State Sequence Handlers

Edward B. Rawson

Rawson Engineering
Lincoln, Massachusetts

Abstract

We describe a new type of sequence control structure which allows an ordinary **:** word to behave as a small state machine. The control structures may be freely mixed with **BEGIN**, **UNTIL**, etc., and with **IF**, **ELSE**, and **THEN**, with conventional nesting restrictions. The new structures include analogs of **BEGIN**, **UNTIL**, etc., as well as several other words which support terse, readable state machine code.

The normal state variable is replaced by an execution pointer, whose handling is mostly hidden in the source code. Application code never need supply a pointer value. Definitions may be understood as standard FORTH procedures, with the understanding that execution may "hang up" in one of the internal loops or called procedures until it is proper to proceed. Repeated calls to the main procedure result in repeated execution of the loop code or the called procedure until continuation is appropriate.

The sequence control structures encourage structured code, make state entry and exit natural, and produce fast, compact, object code. They eliminate the need to assign and manage state numbers.

The structures and several applications are described.

Introduction

State machines are often used in the management of physical hardware. For example, a motor may be running or stopped, or executing some sequence of activities. such as dynamic braking. The nature of the current activity is coded into a state variable, and the software which manages the motor executes the corresponding state handler code periodically in order to sample inputs, to control outputs, and to change the state itself as needed.

The references will give the reader some background on the finite state machine concept. [SMA82] describes the concepts which have evolved for hardware state machines. These are more formal and limiting than we require. [BAS83] gives a simple Forth example of a software machine. [STA83] describes a mixed hardware/software implementation. From our point of view, the example of the motor control code includes all the important features.

The fact that the state handlers can change the state makes this "go-to" code. Any handler can cause a jump to another handler at the next invocation of the main control code.

Even with good support, such state machine code is tedious to write and difficult to understand. Transitions caused by external events may not be anticipated in the design. State numbers often appear as numeric literals. Recursive calls to the master procedure or direct jumps between state handlers are often used to improve speed when the state changes. The more complex the situation, the more the code looks like the "spaghetti" code which was so common 20 years ago. Careful assignment of states and construction of action tables reduces, but does not eliminate, the "go-to" nature of the control flow.

We have recently developed an alternative scheme which works well in situations requiring sequence management. Its utility for other state machine applications has not been tested. Our application environment has several important characteristics:

1) Real-time control of some piece of hardware is involved. Hence the main procedure is being called repeatedly, usually at intervals of 50-100 milliseconds.

2) The control software is handling extended operation sequences, often lasting a minute or more.

3) In addition to sequence management, the control software must do "housekeeping", such as sensing an input and updating a stored value which represents that input. Housekeeping procedures may be state dependent or may be required at all times.

4) States require entry and exit procedures which may be the most complex parts of the code, the handlers sometimes being only waits.

5) State entry and exit are often not well behaved, occurring as a result of external actions which are beyond the programmer's control. An operator may abort a complex sequence and demand that a different one be started, with little regard for the state of the hardware, and of course no knowledge of the state of the software.

Our state sequence scheme allows any : word to act as a miniature state machine. The only required external resource is a variable to store an execution pointer. This replaces the usual state variable. Machines having 10 internal states are accommodated easily. Those having 100 internal states would need to be subdivided. A : word which is acting as a state machine is defined normally and reads normally, except for the inclusion of special control structures of the same sort as **BEGIN ..... UNTIL** loops. These special structures may be nested with conventional loop and conditional structures, but may not be inside of **DO** loops. Repeated calls to such a : word cause only a fraction of its code to be executed, with the executed piece corresponding to the state handler in a conventional state machine.

In the remainder of this article, we often refer to the high-level (inner interpreter) instruction pointer. In the interest of brevity, we use a common alternate name, the IP register.

Basic Syntax

A simple example of the syntax is given in Figure 1.

```
: ssEXAMPLE   HOUSEKEEPING  xPOINTER  ssBRANCH  t/fA
      IF  WORKING_ENTRY
         ssBEGIN  WORKING_CODE  t/fB  ssUNTIL
         WORKING_EXIT
      THEN
      ssBEGIN  IDLE_CODE  ssAGAIN  ;
```

Figure 1

ssEXAMPLE has two internal states, corresponding to WORKING_CODE and IDLE_CODE. For its operation, ssEXAMPLE requires two support variables, ssCURR and xPOINTER. xPOINTER is private to ssEXAMPLE. Another state sequence word would use another variable. xPOINTER corresponds to the state variable in an ordinary state machine. ssCURR is shared by all state sequence words in the system. Its function is to point to the execution pointer, in this case xPOINTER.

HOUSEKEEPING is a procedure which is executed on every call to ssEXAMPLE. xPOINTER has been described above. ssBRANCH saves the location of the xPOINTER storage cell in the working variable, ssCURR, for use by other words. It then tests the stored value in xPOINTER. If it is zero, ssBRANCH takes no other action. If the stored value is non-zero, ssBRANCH copies it into the IP register, causing a direct jump to the indicated point within ssEXAMPLE. We assume for the moment that the stored value is zero.

Pointer relationships are shown in Figure 2. Since ssCURR is a common working variable, it may have been used by another procedure. Hence, it must be set by ssBRANCH on each invocation of ssEXAMPLE. On the other hand, xPOINTER is "owned" by ssEXAMPLE, and its contents may be stable through many invocations.

