Automatic State Table Generation

The HP C1553A DDS tape autoloader requires a complex sequence of simple operations to carry out mechanical retries. These sequences are defined in tables. Cadre's Teamwork was used for input and an automatic tool was used to generate the tables to go in ROM.

by Mark J. Simms

The autoloader firmware for the HP C1553A digital audio tape autoloader was written in the C programming language to run on a Hitachi H8/325 processor. This processor is an embedded system microcontroller with built in ROM, RAM, I/O ports, timers, and serial ports. This allows a very low-cost implementation of the autochanger controller in which most of the functions can be carried out in a single chip. However, the largest ROM size available on the H8 series of processors was 32K bytes. This means that the complex retry algorithms required for controlling such a mechanical device needed to be implemented in as compact a manner as possible.

Our laboratory has a large amount of experience in producing table-driven systems. All of our products have had some form of table-driven control structures in some part of their firmware. However, experience had shown that there can be severe problems maintaining table-driven code because of the difficulty of maintaining the tables. This derives from the lack of readability of software written in C or assembly language that merely defines the contents of data structures. A lot of documentation needs to be added to the source code to explain the meaning of the entries. If this is not maintained, then the declarations rapidly become unreadable. This greatly increases both the time needed to implement changes and the risk of errors.

The designers of the HP 9145A cartridge tape drive and the HP 35470A DDS tape drive attempted to improve this situation by defining state machine languages that can be translated into C source code automatically. These languages offered powerful constructs for defining the tables in terms of state machines. The software would remain in one state until an event was detected. Then a set of actions would be carried out and a new state entered. This approach made the table definition much more readable than the basic data declarations and greatly alleviated the maintenance problems. However, the state machine languages suffered from many of the problems that are characteristic of "unstructured" programming techniques. There was no observable flow in the source code since transitions were permitted between any two states. This made it very difficult to follow the flow of the program and determine what sequence of actions had occurred.

To aid in documenting these state machines, the Teamwork structured analysis tool from Cadre Technologies was adopted. This allowed the initial problem analysis to be carried out graphically. A state transition diagram was produced to document the desired solution (see Fig. 1). This was then implemented using the state machine language. However, as the state machine language description was modified, the diagram gradually became more and more out of date and was updated periodically. This meant that while the diagram could be used to gain an initial familiarity with the software, it could never be guaranteed to be completely accurate.

With the HP C1553A autoloader, these problems became more serious. The state tables were to be used very extensively for mechanical control. Also, there was a very strong need to communicate the control algorithms to mechanical engineers and the product test team. This required that good accurate documentation be available to all within the division. It was felt that any manual system for maintaining such documentation would prove unusable in real situations. As a result, a decision was made to generate the state tables directly from the Teamwork diagrams. This would ensure that the diagrams were always accurate reflections of the software.

Design Implications

Analysis of the HP C1553A motor control software showed that the software divides into two major sections. The first is a number of routines that handle the low-level control of the mechanism. These routines control the motors and solenoids, read sensors, and track the position of the mechanism. They track control information and map that onto the control signals required to operate the motors in the correct direction at the correct power level. They debounce input readings and map them into mechanism position information. These routines are implemented in C and use global variables to interface with the rest of the software. The routines are called in sequence to carry out all the necessary interfacing to the mechanism.

The second section is the control sequencing. This contains a number of state machines. Some of these are directly linked to the individual mechanism parts. Others sequence individual mechanism operations together in response to a single command. These state machines interface with the low-level routines by means of the control information and mechanism position global variables.

Since several state machines are running concurrently with the low-level routines, this concurrency must be reflected in the software. Each state machine, when called, checks to see if a transition needs to be made. If not, control is returned immediately. If a transition is needed, that transition is executed and control is returned. This ensures that all the state

