any other calls to the same cyclic which are on the present screen. Then a special pop-up window is displayed to inform the user that all other calls to that cyclic have also been retracted.

3.6.10 Move/Retract a Cyclic

In order to move or retract a cyclic, the user must point to a single call to that cyclic, and then press the appropriate key to move or retract the command, (if the Select flag is set, see 3.6.11 Select a Group, then neither the move or retract keys will work).

The moving and retracting of cyclics is implemented by calling MoveCalls and RetractCalls (Man_Cycl.PAS TPU). Each of these procedures function in basically the same way; they must check both the *Current* and *Overlap* arrays, and they must make sure they are not interfering with cyclics (or calls) which are in parallel. As these tasks are fairly simple, the procedures MoveCalls and RetractCalls are fairly straight forward and can be seen in their TP unit, Man_Cycl.PAS.

3.6.11 Select a Group

In addition to moving a single command to the secondary processor, the user is able to move a group of moveable commands, all at once. This is accomplished by selecting several moveable commands together (by pressing <F2> and the down arrow key), and then pressing the <F7> (move) key.

When the <F2> key is pressed, MoveManual calls the SelectItems procedure (also in the Man_Manl.PAS TPU) to toggle the state of the Select flag, and do some needed setup for the selection process. The procedure will generally follow two distinct paths; one if the *Select* flag is not already set (not in the process of selecting) and the other if the *Select* flag is presently set (presently selecting). The first case describes what generally would take place most of the time. The second case will only happen when the user is in the process of selecting some commands, and suddenly decides to undo the selection.

To begin with, we will explore the instance where the flag has not yet been set. First, the command presently pointed to is checked to make sure that it is moveable (the only reason to select an item is to move it), and that it is not a cyclic call (cyclic calls cannot be selected to move, they must be moved alone). If both of these tests are passed, then the selection process begins.

First of all in the selection process, the *Select* flag must be set (to a value of "true"). This global flag will remain set until either the select key or the move key is pressed. Next, the TP Window procedure is called to restore the full screen size, and the text "SELECT" is printed at the top of the screen in a flashing color, in the spot where the text "CHKPNT SET 0" message is normally displayed. After the screen size is reduced again (by another call to the TP procedure Window), the StartSel and EndSel global records (variables) are set to values which correspond to the line presently being pointed to, and PrintMarker is called (Man_Misc.PAS TPU) to reprint the marker in a special color.

If the Select flag was not already set, and the command presently pointed to is a cyclic, a pop-up window is displayed which instructs the user to use the <F7> (move) key alone in order to move a cyclic. If the Select flag was not already set, and the command presently pointed to is not moveable, then a pop-up window is displayed which informs the user that the command can not be selected. Remember, the only reason to select commands is to move them, commands do not have to be selected to retract them.

Now we will explore the instance where the *Select* flag was set at the time of the call to SelectItems. First of all, the *Select* flag must be cleared (set to a value of "false"). Next, the text "SELECT" must be cleared from the top of the screen, and the text "CHKPNT SET 0" will be restored (if the *MainChkSet* flag is set). Finally, for any commands presently on the screen which were a part of the selected group, we must reprint the space next to them where the marker would normally appear, (this space was previously filled with a different color for each selected item.)

As far as adding commands to the selected group, this is handled by using the down arrow key while the *Select* flag is set. In other words, to select a group of commands, the user would position the marker next to the first command in the desired group, press <F2> (the select key), and then repeatedly press the down arrow until all of the desired commands were included in the group. At that point, the user could press the <F7> (move) key to move the whole group. The advantage to grouping commands together in this fashion is that this results in instructions for only a single pair of CONFIG macros commands. This is opposed to selecting several commands in a row, each individually, which would result in instructions for a single pair of CONFIG macros for each command.

3.6.12 Checkpoint Set 0

When a CHKPNT macro is encountered in the SRF, SEQTRAN sets a symbol called CHKPNT to a value of one. From that point on in the SRF, if the CHKPNT flag is set to a value of one and an excessive time gap appears, SEQTRAN will assemble commands to switch to hours mode and then back to seconds, at a cost of four CCS words. If the CHKPNT symbol is not set to a value of one and an excessive time gap appears,

then SEQTRAN will assemble the commands to start a new time-event region, at a cost of only three CCS words.

Although the second method (commands for a new time-event region) would seem preferable, the situation is handled as it is so that an excessive time gap will not break-up a checkpointed region of commands. Once past the checkpointed region of commands, creating a new time-event region is generally of no consequence. In fact, it is usually preferable because of its lower word cost.

The manual management system provides the capability to add a specialized SEQTRAN symbol command (CHKPNT SET 0) to the SRF immediately following every BRKPNT macro. This will clear the CHKPNT symbol so that seqtran will assemble commands to add a new time-event region as opposed to switching time modes when an excessive time gap is encountered. This will save one CCS word for every occurrence of a time gap after a BRKPNT macro (before the next CHKPNT macro).