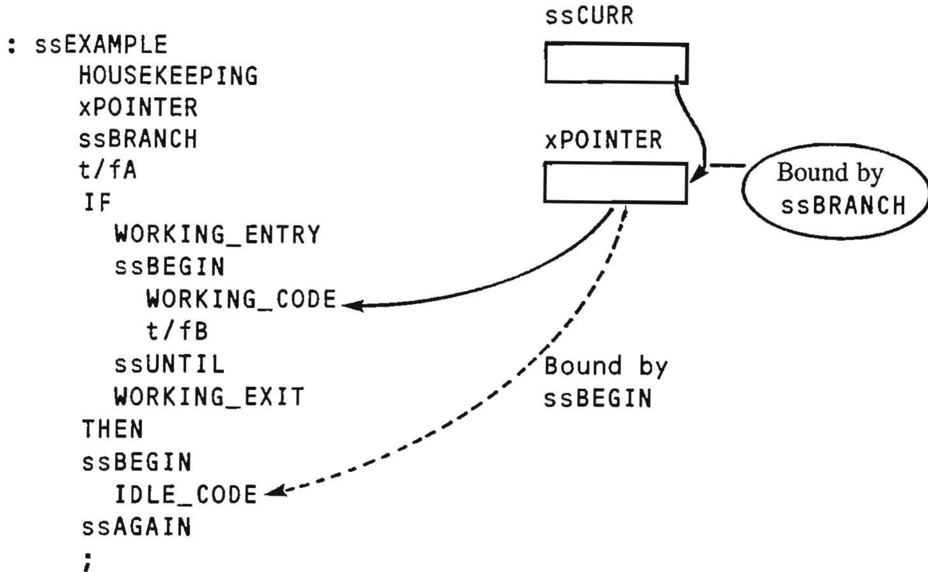

Figure 2

If t/fA returns a true value, execution continues with WORKING_ENTRY and ssBEGIN. ssBEGIN copies the IP register into the xPOINTER storage cell, using the contents of ssCURR to locate the cell. On a subsequent call to ssEXAMPLE, the stored value will cause ssBRANCH to jump directly to the call to WORKING_CODE.

After ssBEGIN has stored the pointer, execution continues with WORKING_CODE, t/fB, and ssUNTIL. If t/fB returns a true value, ssUNTIL is a NOOP, just as UNTIL would be. If t/fB returns a false value, however, ssUNTIL causes an EXIT from ssEXAMPLE. On a following call to ssEXAMPLE, the execution sequence will be as shown in Figure 3.

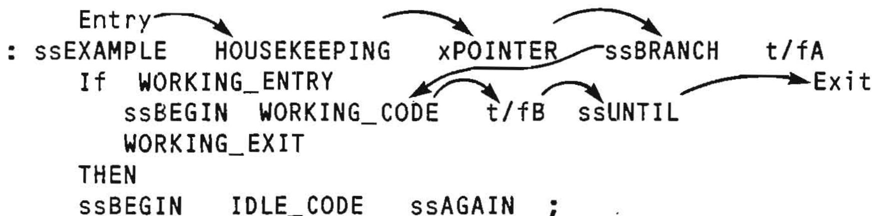

Figure 3

We have created a pseudo-loop, which requires repeated calls for repeated execution.

When t/fB returns a true value, execution continues to WORKING_EXIT and the next ss loop. Execution of ssEXAMPLE finishes with IDLE_CODE. ssAGAIN is an unconditional EXIT (the ; is never executed). Further calls will result in the execution sequence shown in Figure 4.

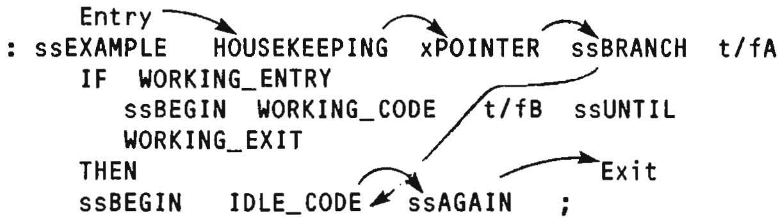

Figure 4

In order to restart the ssEXAMPLE sequence, we need only set the xPOINTER storage cell to zero. This can be done by a procedure within IDLE_CODE, or by some external process. If t/fA returns a false value, execution drops into the second pseudo-loop right away, avoiding WORKING_ENTRY, WORKING_CODE, and WORKING_EXIT.

Single Level Structures

With ssEXAMPLE as an illustration of the approach, we can now list the run-time behavior of the control words which form the basic state sequence "package".

ssBRANCH	Saves the address of the execution pointer in ssCURR. Branches to the location given by the pointer. 0 – no branch.
ssENTRY	Same as ssBRANCH, but uses a supplied truth value to decide whether to branch. True – no branch.
ssBEGIN	Saves the IP register in the execution pointer.
ssUNTIL	Tests the top-of-stack value. True – continue False – EXIT.
ssWHILE	Tests the top-of-stack value. True – continue. False – jump to after following ssREPEAT.
ssREPEAT	Unconditional EXIT.
ssAGAIN	Unconditional EXIT.
ssINIT	Resets the execution pointer to 0. Must be used with ssBRANCH.
ssPAUSE	Marks the execution point just as ssBEGIN does, then EXIT's.
ssEND	Synonym for ssBEGIN.

Examples of definitions for these words are given in screens 36-38. Syntax documentation (another version of the material above) is given in screens 31-35. Implementation details and explanations of non-standard FORTH nomenclature are given in screens 20-22. Our development system is based on the CPM80 version of polyFORTH II™

The word ssPAUSE is useful for constructing loops in which only a portion of the loop is to be executed on each call. Such behavior is often needed for background processes in real-time control systems. Screen 34 gives an example. Such code could of course be constructed with a selection (state) variable, but the code would be longer, harder to write, and harder to understand.

ssEND is provided only to enhance readability of source code. It may be placed at the end of a word which uses ss structures. It makes the word into a noop for any calls after the sequence is complete. Placing ssBEGIN just before the ; would have the same effect, but the code would be confusing for a reader.

```
: +checksum   cHECKSUM-POINTER
     ssBRANCH  cScurr  CV0!
       IH> 100  IH> 1FFF  PsSTART
       ssBEGIN  1sCHECK  PsDONE  ssUNTIL  0  CsSAVE
       IH> 2000  IH> 3FFF  PsSTART
       ssBEGIN  1sCHECK  PsDONE  ssUNTIL  1  CsSAVE
       IH> 4000  IH> 5FFF  PsSTART
       ssBEGIN  1sCHECK  PsDONE  ssUNTIL  2  CsSAVE
       cScurr  CV0NOT  IF  6  !kFLAG  THEN
     ssINIT ;
```

Figure 5

Figure 5 gives an example of the application of state sequence structures to an internal computer problem, that of carrying out a checksum of PROM at run time. We sum only a few bytes on each call, so that only a small part of our processor time is used. As the sum for each prom is completed, we save the cumulative sum, so that an error can be localized to one of the three proms in the system. Prom boundaries are irregular. The first 256 bytes are avoided to avoid interaction with CPM when the prom image is loaded under CPM as a transient program. At the end of the check cycle, if the sum is non-zero, a flag is set to disable the outputs to the controlled hardware and thus assure safety of the machinery. Screens 46-48 show the support definitions for `+checksum`.

The computation is based on a procedure `cKsum`, which takes as arguments a starting address and a byte count, and which returns an 8 bit sum. The main procedure is unusual only in the use of variables to store the state of the operation between invocations of `+checksum`. This is necessary because our state machine does not have private stacks. Every invocation of `+checksum` must finish with the (public) stacks unchanged. The same restriction would apply to any other state machine structure.

