

Description**BACKGROUND OF THE INVENTION****1. Field of the Invention**

This invention relates to a programmable controller for processing finite state applications programs which control the states of operation of machines and/or processes. The invention also relates to debugging a machine and/or process which allows for easy diagnostics of the finite state applications program.

2. The Prior Art

In the 1960's, programmable controllers were developed to replace relay control systems which used electromagnetic relays to control the operation of industrial machines or processes. These programmable controllers were designed to work as if they contained relay circuits, even though the circuits were actually programs in a computer. The relay circuit designs, emulated by these programmable controllers, were called ladder diagrams because of their general appearance. Ladder diagrams are excellent for designing combination logic control problems, where outputs or actions are directly dependent on the states of inputs or conditions. However, for sequential control problems, where the control actions are time-dependent, the ladder diagram approach becomes cumbersome, difficult to design and fault-find. Lloyd, "Graphical Function Charts Programming For programmable Controllers,"

Control Engineering (Oct 1985).

In order to emulate ladder diagrams, a programmer strings together long lists of Boolean equations in the form of software commands either in text form or graphically. Each command controls an internal or external signal. Debugging or fault-finding in such a system usually involves observing an unexpected condition, and then intuitively searching back through some or all possible combinations of factors that could have caused the unexpected condition. Because all of the ladder diagram is scanned each scan cycle it is not possible to eliminate any part as definitely not containing the problem.

Another drawback of the ladder diagram controllers is that in order to determine the causes of the unexpected condition it is desirable to record information related to the cause of the problem. However the ladder diagram system does not have any mechanism to differentiate between relevant and irrelevant changes in system variables and conditions, and this leads to the need to store an impractically large amount of data which would be very difficult to analyse and sort.

Furthermore, ladder diagram controllers are cumbersome to program. By their very nature, ladder diagrams are difficult to understand. Additionally ladder diagrams do not clearly show the broader, macro functionality they are designed to provide, particularly sequential functions. What they show is in effect a circuit that provides that functionality. For many years, machine design and manufacturing engineers, who were unfamiliar with ladder diagrams, could not easily develop control systems for their machines or processes. Instead, they relied on specialised control engineers to program their control systems.

In an effort to simplify programmable controllers, systems were developed which enable engineers to use high level languages to emulate "state diagrams." A state diagram is a graphical representation of the states of operation and the conditions which change the status of the states of operation in a machine or process. In graphical form, a state diagram is a series of nodes with interconnecting arcs. Each node represents a particular state of operation and each connecting arc represents a transition function which causes the status of the state to change. State diagrams are easy to understand and easy to design by all types of engineers. The current systems which emulate state diagrams, however, are difficult to program, and difficult to debug.

An example of one such system is disclosed in United States Patent 4, 562,529 to Drummond. Drummond discusses a method and apparatus for controlling the states of operation of a machine or process by using "state logic variables." A "state logical variable" uniquely represents each operating state of the machine or process. The "state logical variable" can only be one of two values; either TRUE or FALSE. TRUE meaning that the state is active and FALSE meaning that the state is inactive. By using the methods as disclosed the Drummond system operates very similarly to the ladder diagram controllers. Several problems are present in the Drummond method and apparatus.

First, the entire finite state applications program must be cyclically scanned in order for the Drummond system to properly process the "state logical variables". To make assignments to state variables, which is necessary to make changes of state, or to carry out state dependent actions Drummond must evaluate many expressions containing state variables. His disclosure discusses simplified ways for doing this, however the process is still inefficient and can be eliminated altogether.

Second, programming the Drummond controller to emulate state diagrams is a cumbersome process. The language disclosed in Drummond does not lend itself for easily converting the state diagrams into program code. This occurs because the condition statements which effect the status of the "state logic variables" cannot be lumped into specific areas of program code and require explicit inclusion of state variable terms on the left-hand sides. Additionally, virtually every statement in the code requires a "state logical variable" to be assessed, in order to determine whether a state assignment should be made or whether an action should be carried out. As a result, much extra code must be drafted in order to perform simple functions.

Lastly, because every statement in the finite state applications program must be scanned each scan in order to evaluate the status of the state logical variables, the system operates very inefficiently. This drawback severely limits throughput and limits the recording of system activity information for debug purposes. Also, Drummond does not disclose an adequate strategy or structure for keeping track of the multiple operating tasks to be controlled simultaneously.

Note should also be taken of the commercially available GRAFCET systems based on French standard NF C 03-190. GRAFCET does not restrict a state diagram to only having one active state at one time and so knowing the identity of one active state in a GRAFCET state diagram does not totally define the conditions and operations associated with that state diagram at that time. This has significant disadvantages during program debugging.