A very specialized method has been incorporated into MEMMAN in order to add this SEQTRAN symbol command in the needed places in the SRF. In the manual management system, the <F6> key has been designated as the CHKPNT SET 0 option key. The user can press this key to toggle the status of this option (on or off). MEMMAN uses a flag (boolean variable) called *MainChkSet* to maintain the status of this option. When this key is pressed, the only action taken in MEMMAN is that the MoveManual procedure (Man_Manl.PAS TPU) checks the present state, toggles the flag to the opposite state, and then prints a "CHKPNT SET 0" status message at the top of the manual screen.

This *MainChkSet* flag is then used by the GetCurrentCount procedure (Man_Gcnt.PAS TPU). If the procedure CheckForTE (Man_Gcnt.PAS TPU, called by GetCurrentCount) detects an excessive time gap, then depending on the status of the *MainChkSet* flag it will add either four CCS words (if MainChkSet is "false") or three CCS words (if MainChkSet is "true"). This is of course as long as the routine is not processing an active checkpoint region, in which case it will always add four words (as SEQTRAN would). See also 3.4.4 Check Time Gap.

When the main output for the run is being produced (either the new SRF or the runstream), the *MainChkSet* flag is checked to see if the string "CHKPNT SET 0" should be placed in the SRF after every BRKPNT macro call. If so, then either the instructions would be added to the runstream to facilitate this or the commands would be directly added to the new SRF (depending on the status of the variable *OutputFmt*). See also 3.6.17 Generate Output and 3.2.7 Output File/Format.

3.6.13 View Command Cost

Another feature added to the manual management system is one which allows the user to point to any command, press the Tab key, and be informed of that command's word cost (without overhead). This is needed because the word costs are not normally displayed on the screen, and such information might be helpful to the user in choosing commands to move manually.

When the user presses the Tab key, MoveManual calls the procedure ViewCommandCost (Man_Manl.PAS TPU). The first task of ViewCommandCost is to check the *Match* flag (variable) which is set as the command is pointed at while traversing the SRF (see 3.6.5 Find Match). If this flag is not set (if it is equal to "false") then there is no match for the presently pointed to Temp file record in either the *Current* or the *Overlap* array. If this is the case, then there is no way to determine the command cost, and so the user must be notified as such.

Next, if the *Match* flag is set (equal to "true"), a case statement is used to check the variable *LocalType* to determine whether the command is in the *Current* or the *Overlap* array (it is in one or the other since the *Match* flag is set). Finally, the *Cost* field of either the *Current* or the *Overlap* array is used with a call to the NumStr function (GW_Lib.PAS TPU) to display the command name & cost in a pop-up window.

3.6.14 Annotated SRF

An additional helpful option available to the user is the ability to print out an "annotated SRF" to assist with getting an overall picture of the sequence, and the present distribution of CCS words between processors. The printout generally supplies the user with the same information they would get from the manual management screen, however it is in a continuous printed format.

If the user presses the <F4> key, the MoveManual procedure calls the PrintSRF procedure (also found in the Man_Manl.PAS TPU). At the present time, the PrintSRF procedure has been "hard-coded" to be used with the EPSON LQ1500 printer (in the CCS area of 264-535), connected through the Logical Connection peripheral sharing device. Because of this, there are a few special statements included in the procedure which will be discussed in the next few paragraphs.

The PrintSRF procedure begins (generally) by calling the GoTopSrf procedure (Man_Curs.PAS TPU) in order to align all of the file and array pointers at the start of the SRF. Next, the procedure prints (sends to the printer) two strings which are unique to using the Epson LQ1500 and the Logical Connection (as described above). First, it sends the string ~LQ1500 to instruct the Logical Connection to connect to the printer named LQ1500 (the Epson LQ1500 in the CCS area in this case). Next, it sends the string chr(27)'x'chr(0)chr(15) in order to place the LQ1500 in a draft quality (condensed) character mode. This second string provides a faster printing of the SRF (faster than in the default mode).

Next, the procedure PrintSRF prints a summary page for the annotated SRF. This includes the present word count as stored in the variables *MemA*, *AvailA* and *DiffA* (same for CCSB). It should be noted that the word count stored in these variables will not be not accurate if any commands were manually moved since the last word count was obtained (see 3.6.16 Update the Word Count).

Next, the PrintSRF procedure assigns a value to the counter variable *i* so that it points one item before the first in the sequence (*StartTemp* minus one). Once the variable *i* is set up correctly, the procedure loops through the entire Temp file, printing the SRF by calling the PrintLineLst procedure (Man_Manl.PAS TPU).

The final commands sent to the printer are again unique to the Epson LQ1500. First, a chr(12) is sent to the printer to provide a form feed, and then a chr(27) in order to reset (initialize) the printer, restoring it to the default mode.