The major virtue to be found in this code is readability and ease of change. The "irregularities" and requirements mentioned in the problem description are accommodated naturally. Auxilliary arrays are replaced by in-line literals, a reasonable choice when only 3 proms are involved. We avoid one internal loop which would require a stored index. A notable feature of the `+checksum` code is the repetition of the sequence

```
ssBEGIN  1sCHECK  psDONE  ssUNTIL
```

three times. We could have combined `1sCHECK` and `psDONE`, but the repetitive code is a necessity, because `ssBEGIN` must be in the same word as `ssBRANCH`. Placing it in a called procedure will not work, because both words refer to the IP register. We next describe a technique which, in a limited way, gets around this problem, condensing the `ssBEGIN ......  ssUNTIL` sequence into a single procedure call.

Called Procedures

The restriction of state sequence mechanisms to single-level structures can be partially removed. This allows more complex machines without having the main procedure grow too large. In cases such as the checksum procedure shown above, we can also reduce code size by re-using called procedures.

We define a special form of `:` word which can behave, on its own, as an entire

```
ssBEGIN .... t/f ssUNTIL
```

clause when called by the main procedure. Such `:` words must be called directly by the main procedure (the one containing `ssBRANCH` or `ssENTRY`). They manipulate the IP register and the `ss` execution pointer to accomplish the required sequence management. Figure 6 shows the

+checksum code rewritten with such a word. Execution "hangs up" at each call to **PrSUM** until **PsDONE** indicates that the current prom is completed. **ssNEXT** causes the next invocation of **+checksum** to branch to the word following **PrSUM**. Figure 7 shows the pointer relationships and the action of **ssNEXT**.

```
:ssPROC PrSUM   1sCHECK  PsDONE  IF  ssNEXT  THEN  ;

: +checksum   cHECKSUM-POINTER
     ssBRANCH  cScurr  CV0!
       IH>  100   IH> 1FFF   PsSTART  PrSUM  0  CsSAVE
       IH> 2000   IH> 3FFF   PsSTART  PrSUM  1  CsSAVE
       IH> 4000   IH> 5FFF   PsSTART  PrSUM  2  CsSAVE
       cScurr  CV0NOT  IF  6  !kFLAG  THEN
     ssINIT  ;
```

Figure 6

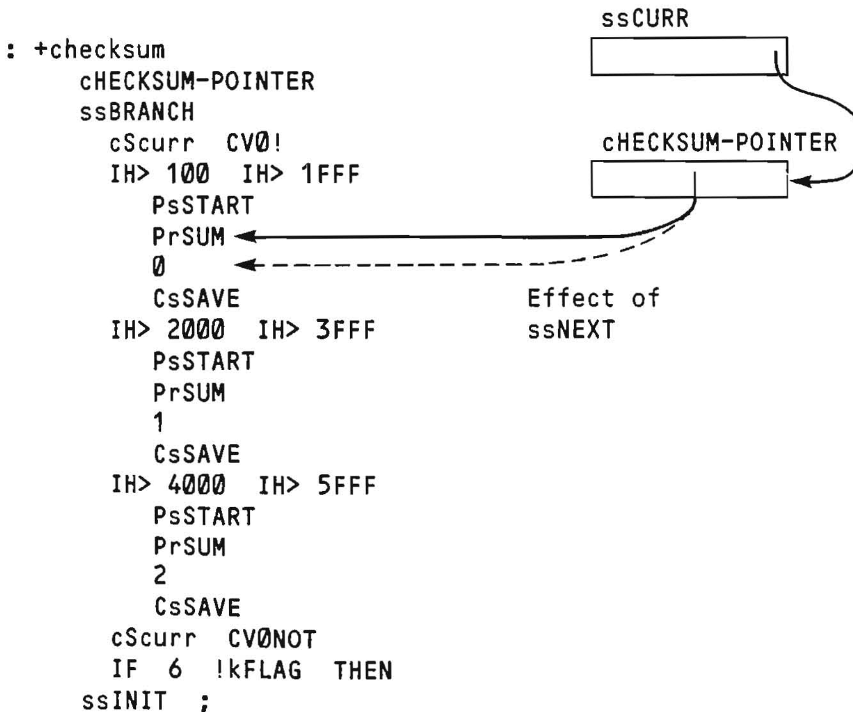

Figure 7

Code for our **:ssPROC** implementation is given in screen 39, using 8080 assembly code. The normal **:** word run-time code is replaced by code which works directly with the caller's IP register value. At call time, this points to the word in the main procedure which follows the call to the **:ssPROC** word. The IP register is backspaced to point again to the called **:ssPROC**, and then saved in the **ss** execution pointer. This is similar to the action of **ssBEGIN**. Next the IP register is loaded with the PFA of the **:ssPROC** word, and execution continues. We have produced a direct jump to the "called" procedure. Nothing has been placed on the return stack. Hence if the **:ssPROC** word ends normally, or if **EXIT** is executed anywhere within it, we will cause an **EXIT** from the main procedure. This is similar to the action of the sequence

```
....false  ssUNTIL
```

appearing in the main procedure. The execution sequence is shown in Figure 8.

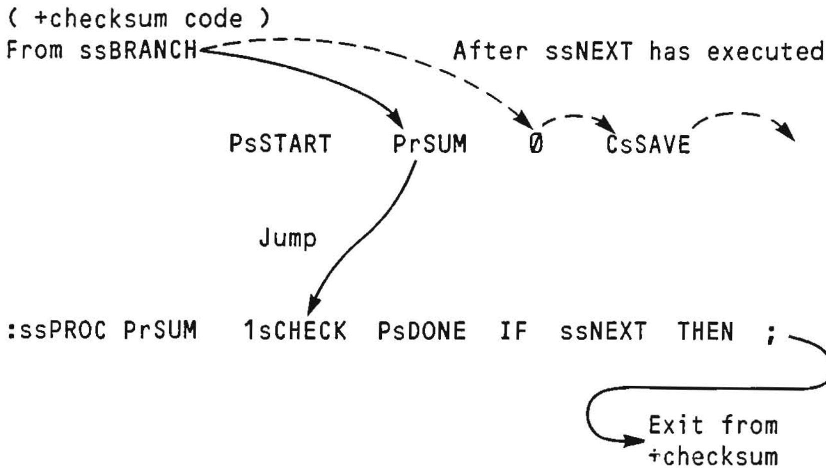

Figure 8

If we take no action to change things, we have created a pseudo-loop. At each entry to the main procedure, `ssBRANCH` or `ssENTRY` will jump to the `:ssPROC` word. This word saves a decremented IP register value, so that the call will repeat, and then executes. At exit, it forces an exit from the main procedure.