Thus programmable controllers have been programmed by a means known as ladder diagrams, or an equivalent means, from the early days. As more and more functionality has been added to the language, (in the form of available ladder diagram symbols and structure), it has become apparent that there are limitations to the ladder diagram that can only be addressed by a change to the basic approach to programming. This is particularly apparent when programmable controllers are used for the control of machinery in flexible automation schemes. The functioning of such schemes relies significantly on the ability of the controller to sequence the machinery in a complex and flexible way with regard to both the activities carried out and the functions that step the sequence on and with regard to the ability of the programmable controller to control many essentially independent sequential processes at the same time. Such things can, and are done with ladder diagram programmable controllers. However such programmes require a translation process to generate a ladder diagram from the definition of the sequencing, which may be in the form of a written specification or a flow chart. When the ladder diagram is inspected, the functions carried out are not obvious and the program and the process of debugging the program are cumbersome. Much of the information inherent in the spec or flow chart is lost or obscured in the translation process which converts the functional requirements into a relay circuit that fulfils those requirements.

Over the last few years there has been much interest in improving this situation. Sequencing methods have been developed which aim at improving the generation, debugging, and comprehension of such programs. The French GRAFCET effort has been notable and this has been accepted internationally as a useful technique. Related French and West German standards exist. Several programmable controllers using variations of the GRAFCET methods of describing sequences using standardised flow charts are on the market. However, there are further issues to address to reach an optimum form of control systems programming and these are not addressed by any currently known system.

The problems that need addressing call for improvements in

- program structure and form
 - to focus on the relevant
 - to simplify understanding of complex systems
 - to suit to high levels of functionality
- for minimisation of transformations
- for avoidance of side effects
- to reduce cost of hardware
- being self explanatory to widest range of people and user friendly
- minimisation of training needs
- maximisation of assistance with debugging

The present invention seeks to provide a control method and apparatus that reduces or eliminates the disadvantages referred to above, and is more readily and freely used by control, manufacturing and design engineers.

The following is a preliminary explanation of terms which will assist the reader to more fully comprehend the nature of the invention that is described in this specification.

1. "State(s) of operation" means a particular combination of conditions and operations which exist for a finite time interval in a machine or process; only a subset of the machine's or process' signals or values are at particular defined values. The "state of operation" of a machine or process can be thought of as being a particular task or subset of tasks presently performed by the machine or process.

2. "Controller" means a device for exercising control over a process or machine or activity, which includes means for discerning the "state of operation." For this purpose, the controller must be kept informed of variations in the values and the status of the machine or process.

3. "Program state" means a particular block of one or more statements of program code used for evaluating, maintaining and controlling a "state of operation."

4. The terms "machine" or "process" are systems which proceed in real time from one specific operating state to a next operating state. The controller reads signals from the machine or process, monitors the operation of the machine or process, tracks the necessary operating states by way of its own internal states, and provides necessary output signals to put the machine into the necessary operating states based on the algorithms embedded in the applications program.

5. The term multitasking means a method of asynchronously running separate programs or separate tasks contained in the same program by only one controller such that all appear to be running simultaneously with relation to one another.

It is the concept of a 'state' that allows one to say that a machine, or a section of a machine, is in a particular state, and that allows the structuring of a program so that a state is defined in one contiguous block only within the program with some form of state delimiting statement or means. From this follows the ability to say that while a particular state exists, the only possible relevant part of the program related to that loop is the block defining the state. A class of outputs is provided as part of the language where no outputs of this type exist from a system, other than those defined in the states active in each loop. This type of output is typically used for activating motors, or valves. A state can therefore also be said to exist in the sense that it produces real world outputs while it exists, and these outputs are functions of the state only, i.e. they are activated conditional on nothing other than the existence of the state.

It may be advantageous to allow qualification of those outputs based on other conditions present in the system, but this would be an optional extension of the system.

It should be noted that it may be advantageous to also provide a class of output that can be activated in one state and deactivated in another, used for instance for driving displays or registers.

It is also the existence of explicit state type statements that allow subclasses such as 'break state', and 'pause state' to be provided for the purpose of controlling the flow of the program.