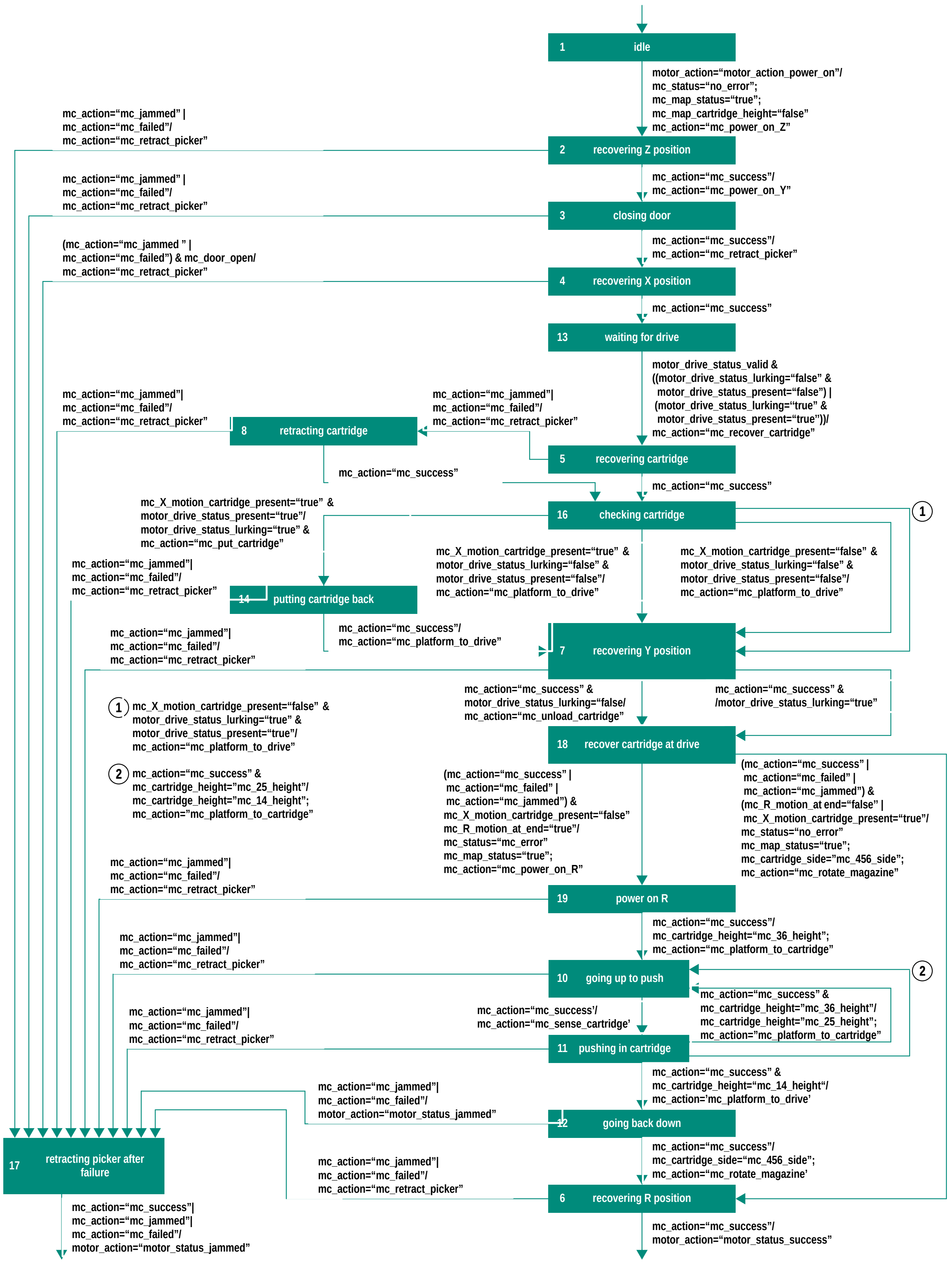
1 idle
motor_action="motor_action_power_on"/ mc_status="no_error"; mc_map_status="true"; mc_map_cartridge_height="false" mc_action="mc_power_on_Z"
mc_action="mc_jammed" | mc_action="mc_failed"/ mc_action="mc_retract_picker"
2 recovering Z position
mc_action="mc_jammed" | mc_action="mc_failed"/ mc_action="mc_retract_picker"
mc_action="mc_success"/ mc_action="mc_power_on_Y"
3 closing door
(mc_action="mc_jammed " | mc_action="mc_failed") & mc_door_open/ mc_action="mc_retract_picker"
mc_action="mc_success"/ mc_action="mc_retract_picker"
4 recovering X position
mc_action="mc_success"
13 waiting for drive
motor_drive_status_valid & ((motor_drive_status_lurking="false" & motor_drive_status_present="false") | (motor_drive_status_lurking="true" & motor_drive_status_present="true"))/ mc_action="mc_recover_cartridge"
mc_action="mc_jammed"| mc_action="mc_failed"/ mc_action="mc_retract_picker"
8 retracting cartridge
mc_action="mc_jammed"| mc_action="mc_failed"/ mc_action="mc_retract_picker"
5 recovering cartridge
mc_action="mc_success"
mc_action="mc_success"
16 checking cartridge
1
mc_X_motion_cartridge_present="true" & motor_drive_status_present="true"/ motor_drive_status_lurking="true" & mc_action="mc_put_cartridge"
mc_action="mc_jammed"| mc_action="mc_failed"/ mc_action="mc_retract_picker"
mc_X_motion_cartridge_present="true" & motor_drive_status_lurking="false" & motor_drive_status_present="false"/ mc_action="mc_platform_to_drive"
mc_X_motion_cartridge_present="false" & motor_drive_status_lurking="false" & motor_drive_status_present="false"/ mc_action="mc_platform_to_drive"
14 putting cartridge back
mc_action="mc_jammed"| mc_action="mc_failed"/ mc_action="mc_retract_picker"
mc_action="mc_success"/ mc_action="mc_platform_to_drive"
7 recovering Y position
1 mc_X_motion_cartridge_present="false" & motor_drive_status_lurking="true" & motor_drive_status_present="true"/ mc_action="mc_platform_to_drive"