There are two ways to allow continuation of the main procedure. The first is `ssNEXT`, which mimics the action of

```
.... true  ssUNTIL  ssPAUSE
```

`ssNEXT` need only increment the execution pointer by 2, so that on the next invocation of the main procedure, the `:ssPROC` word will be skipped, as shown in Figure 8.

The second word, `ssCONTINUE`, mimics the action of

```
.... true  ssUNTIL
```

and in addition causes an immediate exit from the `:ssPROC` word. It acts by incrementing the execution pointer and then loading it into the IP register. This causes an immediate jump back into the main procedure. (Figure 9)

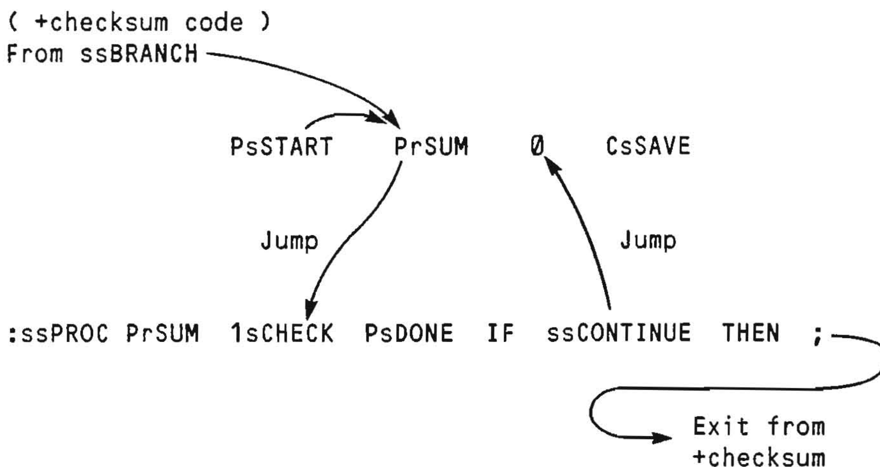

Figure 9

A third technique would be to increment the execution pointer and then push it onto the return stack, so that the return to the main procedure would not occur until an EXIT from the :ssPROC word. This would be similar to (but not always more convenient than) ssCONTINUE, and it would also be slower. We have not implemented such a command.

ssCONTINUE must be executed directly in the :ssPROC word. Since it causes an immediate exit, the stacks must be in order before it is executed. In contrast, ssNEXT can be executed within a DO loop (just once), or by a called procedure. Since it does not cause an exit, stacks may be cleaned up during the remainder of the :ssPROC word.

State Sequence Motor Handler

Screens 90 through 100 give an example of a fairly complex state sequence word and its support. The main procedure is also shown in Figure 10. :ssPROC calls have been underlined. Each call corresponds to a state of the Mprocess state machine.

```
 1 : Mprocess
 2     BEGIN  ?mSEQnew  ssENTRY  mEPROCESS
 3       mTIMER  TWCANCEL  mPgTIME  V0=
 4       IF  ?mSTOP  mTmWAIT  ?mCOMMAND  mLIMIT
 5         IF  mREVERSE  mTmWAIT  sPDYNAMIC  mTmWAIT
 6             mSTOP  mTmWAIT
 7         THEN  mBUSY  CV0!  MOTOR-IDLE
 8       ELSE  -mCLIMIT
 9         IF  ?cSTOP  mTmWAIT  ?mCOMMAND  ?cREVERSE  mTmWAIT
10             ?mSTART  mRUN-PROC
11         ELSE  mSTOP_LOCAL
12         THEN
13       THEN
14     AGAIN  ;
```

Figure 10

The code controls a motor which is turning an actuator shaft. The actuator moves a piece of machinery back and forth between limits. Two limit switches, called the in-limit and the out-limit, close at the ends of the mechanical travel range. Motor control uses a run relay and a reverse relay, each controlled by an I/O bit.

The basic command is to run (in or out) for a given number of real-time clock ticks. A command to run for longer than it takes to go from limit to limit gives the effect of a continuous run command. A command to run for 0 ticks is interpreted as a stop command.

We must sense two errors. In order to detect a stalled motor, we start a timer whenever we start the motor. If the motor runs for too long without reaching a limit, it is considered to be stalled. The other error is a stuck limit switch. If a switch sticks open, the motor will stall against a mechanical stop, and we have a stalled motor error. If the switch sticks closed, we will eventually have both switches closed at once, and the code specificially tests for this condition.

The motor code must accept external commands at any time. It processes and then obeys these commands as best it can, considering the current state of the hardware. New commands override any actions which are in process. The current state of the motor is indicated by a busy flag and by an error flag.

The main motor code (screen 100) has two major states, an idle state and a run state. Substates needed to manage the run and reverse relays lead into the major states. At the end of a run command, or when a limit is reached, the motor is stopped by going to the idle state.

Exit from a major state may occur because of a command, or because a limit or an end-of-motion condition occurs. `Mprocess` has been written as an endless `BEGIN ... AGAIN` loop. If `ssCONTINUE` is executed by one of the major state processes, an immediate re-entry occurs. This allows `Mprocess` itself to handle the sequence of actions required when conditions change, without the latency of waiting for the next invocation. Such a quick response is necessary, for example, when reacting to a limit switch closure.

The state sequence design makes it natural for `Mprocess` to "fall through" unneeded substates without significant latency as execution proceeds toward one of the major states. An exit from `Mprocess` occurs only when the software must wait for some external event before continuing.

State sequencing design is important to code simplicity and clarity in a case such as this. Not only can the software be in several possible states when a command arrives, but the physical hardware can be in several possible states. The universal reaction to an external event is to re-start the `Mprocess` sequence, either by forcing a new sequence the next time `Mprocess` is called, or by looping back to the initial `BEGIN`. The command value controls an initial branch. The subsequences then deal with the hardware situation:

a) Does the motor need to be stopped? If so, stop it, and start the timer.
b) Wait for the timer to time out.
c) Does the motor need to be reversed? If so, reverse it, and start the timer.
d) Wait for the timer to time out.
e) Etc.

Procedures a) and c) may be noop's, and if either is, then the following wait is also a noop.