Debugging and comprehension of the control activity can therefore be based on a 'state structure' as follows :

- an identification of the states that exist and what major control function they provide e.g. while a state may activate several outputs, do some timing, interrogate inputs, print, and eventually take the decision to change to another state, it can also be said to carry out some macro function; e.g., the controller for an elevator might have a state that performs the macro function 'go up one floor', which consists of activities like those noted above. The state could therefore be called GoUpOnefloor as a single word symbolic name in our system, and this provides structure to the program and broad understanding without the need to consider the lower level operations necessary to carry out the macro function ;
- examination of the flow of control through the various states that existed previously, that ultimately led to the overall existing state of the machine, (i.e. tracing);
- enabling the flow of control until a particular state is reached, i.e. trapping with a break state, with subsequent diagnosis of the activity to that stage, or pausing a particular loop with a pause state.

It can therefore be seen that it is most advantageous for a loop, program, group program or whatever it is called to be able to be said to be in some state, and for that state to be defined unambiguously, and to be a function that can be said to exist or not, that can be detected and/or recorded.

For tracing to be effective, the concepts of states and loops must both exist.

SUMMARY OF THE INVENTION

Accordingly, in one aspect, the invention may broadly be said to consist in a programmable controller which executes a finite state applications program and facilitates debugging of said program characterised in that said finite state applications program comprises two or more program loops, each of said program loops emulates a state diagram and has two or more blocks of statements, each block when executed by the controller being a program state, and each program state being active only when one or more conditions or operations cause said program state to occur, and said programmed controller includes:

- (a) means for enabling two or more of said program loops to become active for execution and means for enabling only one block of said blocks within any active program loop to become active for execution at any one time;
 (b) means for recording said occurrences of each said program state for later display; and
 (c) means for recording for later display said one or more conditions or operations causing the occurrence of each said program state.

In a second aspect, the invention may broadly be said to consist in a programmable controller which executes a finite state applications program and facilitates debugging of said program characterised in that said program is stored in pseudo code and executed by a pseudo code interpreter program, said finite state applications program comprising two or more program loops, each of said program loops emulates a state diagram and has two or more blocks of statements, each block when executed by the controller being a program state, and each program state being active only when one or more conditions or operations cause said program state to occur, and said programmed controller includes means for enabling two or more of said program loops to become active for execution and means for enabling only one block of said blocks within any active program loop to become active for execution at any one time.

A preferred embodiment of the invention includes a programmable controller for emulating one or more state diagrams and for controlling the states of operation of at least one machine or process. The controller performs an application-type program which comprises a plurality of blocks of statements. Each of these blocks of statements when executed is a program state defining an internal controller state which corresponds to one of the machine or process states of operation. Each block of statements defines the next blocks of statements one of which may replace the existing block as the active block if the defined transition conditions are satisfied and actions to be taken during the corresponding state of operation by the machine or process.

Program states are organised into two or more program loops which represent independent sequential control schemes for an entire machine or process or for a sub-section of the machine or process. Each program loop essentially emulates a state diagram.

The controller can activate two or more program loops simultaneously, however, each program loop can only allow one of the program states within the program loop to become active. To keep track of the active program loops and the associated active program states, a variable is associated with each active program loop. The variable stores an identifier of the active program state or the address of the start of the active program state. The variable also has an identifier which indicates within which active program loop the particular program state resides. The controller also has a task table which stores the variables for the active program loops. The task table enables the controller to perform several active program loops asynchronously and it determines the sequence of execution of the active program loops. Essentially, the task table keeps track of which program loops and which program states are active and of the order and sequence each program state will be processed.

A method and apparatus are also provided for facilitating "debugging" of the finite state applications program. Means are provided for recording information about the activity of each of the active program states. Specifically, the controller traces a history of the order and occurrence of each program state and the conditions responsible for each occurrence of the program state. Display means are also provided for enabling the programmer to evaluate the recorded information.

Significantly, the present invention allows a control, manufacturing or machine design engineer to more freely understand and even design high-level applications programs representative of state diagrams. The specific design advantage of the present invention is that the programmer is able to organize all program statements related to a particular state of operation into a contiguous segment of program code or block of statements and none of the statements are related to any other state of operation. In this way, the programmer is able to compartmentalize all of the various conditions dependent on each state of operation without having to worry about any of the other states of operation. This procedure does away with "state logical variables" and it facilitates techniques for programming in an organized and structured format. Furthermore, program loops representative of overall state diagram structures are easily designed into code by simply connecting the compartmentalized code together with conditional GOTO statements or the like contained within program state blocks.

The present invention is more efficient than the prior art because all of the statements in the applications program do not have to be scanned each time a particular state of operation needs to be controlled. Instead, only those statements in the particular program state associated with the particular state of operation which is active are scanned. This technique enables the controller to process very quickly and it allows the present invention to more efficiently process a multitude of tasks at the same time. The preferred embodiment of the invention also has a task table for keeping track of the various sequences of tasks to be performed simultaneously and/or determining the order that each sequence of task will be performed.