2 mc_action="mc_success" & mc_cartridge_height="mc_25_height"/ mc_cartridge_height="mc_14_height"; mc_action="mc_platform_to_cartridge"
mc_action="mc_success" & motor_drive_status_lurking="false/ mc_action="mc_unload_cartridge"
mc_action="mc_success" & /motor_drive_status_lurking="true"
18 recover cartridge at drive
(mc_action="mc_success" | mc_action="mc_failed" | mc_action="mc_jammed") & mc_X_motion_cartridge_present="false" mc_R_motion_at_end="true"/ mc_status="mc_error" mc_map_status="true"; mc_action="mc_power_on_R"
(mc_action="mc_success" | mc_action="mc_failed" | mc_action="mc_jammed") & (mc_R_motion_at end="false" | mc_X_motion_cartridge_present="true"/ mc_status="no_error" mc_map_status="true"; mc_cartridge_side="mc_456_side"; mc_action="mc_rotate_magazine"
mc_action="mc_jammed"| mc_action="mc_failed"/ mc_action="mc_retract_picker"
19 power on R
mc_action="mc_success"/ mc_cartridge_height="mc_36_height"; mc_action="mc_platform_to_cartridge"
mc_action="mc_jammed"| mc_action="mc_failed"/ mc_action="mc_retract_picker"
10 going up to push
2
mc_action="mc_success" & mc_cartridge_height="mc_36_height"/ mc_cartridge_height="mc_25_height"; mc_action="mc_platform_to_cartridge"
mc_action="mc_jammed"| mc_action="mc_failed"/ mc_action="mc_retract_picker"
mc_action="mc_success'/ mc_action="mc_sense_cartridge'
11 pushing in cartridge
mc_action="mc_success" & mc_cartridge_height="mc_14_height"/ mc_action='mc_platform_to_drive'
mc_action="mc_jammed"| mc_action="mc_failed"/ motor_action="motor_status_jammed"
12 going back down
mc_action="mc_success"/ mc_cartridge_side="mc_456_side"; mc_action="mc_rotate_magazine'
17 retracting picker after failure
mc_action="mc_jammed"| mc_action="mc_failed"/ mc_action="mc_retract_picker"
6 recovering R position
mc_action="mc_success"| mc_action="mc_jammed"| mc_action="mc_failed"/ motor_action="motor_status_jammed"
mc_action="mc_success"/ motor_action="motor_status_success"