A conventional state machine could be built with the same state assignments that `Mprocess` has. Our experience is that such designs are not the natural first approach with conventional machines. The designer thinks of individual conditions, such as "waiting for the run relay to drop out", and assigns states to them. In contrast, `Mprocess` has three states which fit this description, corresponding to the `mTmWAIT` calls following stop commands on lines 4, 6, and 9.

With a conventional state machine, the re-entry design of `Mprocess` is not obvious. The designer's natural way of handling an external event is to determine what the next state should be, carry out the state exit processing, and invoke the handler for the new state. The strategy of retreating to an initial (software) state when an external event occurs, and then "falling through" unneeded states seems wasteful and indirect. However, for `Mprocess`, this strategy makes for a great simplification of the state transition code. With the state sequence design, it is compact and fast, as well as being a natural design approach.

Nesting State Machines

The state sequence words described above use an auxilliary variable, `ssCURR`, during the operation of any `ss` control structure. If one of these words calls another, the called procedure will destroy the callers information in `ssCURR`. The proper place to protect this information is in the called procedure, since the caller should not have to know whether the called procedure uses `ssCURR`. We can do this by writing

```
: mPROCESS  ssCURR  @  Mprocess  ssCURR  !  ;
```

and then letting `mPROCESS` be the external entry point for servicing a motor. Since `Mprocess` is required to leave the parameter and return stacks unchanged, this will always work.

Comments

We have described a new form of control structure which is useful in building state machines which have (mostly) sequential behavior combined with waits. The resulting code is easy to read, compact, and fast.

In a normal state machine, an early and critical design activity is state assignment. State sequencing must be considered at the same time. When the code is written, sequencing is hidden in the individual state handlers, which usually contain explicit state variable manipulation code.

In contrast, state sequence state handlers never control selection of the next state. The handler can only say "call me again", "start over", or "continue". Selection of the next state is always done by the main procedure, using conventional FORTH control structures. This has a profound effect on the ease with which the code can be read and understood.

The main procedure of a state sequence machine may be written without consideration of even the idea of coded state numbers. The procedure is designed as a normal FORTH word, which has the additional ability to "wait" at any point until it is proper to continue. The programmer need only keep in mind that words prior to `ssBRANCH` (or `ssENTRY`) will be executed repeatedly, in addition to the words in the "wait" loop or procedure.

The facilities offered by state sequence machines suggest new designs which have proved to be very useful in our applications. It is to be hoped that others will also find the scheme useful, and will extend it and its applications.

Acknowledgement

The work reported here was done under contract to Butler Automatic, Inc., of Canton, MA. The author is grateful for support and encouragement during the development work, and for permission to publish this description of the software.

References

[SMA82] Small, C. "Finite-State Machines, Theory, Synthesis, and Minimization" *1982 Rochester Conference on Databases and Process Control.* Rochester: Institute for Applied Forth Research, 1982.

[BAS83] Basile, J. "A FORTH Finite State Machine" *Journal of Forth Application and Research,* Vol. 1, No. 2, December 1983.

[STA83] Starling, M. K. "A Hardware/Software Finite State Machine Implementation" *1983 Rochester Forth Applications Conference.* Rochester: Institute for Applied Forth Research, 1983.

Manuscript received June 1985.

Dr. Rawson received his B. A. degree from Swarthmore College in 1948. He received the Ph.D. degree in physics from Yale University in 1953. He worked for several years at MIT Lincoln Laboratory, on both radar and data handling systems. His work in these fields was continued at Science and Engineering Institute, in Waltham, Mass. In 1968, Dr. Rawson was one of the founders of Searle Medidata, a firm specializing in medical data systems. He served as Vice President and Engineering Director of the firm until 1974.

Since that time, Dr. Rawson has been in private practice as a consultant, data system designer, and computer programmer. His clients have included government agencies, hospitals, and industrial concerns. During the past few years, he has concentrated on software projects using the FORTH language.