3.6.15 String Search

An additional option available to the user is the ability to search through the entire Temp file (after the SEQGO macro) for a particular string of characters. Those characters could be alphabetic or numeric, and could be time tags, commands or even parameters. The user can search for virtually any string of characters which appears after the SEQGO macro in the SRF.

When the user presses the <Alt><S> keys (together) at the manual management screen, the procedure Search (Man_Srch.PAS TPU) is called with the parameter *WhatKind* equal to NewSearch (see *SearchStrType*

in Man_Vars.PAS TPU). This procedure will then proceed to look (in a forward direction) through the Temp file for the next occurrence of the string *SearchStr*. If the user presses <Alt><R> instead of <Alt><S>, the procedure is called with the parameter *WhatKind* equal to RepeatSearch, and the procedure repeats the last search using the previous value of *SearchStr* (if there is one).

The procedure begins by saving the byte variable *TextAttr* (screen color text attribute) in the variable *OldAttr* (*TextAttr* is destroyed during the search process). Next, it checks the parameter *WhatKind* to see if it is being called for a NewSearch. If so, the procedure pops-up a window, prompts the user for a string of characters (which it stores in the variable *SearchStr*) and then closes the window.

Next, it checks for the condition where the call is for a RepeatSearch but the variable SearchStr is empty. This condition will occur if the user presses <Alt><R> (repeat search) without previously pressing <Alt><S> (search).

If the variable *SearchStr* is not empty, the routine saves copies of all of the critical manual management variables (just in case the string is not found). Then it proceeds to loop through the Temp file (from the present position) checking each record of the file for an occurrence of the character string *SearchStr*. If the string is found, the variable *DesiredPtr* is set, and the routine TempSeek is called to display the newly found record. If the command is not found, all of the variables which were saved earlier are restored to their original values, and the user is notified (via a pop-up window) that the string could not be found.

Finally, the text attribute variable *TextAttr* is restored, and HideCursor is called to remove the blinking cursor from view.

3.6.16 Update the Word Count

While moving and retracting commands during manual management, the sequence word count totals (displayed in the header of the manual management screen) are not automatically updated. When the user is ready, they can update these totals by pressing the <F9> key to re-calculate the word count.

When the user presses the <F9> key, MoveManual (Man_Manl.PAS TPU) first checks to see if the *Select* flag is presently set (see 3.6.11 Select a Group). If the *Select* flag is set, then a pop-up window is displayed with a message telling the user that they cannot calculate while they are selecting. Note that it is technically possible to do so, but not allowed anyway.

Next, the ClearMarker procedure is called (Man_Misc.PAS TPU) to clear the marker in its present position, and the procedure GetCurrentCount (Man_Gcnt.PAS TPU) is called to update the word count variables (see also 3.4 Get Word Count).

Once the word count variables have been updated, the procedure FreshHeader is called (Man_Misc.PAS TPU) in order to update the printed values in the screen header. Then the procedure ReprintScreen is called (also in Man_Misc.PAS TPU) so that any overhead markers will be updated, and the MoveMarker procedure (also found the in Man_Misc.PAS TPU) is called (with zero lines as a parameter) in order reprint the line marker.

Finally, after the variables and the screen are updated, the procedure ShowMessages is called (Man_Misc.PAS TPU) in order to notify the user of any problems encountered while determining the word count.

3.6.17 Generate Output

To generate output (either in the form of a new SRF or an editor runstream) is the main goal of the memory management program. As can be seen in Figure 2, it is the final step in memory management. When the user is satisfied with the combined results of the automatic and the manual management, they would press the <F10> key (from the manual management screen) to generate the output and exit the program.

When the user finally does press the <F10> key, several constraint checks are made to validate the user's choice to quit. First of all, MoveManual (Man_Manl.PAS TPU) checks to see if the *Select* flag is presently set (see 3.6.11 Select a Group). If so, it will not allow the user to proceed with the desired option. Next, the user is prompted (via a pop-up window) to confirm their desire to quit and generate a runstream. If the quit is confirmed, the variables *AutoCount* and *ManualCount* are then checked to assure that they aren't zero. If they are both equal to zero, then no output can be generated (it would be empty). In this case, again the user would be notified, and they would then be returned to the manual management screen.

Finally, if all of the above conditions are met, the flag (variable) *CompleteQuit* is then set to a value of "true" so that the GenerateOutput procedure (Man_Gen.PAS TPU) will be called, and the main loop in MemMan (the main program code, found in Man_Main.PAS TPU) will be exited.

The GenerateOutput procedure begins by calling the MakeLogEntry procedure (Man_Log.PAS TPU) in order to place an entry in the log file (see 4.7.1 Update the Log File). Next, depending on the state of the variable OutputFmt it calls either MakeRunstream or MakeSRF (both found in the Man_Gen.PAS TPU) in order to actually generate the output file. Next, GenerateOutput calls PrintHistory (Man_Hist.PAS TPU) in order to print a complete report on the memory management run, and settings needed to reproduce that run. Finally, GenerateOutput calls the UpdateSeqConfigFile procedure (Man_Gen.PAS TPU) to update the sequence's configuration file with any new information (see 4.5 Configuration FIle for more about the information stored in the configuration file).