Fig. 1. State transition diagram.

machines and low-level routines can be called in sequence, thereby maintaining the required concurrency.

From this analysis, the following design criteria were derived for the state machine implementation:

- The state machines must be able to respond to the values of mechanism position variables and execute transitions accordingly.
- The state machines must be able to set mechanism control variables when a transition occurs.
- A timeout mechanism is required that can handle times up to 30 s with a resolution of 1 ms.
- Each state machine must execute at most one transition when executed.
- Each transition must be executed as a complete unit to lessen the risk of data contention problems.
- The state machine implementation must use the minimum space possible.
- The ability to store a history of trace information must be provided for debugging purposes.

Interpreting Teamwork/RT

The Teamwork/RT product provides a graphical state machine editor that has the following features:

- There is a set of states, each of which has a unique number and a unique name.
- There is an initial state, indicated by a single initial transition.
- Transitions out of states may enter other states or may indicate that the state machine has exited.
- Transition information indicates the condition under which the transition occurs and may give actions that are to be carried out on that transition.
- Each transition condition is a logical expression consisting of a number of comparisons of variables with values.
- Each transition action is an assignment of a value to a variable.

The full syntax of this state machine description is supported by the code generator tools. This gives a fairly rich design environment in which to work. It has the additional advantage that if the diagram is correct according to the Teamwork syntax checker, it should generate code correctly and that code should compile.

To parse the state machine diagrams, a program was written to access the Teamwork database. This retrieves the data structures for a complete diagram, parses them, and generates the required code table. The program connects to the Teamwork database, retrieves the state transition diagram, and follows the linked list of states. For each state, it identifies each transition that exits that state. For each transition, it parses the associated text and generates the required data structures for its condition, actions, and next state.

Finding the start and end states of a transition and finding the transition information that is bound to a given transition involves the concept of an instance number. Each item in a Teamwork diagram is given a number to identify it. This number is unique across all items in the given diagram, whether an item is a text block, a transition, or a state.

The instance numbers of the initial and final states and the text block associated with a given transition are stored in that transition's entry in the linked list of transitions. The state can be identified by searching through the linked list of states to find the one with the same instance number as in the transition entry. The text block holding the transition information can be identified by searching through the linked list of text blocks to find the one with the same instance number.

To increase the performance of the code generator, an array of pointers to text blocks and states is set up at the start of the program. The list of text blocks is searched and the entry in the array indexed by the instance number of a given block of text is set to point at that block of text. The list of states is searched and the entry in the array indexed by the instance number of a given state is set to point to that state's entry. This allows the start state, end state, and text block associated with a given transition to be found by a single table look-up each.

Parsing Transition Information

The transition information associated with a transition consists of an event or an event, a / character, and a semicolon-separated list of actions.

The event is a logical expression consisting of a number of comparisons. Comparisons are linked with logical OR operations indicated by the | character and logical AND operations indicated by the & character. The logical NOT operator, indicated by the ~ character, can be used to invert an expression or comparison. Order of execution can be indicated by the use of parentheses. Each comparison consists of either just a variable or a variable, a comparator symbol, and a constant value in double quotation marks. The comparators can be equal =, greater than >, less than <, not equal ~=, less than or equal to <=, or greater than or equal to >=.

Each action consists of the name of a variable, an equals sign =, and the value to be assigned to that variable in double quotation marks.

This gives a text block of the form:

```
mc_timed_out |
( mc_jam_sense &
  mc_picker_state="mc_picker_open" ) /
mc_X_motion_direction="mc_X_brake";
mc_action="mc_jammed"
```

This should be read as follows: If mc_timed_out is true, or both mc_jam_sense is true and mc_picker_state is mc_picker_open, then set mc_X_motion_direction to the constant value mc_X_brake and set mc_action to the constant value mc_jammed and transition to the next state.

This text is parsed using a parser written using the yacc and lex tools provided with the HP-UX* operating system. this generates a reverse Polish form for the logical expression along with the values required for the actions.

Generating the State Table

The major issue with the state table design was to implement the full syntax supplied by the Teamwork/RT state transition diagrams in as compact a form as possible. Performance was not an issue. The maximum response time required of the firmware was of the order of 3 ms. This was easily achievable with the designs chosen.

To produce a machine readable form of the transition information, each part is taken in turn.

To reference variables, the H8 processor uses 16-bit addressing. This can be truncated to 10 bits to limit the address space to the 1024 bytes of RAM available on the processor. Since there are over 3000 references to less than 50 variables, an index table is far more efficient. In the transition information, a six-bit index value is stored. This is used to look up the address of the variable in an array of pointers.

Since every variable in the logical expression is associated with a comparator, it would be useful to store the information indicating the comparator in the spare two bits with the variable index. Unfortunately, there are six comparators, which would require at least three bits to store. However, the comparators form pairs of inverses. Equal is the opposite of not equal, greater than is the opposite of less than or equal to and less than is the opposite of greater than or equal to. As such, the logical NOT operation is used so that only the three basic comparators have to be used in the state table, allowing the comparator to fit in the two spare bits of the variable index. Since most comparisons are equalities, this saves a great deal of space while preserving the simple table format with no items crossing byte boundaries. All comparisons are with 8-bit values. These are stored in the byte after the variable index and operator.