Dr. Rawson holds both U.S. and foreign patents. He is a member of Sigma Xi, the AAAS, the ACM, and the IEEE.

```
SCREEN: 20
 0 ( Implementation notes - 5/85)        EXIT
 1 Development environment is based on the CPM80 version of
 2 polyFORTH II  tm  .  Processor is a Z80.
 3 Assembler is an 8080 assembler with a full set of Z80
 4 extensions.
 5 Register assignments -
 6    I  ~  BC    --   High level instruction pointer (FORTH IP)
 7    I' ~  C
 8    W  ~  DE    --   Points to PFA - 1 at entry into a word.
 9                     Otherwise it is a scratch register.
10    W' ~  E
11          HL    --   Scratch
12    Alternate register set -- scratch
13 The USER area pointer and the return stack pointer are
14 contained in the named variables U and R.
15 The CPU stack is used as the parameter stack.

SCREEN: 21
 0 ( Implementation notes - 5/85)        EXIT
 1 UOFFSET returns the offset of any named user variable
 2 US) is a macro for handling user variables.
 3 : US)   UOFFSET W LXI   U LHLD   W DAD   M  ;
 4 For example:   BASE US) A MOV  moves BASE to the accumulator.
 5
 6 n  :NBRAM Name   defines a ram area.  VARIABLE is 2 :NBRAM .
 7
 8 V0!  CV0!  V0=  etc. are operations on memory locations.
 9 Each takes an address as its only argument.  Byte operation
10 names start with C .  If a truth value is returned, true = -1.
11
12
13
14
15

SCREEN: 22
 0 ( Implementation notes - 5/85       EXIT
 1 I/O bits are addressed by name.  An input bit name returns
 2 true or false.  An output bit name returns an access constant,
 3 which may be used to set, clear, or read the bit.  These
 4 operations are carried out by the words
 5       1bit  0bit  Getbit
 6 respectively.
 7
 8
 9
10
11
12
13
14
15
```

```
SCREEN: 31
 0    ( State sequence documentation                         10/84) EXIT
 1    A state sequence definition is an ordinary : definition
 2 containing the ssXXXX directives to control the execution
 3 sequence during multiple invocations of the definition.
 4 The syntax appears as:
 5 : Name  ...... t/f  Execution_pointer  ssENTRY  ........
 6          ssBEGIN  ..............    t/f  ssUNTIL  ..........
 7          ssBEGIN  ......... t/f  ssUNTIL  .......
 8          ssPAUSE  ssBEGIN  ............ ssUNTIL  ......
 9          ......  ssEND  ;
10 where  .........  stands for the application coce.
11    Code up to ssENTRY is executed on each invocation.  If the
12 ssENTRY argument is true, execution continues.  If it is false
13 execution jumps to after the last ssBEGIN executed during the
14 previous invocation.  ssUNTIL with a false argument causes an
15 EXIT.  With a true argument it is a NOOP.
```

```
SCREEN: 32
 0    ( State sequence documentation                         11/84) EXIT
 1    To simplify application code, ss analogs of AGAIN and
 2 REPEAT are also provided.  The syntax is -
 3    ......... ssBEGIN  ...  t/f  WHILE  ........ ssREPEAT ...
 4    ......... ssBEGIN  .................. ssAGAIN  ;
 5    State sequence directives may be used in combination with
 6 normal conditional execution structures, provided that the
 7 standard nesting rules are observed.  Any conditional control
 8 structure may be totally contained in a single ss clause.
 9 Any ss loop must be totally contained in a control structure
10 clause.
11    The syntax   ssENTRY  ......  t/f
12 IF ssBEGIN .... t/f ssUNTIL ELSE ssBEGIN ... t/f ssUNTIL THEN ..
13 avoids execution of the leading t/f after the first call to the
14 procedure.  The loop chosen is executed until complete.
15
```

```
SCREEN: 33
 0    ( State sequence documentation                         10/84) EXIT
 1    The word ssEND immediately before a ; causes the procedure
 2 to be a NOOP if it is invoked after the sequence is complete.
 3    The word ssPAUSE causes an EXIT, but when the procedure is
 4 executed again, execution will start with the word following
 5 the ssPAUSE.
 6    Typical use of state sequence words is in handling I/O waits
 7 under conditions where it is not practical to devote a task to
 8 the I/O device, and where the complications of a state machine
 9 are unnecessary.
10    Using different pointer variables, a single task can "tend"
11 many devices.  When handling multiple devices of the same
12 type, the ss code will be re-entrant as long as a different
13 pointer variable is used for each device.
14
15
```

```
SCREEN: 34
 0    ( State sequence documentation                       1/85)  EXIT
 1    An endless loop can easily be created with the words
 2 ssBRANCH and ssINIT.  The loop can be arranged so that each
 3 time it is entered, another portion is executed.  The syntax
 4 is
 5    Execution_pointer  ssBRANCH  Proc1  ssPAUSE  Proc2  ssPAUSE
 6        Proc3  ssPAUSE  etc. ..... ssINIT  ;
 7 An external initialization procedure must set the storage
 8 pointer to zero.  ssBRANCH is a NOOP on first entry, but on
 9 subsequent entries, it causes a branch to after the most
10 recently executed ssPAUSE.  ssINIT clears the variable and
11 thus re-initializes the sequence.
12    Other ss words may be used inside of such a sequence.  ssINIT
13 may appear inside of conditional clauses within the procedure,
14 or in a called procedure.
15

SCREEN: 35
 0    ( State sequence documentation                       4/85)  EXIT
 1    In order to provide more compact main sequence words, a
 2 special : word is provided, defined with :ssPROC .  One of these
 3 words can behave as an entire  ssBEGIN  ...  t/f  ssUNTIL
 4 clause.  It "captures" the execution point in the main word in
 5 the same way that ssBEGIN does.  It allows execution to continue
 6 in the main word by executing one of the directives ssCONTINUE
 7 or ssNEXT.
 8    ssCONTINUE has the same effect as  .. true  ssUNTIL ,
 9 exiting immediately from the :ssPROC word, and continuing
10 in the main word without a pause.
11    ssNEXT does not exit immediately from the :ssPROC word, and
12 the main word execution continues only on the next call, just as