In general, the procedure MakeRunstream will read through the *Auto* and *Manual* arrays, and create a Univac 1100 editor runstream which can be used to edit the original (input) SRF, placing the needed CONFIG macro calls in order to manage the sequence's memory. The MakeSRF procedure is similar, except that it reads through the input SRF, echoing records read to the new output SRF, while simultaneously adding CONFIG macro commands to the new SRF in the proper places (stipulated by the *Auto* and *Manual* array).

4. Associated Files

4.1 Temp File

The Temp file is a structured file consisting of ASCII records. The file is created and assembled during the reading of the SRF, (see 3.3.3 Temp File Creation). The file is almost an exact duplicate of the SRF in content, but is formatted slightly different. Also, much of the data included in the SRF is omitted in the Temp file for the sake of speed when scrolling through SRF in the manual screen. In fact, the sole reason for having the Temp file is to have the SRF data in a more desirable format.

The filename contains the SRF version as part of its name, so that even though it is not necessary, every run has its own Temp file. For example, a run with the SRF version of B005F would create a Temp file called B005F.TMP, which would be used during the run, and then deleted upon termination of the program.

4.2 System Parameter File

The system parameter file is an ASCII text file (MEMMAN.SYS) which contains several variable names, and their default values (see page 41, A.1 Sample System Parameter File). The file is read at the start of MemMan near the end of the Initialize procedure in Man_Init.PAS, (which is an include file in the Man_Main.PAS TPU).

As can be seen in A.1 Sample System Parameter File, each variable is represented by its ASCII name. Any default values must be enclosed in quotation marks. Notice that for each variable name in the system parameter file, there is a corresponding MEMMAN variable by that exact name in the Man_Vars.PAS TPU. For each variable shown in the sample file, there is some code in the procedure ReadSysParFile (Man_Sys.PAS TPU) which takes the specified value from the system parameter file and assigns it to the actual variable (by the same name).

The system parameter file can be edited by any simple text editor, but the only items that can be changed are the parameters for the variables themselves. Any of the variable name & parameter pairs may be removed from the file, but none (in addition to those shown in the sample) may be added without adding a case to handle that new variable in the procedure ReadSysParFile (see also 4.2.1 Reading the System Parameter File).

4.2.1 Reading the System Parameter File

The system parameter file is read only once at the start of MemMan, near the end of the Initialize procedure in Man_Init.PAS, (which is an include file in the Man_Main.PAS TPU). The procedure ReadSysParFile simply opens the system parameter file, checks for the string "$$SYSPAR" at the start of the file & "$$EOF" at the end of the file, and the reads every other record attempting to parse each one.

For each allowed variable in the file, there is a corresponding IF-THEN clause to see if the record just read from the file was one which contains a particular variable and value. If the record read from the file pertains to a particular IF-THEN clause, then generally the variable *VarOK* is set to true, and the procedure GetQuoteVar (Man_Misc.PAS TPU) is called with several parameters including a pointer to the target variable, a variable type identifier constant (such as IntVar, declared in GW_Lib.PAS), the record text (buffer), the file name, and the record number (in case of an error while attempting to obtain the quoted value).

The only unusual case (at the present time) is for the variable *OutputFmt*. Here some extra checking must

be done to assure that the value given in the system parameter file is either FmtSRF or FmtRS.

4.3 Category File

The category file allows the user to create up to 15 groups of commands, or categories. Any categories listed in this file will then appear on the main user input screen, allowing the user to characterize each category as moveable or non-moveable (see page 42, A.2 Sample Category File). There are three main routines which relate to the category file; ReadCatFile, CatCanMove and SetCategory (all found in the Man_Cat.PAS TPU).

The procedure ReadCatFile is normally called only once at the start of the procedure SelectMoveItems (Man_Scr1.PAS TPU). If the category counter variable (*CatCount*) is equal to zero, then the procedure is called. The procedure CatCanMove is called by the LoadArray procedure (Man_Manl.PAS TPU) to see if a command's movement is constrained by a category status. The procedure SetCategory is called only in rare instances by the procedures ProcessCommand (Man_Pcmd.PAS) and LoadArray (Man_Load.PAS TPU) when it is determined (by program logic) that a category should no longer be moveable. In this case, the procedure is passed the name of the category along with the desired state (normally false) and the category will no longer be moveable.

The category entry format in the file is as follows:

```
CATEGORY "Category Name":Moveable
MASK "Mask 1"
(command)
(command)
(command)
MASK "Mask n"
```

Note the colon (:) after the category name, prior to the Moveable field. Also, upper and lower case letters are not differentiated between (varied here simply to help discern between fields). The following several paragraphs will attempt to explain each parameter.