The logical operators are stored in single-byte values. This is wasteful since only a couple of bits are really required for these. However, it maintains the simplicity of the generator and interpreter by avoiding the necessity of table items crossing byte boundaries.

The expression is stored in the table in reverse Polish format with the comparisons as values. The expression is terminated with a zero byte to indicate where calculation should stop.

Each action consists of the six-bit index for the variable stored in one byte and the single-byte value to be assigned stored in the next byte. The list of actions is terminated by a zero value to indicate the end of the list.

While most of the actions involve assignments of a single byte, the setting up of timeouts requires that a 16-bit value for the timeout in milliseconds be set. This variable is considered a special case by the state machine generator and two assignments are generated. This maintains the readability of the state machine diagram without adding complexity to the state machine table to handle 16-bit values.

The final part of the transition information is the next state. This is given as a single-byte value.

The state table entries for the transition text example given earlier are:

```
( mc_timed_out_index | 128 ), 1, /* == */
( mc_jam_sense_index | 128 ), 1, /* == */
( mc_picker_state_index | 128 ), mc_picker_open, /* == */
1, /* AND */
2, /* OR */
0,
mc_X_motion_direction_index, mc_X_brake,
mc_action_index, mc_jammed,
0,
0,
```

The transition data for each state is generated in turn. At the end of the transitions out of a given state, a single byte of value 255 is inserted to indicate the end of the transitions associated with that state.

To access the transitions associated with a given state, a table of pointers is used. The pointer indexed by a given state number points to the first transition from that state.

A single byte of data is used to hold the number of the current state. This is the same value as appears on the Teamwork/RT diagram.

Finally, a structure is generated that holds a pointer to the state pointer table, a pointer to the index variable, a pointer to the state variable, and a code to be used for logging.

The complete data structures for the state table are shown in Fig. 2.

State Table Interpreter

To execute the state table an interpreter routine is used. This reads the state table and carries out the actions required.

The interpreter routine is passed a pointer to the header structure for the state machine to be executed. It uses the pointer to the state variable to get the current state. It uses this as an index into the state pointer table to get a pointer to the transitions for the current state. It then scans the transition information.

The first thing in the transition information must be an expression, terminated by a zero byte. The interpreter routine checks each byte in turn to see if it is a comparison, an operation, or the terminating zero.

If either or both of the top bits are set, then it is a comparison and the next byte is the value to be compared. The bottom six bits of the byte are used as an index into the index table to get the pointer to the variable to be checked. This is then used to get the variable's value. The first two bits are used to determine whether the comparison should be for equality, greater than, or less than. The next byte is read, the comparison is executed, and the result is placed on a stack. The current byte is then incremented past both bytes of the comparison.

If neither of the top two bits is set but the value is not zero, then the byte indicates a logical operator. If the operator is AND or OR, then the top two values are removed from the stack, the operation is carried out on them, and the result is pushed back on the stack. If the operator is NOT, then the top value is pulled off the stack, inverted, and pushed back on. The current byte is then incremented past the operator.

This is repeated until a zero byte is found as the current byte. Then the top value is pulled off the stack to indicate whether the expression evaluated to TRUE or not.

If the expression is TRUE, the actions are read in turn as byte pairs until a zero byte is found as the first byte. For each byte pair, the value in the second byte is assigned to the variable pointed to by the pointer in the index table indicated by the index in the first byte. When the terminating zero byte is found, the next byte is assigned to the state variable and the interpreter routine exits, having completed a transition.