13 though it had contained  .. true ssUNTIL  ssPAUSE  .  ssNEXT
14 may appear in a procedure called by the :ssPROC word.
15

SCREEN: 36
 0    ( State sequence definitions                             10/84)
 1
 2 2USER ssCURR
 3 ( t/f  Pointer_addr  ssENTRY  -->  State sequence executed
 4      t  ~  Initial entry
 5      f  ~  Re-entry into state sequence loop   )
 6 CODE ssENTRY   U LHLD   ssCURR UOFFSET W LXI   W DAD
 7      W POP   W' M MOV   H INX   W M MOV  ( Store var. ptr.)
 8      XCHG   W POP   W' A MOV   W ORA   0=
 9      IF  M I' MOV   H INX   M I MOV
10      THEN   NEXT JMP
11 FORTH
12
13
14
15
```

```
SCREEN: 37
 0    ( ssBRANCH  ssINIT                                    12/84)
 1
 2 ( Storage_loc  ssBRANCH  -->  State sequence executed
 3       Variable = 0  ~  Initial entry
 4       Variable = branch_loc  ~  Re-entry  )
 5 CODE ssBRANCH   U LHLD    ssCURR UOFFSET W LXI   W DAD
 6      W POP   W' M MOV    H INX   W M MOV  ( Store var. ptr.)
 7      XCHG   M A MOV    H INX   M ORA   0= NOT
 8      IF  M I MOV   H DCX    M I' MOV  ( Branch)
 9      THEN   NEXT JMP
10 FORTH
11 : ssINIT   ssCURR  @  V0!  ;
12
13
14
15

SCREEN: 38
 0    ( State sequence control                           10/84)
 1
 2 SUBROUTINE ssbegin   U LHLD   ssCURR UOFFSET W LXI   W DAD
 3         M W' MOV   H INX   M W MOV   XCHG
 4         I' M MOV   H INX   I M MOV   RET
 5 CODE ssBEGIN   ssbegin CALL   NEXT JMP
 6 CODE ssPAUSE   ssbegin CALL   ' EXIT JMP
 7 CODE ssUNTIL   W POP   W' A MOV   W ORA
 8      0= IF  ' EXIT JMP  THEN   NEXT JMP
 9 : ssEND   COMPILE  ssBEGIN  ;  IMMEDIATE
10 : ssREPEAT   COMPILE EXIT  [COMPILE] THEN  ;  IMMEDIATE
11 : ssAGAIN   COMPILE EXIT  ;  IMMEDIATE
12 : ssWHILE   [COMPILE] WHILE  ;  IMMEDIATE
13
14
15

SCREEN: 39
 0    ( :ssPROC                                          2/85)
 1 (     :ssPROC -- Define a : procedure which "captures" the
 2    execution point in the calling procedure, just as ssBEGIN
 3    in that procedure would.
 4       Execution in the calling procedure keeps jumping to a
 5    :ssPROC word until the :ssPROC word itself executes
 6    ssNEXT or ssCONTINUE.
 7       :ssPROC words must appear in the procedure containing
 8    ssBRANCH or ssENTRY.  They cannot be called via intermediate
 9    procedures.)
10 : :ssPROC   :  ;CODE  W INX   W PUSH   I DCX   I DCX
11            ssCURR  US) W' MOV   H INX   M W MOV   XCHG
12            I' M MOV   H INX   I M MOV   I POP   NEXT JMP
13 FORTH
14
15
```

```
SCREEN: 40
 0    ( ssCONTINUE  ssNEXT                                 2/85)
 1 ( Use ssCONTINUE and ssNEXT inside of :ssPROC's to control the
 2    execution point of the calling procedure.)
 3
 4 ( ssCONTINUE  -->  Exit from this procedure, continue in
 5       calling procedure without pause.  Must appear in the
 6       :ssPROC word itself, not a called procedure. )
 7 CODE ssCONTINUE   ssCURR US) W' MOV   H INX   M W MOV
 8         XCHG   M I' MOV   H INX   M I MOV   I INX   I INX
 9         NEXT JMP
10 ( ssNEXT  -->  Increment pointer so that next execution of
11      calling procedure will skip this code.  May appear in
12      the :ssPROC word itself or in a called procedure.)
13 : ssNEXT   ssCURR  @  V2+  ;
14
15

SCREEN: 41
 0    ( State sequence tests                               10/84)
 1   EXIT *************
 2 VARIABLE SSSTATE
 3 VARIABLE SSVAR
 4 :ssPROC SScall   200  MS  48 .  @KEY  68  =
 5         IF  ssNEXT  THEN  ;
 6 :ssPROC SsCALL   200  MS  50  .  @KEY  69  =
 7        IF  ssCONTINUE  THEN  ;
 8 : SST   SSVAR  ssBRANCH
 9        SSSTATE  @  2  =
10        IF  ssBEGIN  47  .  200  MS  @KEY  66  =
11            ssUNTIL
12        THEN  ssBEGIN  49  .  200  MS  @KEY  67  =  ssUNTIL
13        SScall  SsCALL  ." Done "   ssEND  ;
14
15

SCREEN: 46
 0    ( Memory checksum computation                        12/84)
 1
 2 ( Start_addr  #bytes  cKsum  -->  8 bit sum, overflow
 3                                      is discarded )
 4 +LB> CODE cKsum   W POP   W' A MOV   A ORA
 5     0= NOT  IF  W INR  THEN   H POP   A XRA
 6     BEGIN
 7        BEGIN  M ADD   H INX   W' DCR  0=  UNTIL  W DCR  0=
 8     UNTIL   APUSH JMP
 9 FORTH
10
11
12
13
14
15
```

```
SCREEN: 47
 0    ( Checksum computation - dynamic                         12/84)
 1 CVARIABLE cScurr    ( Working checksum)
 2 3   :NBRAM cSsave    ( 3 byte ram area - cum. checksums)
 3 : CsSAVE   cSsave  +  cScurr  SWAP  1MOVE  ;
 4 VARIABLE cSaddress    ( Working address)
 5 VARIABLE cSlast
 6 VARIABLE cHECKSUM-POINTER    ( Execution pointer - +checksum)
 7 : cSlength   cSlast  @  1+  cSaddress  @  -  64  MIN  ;
 8 ( 1sCHECK  -->  Next 64 bytes, or rest of prom, summed)
 9 : 1sCHECK   cSlength  cSaddress  @  OVER  cKsum  cScurr  C+!
10                cSaddress  +!  ;
11 : PsDONE   cSaddress  @  cSlast  @  -  0>  ;
12 ( Start_addr  last_addr  PsSTART)
13 : PsSTART   cSlast  !  cSaddress  !  ;
14
15

SCREEN: 48
 0    ( Checksum computation - dynamic)                        12/84)
 1
 2 : +checksum   cHECKSUM-POINTER
 3      ssBRANCH
 4        cScurr  CV0!  IH> 100  IH> 1FFF  PsSTART
 5        ssBEGIN  1sCHECK  PsDONE  ssUNTIL  0  CsSAVE
 6        IH> 2000  IH> 3FFF  PsSTART
 7        ssBEGIN  1sCHECK  PsDONE  ssUNTIL  1  CsSAVE
 8        IH> 4000  IH> 5FFF  PsSTART
 9        ssBEGIN  1sCHECK  PsDONE  ssUNTIL  2  CsSAVE
10        cScurr  CV0NOT  IF  6  !kFLAG  THEN
11      ssINIT  ;
12
13
14
15

SCREEN: 90
 0 ( Storage & constants)                        5/85)  EXIT
 1 CVARIABLE mBUSY   ( Motor status)
 2 VARIABLE mCpointer  ( Mprocess execution pointer storage)
 3 ( External command storage)
 4    VARIABLE PGTIME    ( # of clock ticks to run motor)
 5    VARIABLE mDIRECTION ( Commanded direction - Positive ~ In )
 6    CVARIABLE mCMDFLAG ( Non-zero ~ New command available)
 7 ( Local copies of command data)
 8    VARIABLE pGtIME
 9    VARIABLE mdirection
10    CVARIABLE mCmdflag
11 mSpTIME  and  mRvTIME  are the # of clock ticks for operation