`"Category Name"`

The category name is the string of characters which will appear on the main user input screen, representing all commands containing any of the Masks immediately following the Category Name. The maximum length of a Command Name is *CatNameLen* characters, and there is a maximum of *MaxCats* categories allowed (see the Man_Vars.PAS TPU). The category name must be surrounded with quotes.

`Moveable`

The Moveable field is a default (only a default), specifying whether or not this category is allowed to move freely during automatic memory management. There are only two possible values for this field, TRUE or FALSE. Any other characters after the colon (:) will cause a fatal error. The Moveable field should NOT have quotes surrounding it.

`Mask`

The Mask entries specify strings of characters which should be compared throughout each command record

(read from the SRF) to see if that command belongs to the associated category.

For example a mask of "DC2P" would match the following SRF record:

```
PROG    90-300/12:00:00:.945,DC2P
```

A mask of "DC2PR" would not match the above SRF record.

Each Mask entry has a maximum length of *MaskLen* characters (see the Man_Vars.PAS TPU), and must be surrounded by quotes. The number of Mask entries for each category is limited only by the system memory available at run time, as they are allocated dynamically as they are needed.

4.4 SMD File (SMDF)

This is the SEQTRAN Macro Description File (SMDF), and is called "MEMMAN.SMD". Each non-comment record in this file describes a SEQTRAN macro with information which is needed for memory management (see page 43, A.3 Sample SMD File). This file can only be used to describe "general" SEQTRAN macros which might appear in the SRF. Any "complicated" commands requiring special parsing or decision making will have to be added to the procedure ProcessCommand (Man_Pcmd.PAS TPU).

The procedure ReadSMDF (Man_Smdf.PAS TPU) is used to initially read the SEQTRAN Macro Description file (SMDF), and place the contents in a linked list of dynamic variables of type *SMDtype*. Later, in the ProcessCommand procedure (Man_Pcmd.PAS TPU), the boolean function InSMDF is called as a last resort if the command wasn't recognized (the command might be included in the SMD file). The function InSMDF returns a function value of either true or false. If true, the variable parameter points to the dynamic variable in the SMD linked list which describes the command.

Description records must contain each of the following parameters:

```
MACRO_NAME,WORD_COST,PROCESSOR,AUTO_MOVE,MANUAL_MOVE
```

Parameter descriptions:

MACRO_NAME — This parameter is a string of characters enclosed in double quote signs ("). This string reflects the actual SEQTRAN macro name, found in column 8 of the SRF. This is a required parameter.

e.g. `"BR01GO"`

WORD_COST — This parameter reflects the CCS word cost for this command description. The amount should be for one processor only, with the ability to specify the processor as "BOTH" using the PROCESSOR parameter. If you know that the macro is to be assembled in both processors (parallel), and that the cost for the two CCS processors is different, then this command CANNOT be added to the SMDF, it must be specially added to the ProcessCommand procedure (MAN_PCMD.PAS).

e.g. 2 (Use 0 if there is no word cost)

PROCESSOR — The CCS processor where MEMMAN will cost this command. Note that this parameter does not control where SEQTRAN actually assembles the command, but only where MEMMAN "thinks" SEQTRAN will assemble it. If the parameter is EITHER, MEMMAN will cost the command in the current processor (based on CONFIGs in the SRF, or the master processor if there haven't been any CONFIGs.

Valid PROCESSOR values:

PROCA	CCS Processor A
PROCB	CCS Processor B
BOTH	Parallel
MASTER	Master CCS (Chosen from a MEMMAN input screen)
SLAVE	Slave CCS (Opposite processor)
UNDEFINED	Neither processor (command has no MEMMAN impact)
EITHER	Follow current assembly path (processor)

AUTO_MOVE — This is a boolean parameter which governs whether this command will be allowed to move during the automatic memory management. The parameter can be either TRUE (allowed to move automatically) or FALSE (not allowed). However, if the PROCESSOR parameter is BOTH or SLAVE, then AUTO_MOVE will automatically be FALSE, regardless of the value specified in the file.

MANUAL_MOVE — This parameter is similar to the AUTO_MOVE parameter, except that it controls movement during the manual management of a sequence. The same constraints regarding the PROCESSOR parameter with the AUTO_MOVE parameter apply here.

4.4.1 Reading the SMDF

The SMD file is read by a call to ReadSMDF (Man_Smdf.PAS TPU). It is read in much the same fashion as similar ASCII MEMMAN files (e.g. system parameter file, force file, etc...), however, the method of storing the file contents in internal memory is very different. For the SMDF, all of the descriptions are stored in a dynamic linked list. The list elements are created [dynamically] using the TP procedure New as they are needed.

When the first element of the dynamic linked list is created, its pointer value (address) is stored in the global variable *FirstSMD*. For subsequent new elements, the previous element's *Next* field is always set to the address of the new element (otherwise this *Next* field is initialized to a value of zero, or nil). With this setup, the list can be traversed by starting with the element stored at the address contained in *FirstSMD* and continuously pointing to the next element by using the *Next* field, until a *Next* field is encountered with a value of nil (the last element in the list).