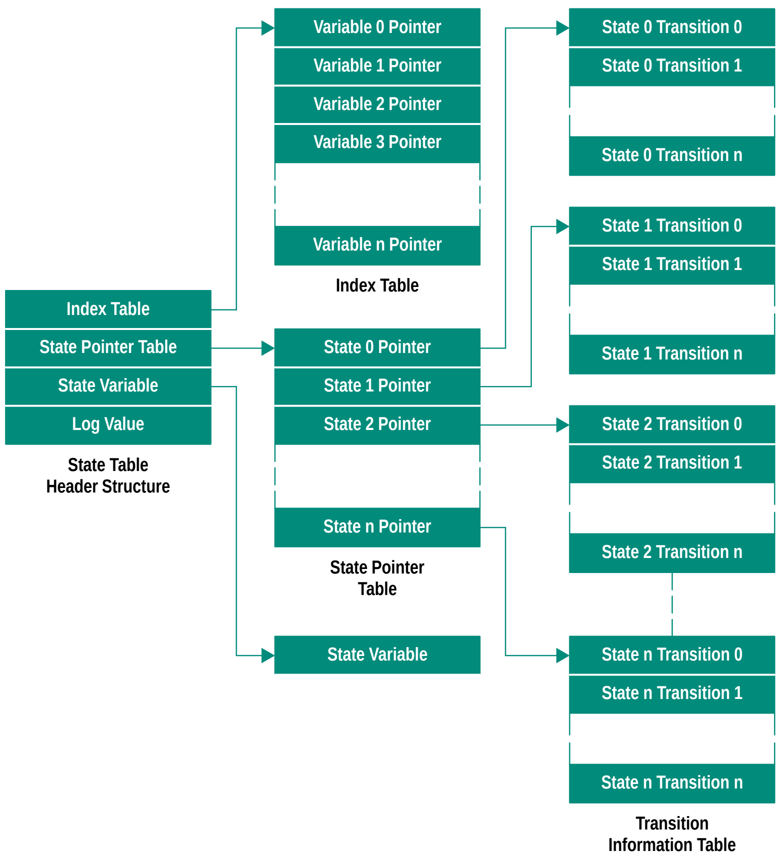

Fig. 2. State table data structures.

If the expression evaluates to FALSE, the actions are skipped over until a zero index is found. The next byte is skipped as well, since this is the next state value. The subsequent byte is checked to see if there are any more transitions to process. If this byte is not 255, then another transition follows and it is processed in the same manner. If this byte is 255, then all the transitions have been processed and none have conditions that are TRUE. The interpreter routine exits without carrying out any actions or altering the state variable.

Initialization and Exception Conditions

The state variable must be initialized at startup for the initial state of the system. Rather than provide a mechanism for the initialization routines to know the initial state, an additional state 0 is added to the state machine. Any transitions on the diagram that connect off-page are viewed as connecting to this additional state. Since the initial transition that identifies the initial state is the only one that can come from off-page, this allows the initial state to be set merely by setting it to zero. As soon as the state machine is executed, it will transition into the initial state on the diagram.

When an exception occurs, it is useful to be able to restart the state machine to initialize those areas of the mechanism under its control. Rather than connect all the transitions that handle exception conditions to the initial state, these are allowed to run off-page. In the Teamwork/RT notation, this means that the state machine has exited and that it should be restarted from the initial state on its next invocation. Since these off-page connectors connect to the additional state 0, this has exactly the desired effect. The exception transition can then be made to restart the state machine without the clutter of unnecessary connections on the diagram.

Debugging and Trace Logging

To be able to debug problems with a real-time control system, it is important to be able to get trace information back from the unit after a failure to determine what the system was doing at the time of failure. With a system that is largely implemented as state tables, it is possible to follow the flow of actions in terms of the state changes involved. As such, the state table interpreter was designed with a tracing function built in.

Whenever a state change occurs, the state table interpreter logs the current time as a 16-bit rolling clock counting in milliseconds, the 8-bit log value in the state table header structure that identifies which state machine is being executed, and the 8-bit value of the new state. The resultant 32-bit value is stored in an internal rolling buffer and is also transmitted on one of the H8 processor's built-in serial lines.

To decode this trace information, host-based interpreter programs are used. These decode the information in the trace log to identify exactly which state table and state within the table are involved. These are then printed out using the names on the Teamwork/RT state transition diagrams. This enables the changes in state to be followed on the diagrams merely by following the names given. The time each state was entered is listed alongside the name of the state to facilitate the interpretation of the mechanical factors that may have caused the state change. The use of milliseconds throughout, both for timeouts and for trace logging, allows engineers debugging failures to work in real-world values rather than units that are solely dependent on the software design.

During product test, the serial output from the microprocessor is monitored and the data is interpreted in real time. This allows both real-time debugging of failures and the gathering of a complete history of a test. In the field, the rolling buffer is returned from the autochanger via its SCSI interface. This only allows a recent history to be returned, but does not require additional hardware to monitor the serial port.