12    of the stop and reverse relays.
13 mSpDYNAMIC is the # of timer ticks for reverse plugging of the
14    motor when it has reached a physical limit - provides
15    dynamic braking.
```

```
SCREEN: 91
 0 ( Timer operations                         5/85)  EXIT
 1    A timer is a named ram area.  When executed, a timer name
 2 places a pointer to the ram area in a common working variable.
 3 Timer operation words all operate on the ram pointed to by
 4 this variable - that is, on the most recently executed timer
 5 object.
 6    All timer operations use ticks of the real-time clock as the
 7 unit, to save computation.
 8   n  TWSTART  -->  Timer started for n ticks of running
 9      TWCANCEL  -->  Timer stopped
10      TIMEOUT  -->  t/f  --  t ~ timed out or not running
11 TIMEOUT actually "runs" the timer, stopping it if the clock
12 has passed the alarm time.  Timing accuracy is no better
13 than the repetition rate at which TIMEOUT is executed.
14
15

SCREEN: 92
 0 ( Action of words for which code is not shown)   EXIT
 1 mIlm  -->  t/f  --  Is motor at inner limit?
 2 mOlm  -->  t/f  --  Is motor at outer limit?
 3 mRUN  mREV  --  I/O bits which control the run and reverse
 4                 relays.  (See implementation notes)
 5 >mlimit  -- Set the vector for mLIMIT to sense the limit
 6             switch corresponding to the mREV setting.
 7 mLIMIT  -->  t/f  -- Is motor at limit expected for present
 8                      motion direction?
 9
10
11
12
13
14
15

SCREEN: 93
 0 ( Action of words for which code is not shown)  EXIT
 1 -mCLIMIT  -->  f ~ Current command would cause immediate limit
 2                    flag if executed.
 3                t ~ Current command would not cause limit flag
 4 ?cSTOP  --  If the new command requires a reversal or if
 5             mLIMIT is true, and the motor is running, stop
 6             the motor and start the operation timer to
 7             allow relay oepration time.
 8 ?cREVERSE -- If the new command requires a reversal, reverse
 9              the motor and start the operation timer to
10              allow relay operation time.
11 mSTART-ACCOUNTING, mSTOP-ACCOUNTING --  Procedures which tally
12   motion of a motor, to maintain a pseudo-position value,
13   measured in timer ticks.
14
15
```

```
SCREEN: 94
 0    ( Basic operations                                5/85)
 1
 2 ( Start the motor, do necessary accounting )
 3 : mSTART   mRUN  1bit  mSTART-ACCOUNTING
 4         mToTIMER  mToTIME  TWSTART  ;
 5
 6 ( Start the motor if not already running, update plug time)
 7 : ?mSTART   mRUN  Getbit  0=  IF  mSTART  THEN
 8      mPgTIME  @  ABS  mTIMER  TWSTART  ;
 9
10 ( Wait for the motor operation timer)
11 :ssPROC mTmWAIT   mTIMER  TIMEOUT  IF  ssCONTINUE  THEN  ;
12
13
14
15

SCREEN: 95
 0    ( Basic operations                                   5/85)
 1 ( Issue a local stop command to Mprocess, not affecting any
 2    available external command.)
 3 : mSTOP-LOCAL   mCmdflag  CV1!  mPgTIME  V0!  ;
 4
 5 ( Stop the motor if it is running, doing all housekeeping
 6     except for mSTOP-ACCOUNTING)
 7 : mSTOP   mRUN  Getbit
 8     IF  mRUN  0bit  mToTIMER  TWCANCEL
 9        mTIMER  mSpTIME  TWSTART
10     THEN  ;
11
12 ( Stop the motor if it is running)
13 : ?mSTOP   mRUN  Getbit  IF  mSTOP-ACCOUNTING  mSTOP  THEN  ;
14
15

SCREEN: 96
 0    ( Basic operations                                      5/85)
 1
 2 ( Plug the motor for mSpDYNAMIC ticks)
 3 : sPDYNAMIC   mSpDYNAMIC
 4      IF  mRUN  1bit  mTIMER  mSpDYNAMIC  TWSTART  THEN  ;
 5
 6 ( Reverse the motor & set the limit vector for the expected
 7    limit)
 8 : mREVERSE   mREV  bTOG  >mlimit  mTIMER  mRvTIME  TWSTART  ;
 9
10
11
12
13
14
15
```

```
SCREEN: 97
 0    ( Error checks                                        10/84)
 1
 2 : !mERROR   mEFLAG  @  SWAP  mEFLAG  !  0=
 3          IF  mStop_local  THEN  ;
 4
 5 : ?lm-ERROR   mOlm  mIlm  and  IF  -1  !mERROR  THEN  ;
 6
 7 : lmERRORclear   mEFLAG  @  -1  =
 8       IF  mIlm  mOlm  and  NOT  IF  mINIT  THEN  THEN  ;
 9
10 ;ssPROC mEPROCESS   mEFLAG  V0=  IF  ssCONTINUE  THEN
11         ?mstop  mTIMER  TIMEOUT  IF  lmERRORclear  THEN;
12
13
14
15

SCREEN: 98
 0    ( Sequence & command support                         2/85)
 1
 2 ( ?mSEQnew --> t/f  Pointer_addr  ---  args for ssENTRY  )
 3 : ?mSEQnew   mCmdflag  C@  mCmdflag  CV0!  mCpointer  ;
 4
 5 ( Process remote command.  t/f  ~  command present)
 6 : ?McOMMAND   0  mCMDFLAG  C@
 7      IF  1-  mCMDFLAG  CV0!  mCmdflag  CV1!
 8          PGTIME  mPgTIME  2MOVE  mDIRECTION  mdirection  2MOVE
 9      THEN  ;
10
11 ( Process remote command.  If present, make Mprocess recycle)
12 :ssPROC ?mCOMMAND   ?McOMMAND
13        IF  ssNEXT  ELSE  ssCONTINUE  THEN  ;
14
15

SCREEN: 99
 0    ( Motor-idle  MRun-Proc                           2/85)
 1
 2 ( Idle until an error or a command occurs)
 3 :ssPROC MOTOR-IDLE   ?lm-ERROR
 4     mEFLAG  CV0NOT  IF  ssNEXT  EXIT  THEN
 5     ?McOMMAND  IF  ssCONTINUE  THEN  ;
 6
 7 ( Run until   LIMIT  or  mTIMER timeout     ~  Stop
 8               mToTIMER timeout              ~  Error
 9               Command                       ~  Obey it  )
10 :ssPROC mRUN-PROC   mToTIMER  TIMEOUT
11        IF  -2  !mERROR  ssCONTINUE  THEN
12        mTIMER  TIMEOUT  mLIMIT  OR
13        IF  mSTOP_LOCAL  ssCONTINUE  THEN
14        ?McOMMAND  IF  ssCONTINUE  THEN  ;
15
```

```
SCREEN: 100
 0    ( Mprocess                                    2/85)
 1 : Mprocess
 2     BEGIN  ?mSEQnew  ssENTRY  mEPROCESS
 3       mTIMER  TWCANCEL  mPgTIME  V0=
 4       IF  ?mSTOP  mTmWAIT  ?mCOMMAND  mLIMIT
 5         IF  mREVERSE  mTmWAIT  sPDYNAMIC  mTmWAIT
 6             mSTOP  mTmWAIT
 7         THEN  mBUSY  CV0!  MOTOR-IDLE
 8       ELSE  -mCLIMIT
 9         IF  ?cSTOP  mTmWAIT  ?mCOMMAND  ?cREVERSE  mTmWAIT
10             ?mSTART  mRUN-PROC
11         ELSE  mSTOP_LOCAL
12         THEN
13       THEN
14     AGAIN  ;
15
```