4.5 Configuration File

There is (should be) a configuration file for each new version of every SRF (for each sequence). The file contains all of the information from the automatic management screen (see 3.2 Auto Screen), and several parameters from the manual management screen (see below). Basically, when an SRF version is entered at the automatic management screen, and the same SRF version has already been run through the MEMMAN

system once, the that SRF's configuration file will be retrieved and all of the automatic & manual parameters will be restored to what they were in the previous run.

All of the data is stored in a single file record variable of type *RunConfigType*, which is then written to the configuration file. Consequently, each configuration file (for each SRF version) has only one record in it. The following table shows all of the items saved in the configuration file. Each entry shows the name of the variable saved, a description of the variable's function, and the name of the *RunConfigType* field it is stored in.

Variable	Description	RunConfigType Field
PriProc	Primary (master) processor	Proc
AutoGoal	Automatic management goal	AGoal
UpLimA	Upper memory limit for CCSA	ULA
UpLimB	Upper memory limit for CCSB	ULB
LwLimA	Lower memory limit for CCSA	LLA
LwLimB	Lower memory limit for CCSB	LLB
MemA	Latest CCSA word count	MemoryA
MemB	Latest CCSB word count	MemoryB
MoveCycs	Flag governing cyclic movement	MvCycs
MovePlats	Flag governing scan platform command movement	MvPlats
MainChkSet	Flag for CHKPNT SET 0	MnChkSet
ManualCount	Number of entries in the Manual array	ManCnt
Manual (array)	Array of descriptions of manually moved items/groups	ManArray (array)
CatCount	Number of command categories	CatCnt
Category^[n].Moveable	Flags governing movement of categories 1 through n	CatStat (array)
OutputFmt	Output file format (SRF or runstream)	OutType
InSrfTime	Time stamp from input SRF	TLastSRF
LastVer	MEMMAN version used to create the configuration file	VLastMM
ShowComments	Display comments flag	Comments

As noted above, an attempt to read the configuration file is made any time the user enters a new input SRF

version at the automatic management (main) screen. The file is written however on three separate occasions.

First of all, after the user has entered the necessary data at the automatic management screen, they would press "G" to begin the automatic management attempt (see 3.3 SRF Reading). When the SelectMoveItems procedure (Man_Scr1.PAS TPU) detects the press of the "G" key, it will exit the main loop, and prepare to exit the procedure. The final step however is to check the *Abort* flag to see if it is set (it is set to "true" if the user pressed the Escape key to leave the MEMMAN program). If the flag is not set (nominally the case) the procedure UpdateSeqConfigFile will be called (also in the Man_Scr1.PAS TPU) to write the updated parameters (listed above) to the configuration file.

Secondly, if the user presses the <F3> key (re-attempt automatic management) during manual management, the procedure UpdateSeqConfigFile is called to update any manual management parameters in the configuration file, before the user is returned to the automatic management screen.

Finally, one of the final steps in the GenerateOutput procedure (Man_Gen.PAS TPU) is to call the UpdateSeqConfigFile procedure in order to update any manual or automatic management parameters in the configuration file, before the program ends.

4.6 Force File

The force file (named SRFversion.FRC) is an ASCII text file which contains a list of command masks (strings) with instructions to force matching commands to a specific processor. The format of the force file can be seen in the sample shown on page 44 (A.4 Sample Force File). Notice that each command may have exceptions to the force instruction, as shown with the "PROG" macro command. In this case, any records read from the input SRF which contain the strings "CC16" or "TAPPOS" will be forced into CCSB (via CONFIG macros). In addition, any records read from the input SRF which contain the string "PROG" except for ones which also contain either the string "DC2P" or "CC3A". Exceptions only apply to the force command which immediately precedes them.

There are four procedures (functions) which relate to the force file and are accessed by other TP units; ReadForceFile, ShouldForce, ForceCommand and NewForce (all found in the Man_Frc.PAS TPU). In addition, a local function called ForceException is called by ShouldForce.

The procedure ReadForceFile opens the force file and reads all of the instructions into memory. For further information on how the force file is read from disk, see 4.6.1 Reading the Force File.

The boolean function ShouldForce is called by the LoadArray procedure (Man_Load.PAS TPU) after a command has been added to either the *Current* or *Overlap* array. The function is passed the input SRF buffer contents (see type *BufStrType* in the Man_Vars.PAS TPU), the relative command extracted from that buffer and the column it appears in. The ShouldForce routine simply loops through the complete *Forced* array to see if any of the *ForceText* fields (of the *Forced* array) match any characters in the present buffer. Exceptions are not checked at this time, they are checked later in ForceCommand. This function continues to loop through the *Forced* array even after a match has been found, to see if more than one force record applies to the buffer. If so, the user will be notified and the instructions from the first occurrence will be used.

The ForceCommand procedure is also called by the LoadArray procedure (Man_Load.PAS TPU), but it is called after the ShouldForce function has returned a value of "true" to indicate that the present command should be forced. The procedure is called with the boolean flag *InOverlap* which is set to "true" if the

command is an overlap event. The procedure begins by checking the *Proc* field of the matching *Forced* array element to see if it is the same as the primary processor (variable *PriProc*). If it is the same (in other words a force to the master) then the command is simply marked as unmovable (as long as a call to ForceException does not return a value of "true").

If the command is instead to be forced to the secondary (usually the case) then the following steps are taken. First of all, the present value of the global variable *Proc* (present command's processor) must be checked to see if the command is still in the primary and is not a force exception (should be forced to the secondary in the *Auto* array), or if it is already in the secondary (should be marked to skip in the *Auto* array).

If the command is presently in the primary, then the global variable *Proc* will be assigned the value of *SecProc* and the command's *Current* or *Overlap* array element's *Proc* field will be updated (to the new value of the global *Proc* variable). In addition, the command's *Moveable* and *ManMove* boolean fields will be set to "false" so that they are no longer moved (during any automatic or manual management). Finally, to force the command movement, the command must be in the *Auto* array with an event type of Force, or a move type of Use with the *Occur* field of the *Auto* array element adjusted to reflect that the command will be moved (via CONFIG macros calls) during output file generation. If a call to InAutoList (Man_AList.PAS TPU) returns a value of "true" and the found entry is Use type, then there is already an entry in the list for this command and the *Avail* and *Occur* fields will just be updated. Otherwise, if no entry could be found in the *Auto* array to match the command, then AddToAutoList (Man_AList.PAS TPU) is called to add the new command to the *Auto* array as a Force type with the *Occur* field set to a value of one.

If the command is already in the secondary, then almost the same steps will be followed as if the command were in the primary. The differences will be that the global variable *Proc* will not be changed to the value of *SecProc* (they are already the same), and the matching *Auto* array element will be of type Skip and not Use (either this matching element already exists and will be updated, or it will be added as a Force type, similar to above).

4.6.1 Reading the Force File

In general, the force file is read in much the same manner as the other ASCII MEMMAN files (such as the system parameter file, the category file and the SMD file). The file must begin with a record which contains the string "$$FORCE", and must end with a record which contains the string "$$EOF". As can be seen in the sample shown on page 44 (A.4 Sample Force File) valid commands in the force file are FORCE and EXCEPT. The FORCE command must be followed by a command mask (a string in quotes), a colon, and either an A or B to signify which processor the command belongs in. The EXCEPT command requires simply the command mask (a string in quotes) since it applies to the preceding FORCE command.

As the records are read from the file, they are checked for correct syntax, and then stored in memory. All of the valid force commands are stored in the *Forced* array (type *FrcArray* which is an array of elements of type *ForceType*). All of the exceptions to a particular force statement are stored in a dynamically created linked list of *ExceptType* variable. The *Forced* array elements have a field called *ExceptPtr* which is a pointer to the first (last if only one) *ExceptType* variable. Then each *ExceptType* variable in turn points to another, until the last one, which has a pointer value of zero (nil).

The size of the *Forced* array is contained in the variable *ForceCount*, and is adjusted every time a force command is read from the file. Then the *ForceText* field of the array element is given the string from between the quotes, and the *Proc* field is assigned a value of ProcA or ProcB (depending on the processor character).

If exception commands follow the force command (in the file) then for each one, a new dynamic variable (called *Exception*, type *ExceptPtrType*) is created. Then, the *ExceptText* field is given the string from between the quotes, and the *NextPtr* field is given a value of zero (nil). If the *ExceptPtr* field of the present *Forced* array element is equal to zero (nil), then this is the first exception, and the pointer field is pointed to the new exception variable [location]. If the *ExceptPtr* field of the present *Forced* array element is not equal to zero (nil) then this is not the first exception, and the linked list of exceptions must be traversed to find the last exception in the list (the *NextPtr* field of the last exception in the list will have a value of zero or nil). Then the zero value of that last *NextPtr* field is replaced with the address of the new exception, thus making it [the new exception] the last item in the linked list.

The resulting data structure is sort of a two dimensional array, with one dimension containing all of the force command descriptions, and the other containing variable length lists of exceptions (one list for each force).

4.7 Log File

The log file is a simple ASCII text file which contains information related to every completed run of MEMMAN. An entry in the log file is intended to provide a record of MEMMAN runs; the input SRF version, the users, and the results (output file names). A completed run is defined as a run where the user terminates in the normal fashion, by generating an output file such as a new SRF or a Univac runstream.

The sample log file seen on page 45 (A.5 Sample Log File) shows the information contained in the log file. The only procedure relating to the log file is one called MakeLogEntry (Man_Log.PAS TPU). This procedure is explained in more detail in 4.7.1 Updating the Log File. The log file is only updated from one place in the program. MakeLogEntry is called at the start of the GenerateOutput procedure (Man_Gen.PAS TPU).

4.7.1 Updating the Log File

The procedure MakeLogEntry (Man_Log.PAS TPU) opens the log file (MEMMAN.LOG), writes information relating to the present run, and the closes the file. If an error occurs upon attempting to open the log file, then the error code is checked to see if the error was that the file does not exist. If this is the case, then the file is simply created and the header record (column titles) is written to the file. Then whether the file existed or not, the information from the run is placed (written) in the file and the file is closed.

Appendix A: Sample ASCII Files

A.1 Sample System Parameter File

```
$$SYSPAR
DataPath      "C:\seq\"
SysPath       "C:\Seq-Tool\MemMan\"
LowBoundA     "4266"
LowBoundB     "4277"
UpBoundA      "6660"
UpBoundB      "6660"
OutputFmt     "Runstream"
ShowComments  "False"
$$EOF
```

A.2 Sample Category File

```
$$CATEGORY
;
; ISSCHG command (not used during VIM)
;
CATEGORY "ISSCHG":TRUE
MASK "ISSCHG"
;
; Cyclics
;
CATEGORY "CYCLICS":TRUE
MASK "MEND"
; (Using the MEND macro is the only way to "mask" cyclic
definitions.)
;
; TWNCs and MOD INDEX commands
;
CATEGORY "DC2P/PR/CC3A":TRUE
MASK "DC2P"
MASK "DC2PR"
MASK "CC3A"
;
; FDS datamode commands
;
CATEGORY "06BB Cmds":TRUE
MASK "06BB"
;
; AACS scan platform related commands
;
CATEGORY "Scan Plat Cmds":TRUE
MASK "PLATPO"
MASK "PLINC"
MASK "MOSAIC"
MASK "SCAN"
;
; AACS deadband control commands
;
CATEGORY "AC7PAR 6741/42":TRUE
MASK "AC7PAR,(6741"
MASK "AC7PAR,(6742"
;
$$EOF
```

A.3 Sample SMD File

```
$$SMDF
;=========================================================
;
; BR1 - Gyros on (GYONAB, GYONBC)
;
"BR01GO",2,PROCB,FALSE,FALSE
;
;=========================================================
;
; BR2 - SS-HGA Calibration (ASCAL)
;
"BR02GO",2,PROCB,FALSE,FALSE
;
;=========================================================
;
; BR3 - Magnetometer roll maneuver (MAGROL)
;
"BR03GO",2,PROCB,FALSE,FALSE
;
;=========================================================
;
; MTRBRN - AACS Motor Burn
;
"MTRBRN",21,BOTH,FALSE,FALSE
;
;=========================================================
;
; COMSIM ENV INPUT - Transfer (interrupt)

"TXFER",0,UNDEFINED,FALSE,FALSE
;
;=========================================================
;
; DSS - Tape position command
;
"TAPPOS",2,EITHER,FALSE,FALSE
;
;*********************************************************
$$EOF
```

A.4 Sample Force FIle

```
$$FORCE
FORCE "CC16C":B
FORCE "TAPPOS":B
FORCE "PROG":B
EXCEPT "DC2P"
EXCEPT "CC3A"
$$EOF
```

A.5 Sample Log FIle

```
  INPUT     OUTPUT       DATE       TIME     USER
A001H.SRF A001H.CFG 02-26-1990 14:55:32 GREG
A001H.SRF A001I.SRF 03-02-1990 09:31:28 WELCH
A001H.SRF A001I.SRF 03-02-1990 09:37:40 WELCH
A001H.SRF A001I.SRF 03-02-1990 09:42:54 WELCH
A002D.SRF A002D.CFG 03-08-1990 13:12:16 WELCH
A002D.SRF A002D.CFG 03-08-1990 13:20:45 KO
A002D.SRF A002D.CFG 03-08-1990 13:22:32 KO
B005E.SRF B005E.CFG 04-20-1990 15:58:58 CHUCK
B005E.SRF B005E.CFG 04-20-1990 16:10:28 WELCH
```

Appendix B: Glossary

ASCII	American Standard Code for Information Interchange
DOS	Disk Operating System
Master	Default CCS processor where command will be placed (unless otherwise instructed)
MSDOS	Microsoft Disk Operating System
PKARC	MSDOS program used to place files into an archive
PKXARC	MSDOS program used to extract files from an archive
Primary	Same as Master
Secondary	Same as Slave
Slave	CCS processor which supports the Master processor (where commands will be placed if there is not enough room in the Master).
TP	Turbo Pascal
TPU	Turbo Pascal Unit
VIM	Voyager Interstellar Mission

Appendix C: Notes

1. Jet Propulsion Laboratory, "MemMan Functional User's Guide"

2. Borland International, "Turbo Pascal 4.0 Owner's Handbook", Copyright 1987