Lastly, because the applications program is designed with program states, pin-pointing the causes of unexpected events is greatly facilitated. Means are provided for recording the sequential order of occurrence of each active program state as they occur and one or more conditions responsible for transitions occurring from each program state to the next or responsible for initial activations or program states. By analysing a display of the occurrences of each of the program

states and by analysing the conditional statements causing the transitions to occur the analyst can quickly determine the location and the cause of the unexpected event in the program.

BRIEF DESCRIPTION OF THE DRAWINGS

Figure 1 shows a programmable controller for controlling the operation of a machine or process in accordance with the present invention and also shows a simple machine or process being controller;

Figure 1a is a state diagram for a controller for apparatus and controls similar to figure 1;

Figure 1b is a ladder diagram for a controller for apparatus and controls similar to figure 1;

Figure 2a is a different state diagram of the various states of operation of an elevator and the conditions which cause transitions to occur between states;

Figure 2b is a diagram of blocks of statements corresponding to the states of operation of the state diagram in Figure 2a;

Figure 2d is a more complicated diagram showing several independent state diagrams;

Figure 3a is a diagram of a program loop pointer;

Figure 3b is a diagram of the task table;

Figure 3c is a diagram of a task table pointer;

Figure 4 is a logical representation of a trace table shown in the form of an endless loop;

Figure 5 is a schematic block diagram of the subroutine called START PROGRAM;

Figure 6 is a schematic block diagram of the subroutine FIRST OPERATOR;

Figure 7 is a schematic block diagram of the subroutine ADVANCE P-CODE POINTER;

Figure 8 is a schematic block diagram of the subroutine ADVANCE TASK TABLE REGISTER;

Figure 9 is a schematic block diagram of the subroutine NEXT OPERATOR;

Figure 10 is a schematic block diagram of the subroutine GO TO;

Figure 11 is a schematic block diagram of the subroutine START;

Figure 12 is a schematic block diagram of the subroutine GO SUB;

Figure 13 is a schematic block diagram of the subroutine RETURN;

Figure 14 is a schematic block diagram of a subroutine called STORE HISTORY;

Figure 15 is a schematic block diagram of a debugging routine called DISPLAY HISTORY;

Figure 16 is a schematic block diagram of a debugging routine called DISPLAY LINE;

Figure 17 is a schematic block diagram of a debugging routine called DISPLAY PROGRAM STATE;

Figure 18 is a schematic block diagram of a debugging routine DISPLAY LOOP;

Figure 19 is a schematic block diagram of a debugging routine FOUND LOOP.

DETAILED DESCRIPTION

Before considering the description and implementation of the language interpreter software we will describe the principles on which the language is based and give examples of how programs according to the invention are constructed.

The Loop-State structure is the single most important feature of the language system which enables simple management of the complex and changing requirements for control in complex systems.

The simple concepts are as follows. Consider an elevator such as may be used to raise/lower a platform on a building site. It has an Up button and a Down button at the top and an the ground. It can be said to be in the states of Stopped, Going Up, or Going down, but only in one of those at any one point in time. Its operation can be described with the help of the state diagram in figure 1a.

The following should be noted

- the states with their numbers and names included in circles;
- the transition conditions defined in words and shown beside the arcs that show which state the condition causes to change to which;
- the way in which a state diagram description minimises and defines what is relevant; e.g. in Going Up, what is happening to the 'Down buttons', 1 or 2, is quite irrelevant.
- Similarly the way all relevant activities are shown associated with the state they relate to E.g. 'Energise motor down' in GoingDown.
- The ease with which the operation of the elevator can be understood. E.g. if it is stopped and up button 1 is pushed, it goes up, but if it is going down and up button 1 is pushed, it ignores it.

The program using our language which is functionally equivalent to the above state diagram is

Stopped

5 Brake ON

IF UpButton1 ON OR UpButton2 ON GOTO GoingUp

10 IF DownButton1 ON OR DownButton2 ON GOTO GoingDown

GoingUp

UpMotor ON

15 If UpButton1 OFF AND UpButton2 OFF GOTO Stopped

GoingDown

DownMotor ON

20 IF DownButton1 OFF AND DownButton2 OFF GOTO Stopped

The only parts omitted from this program are defining statements that allocate particular names to particular variables, i.e. that define UpButton1 as connected to input terminal 21 say.

25 The ladder diagram equivalent is as shown in figure 1b
In the ladder diagram

-] [is a normally open contact
-]/ [is a normally closed contact
- () is a relay coil

Note how although the circuit operation is relatively easy to follow, what it actually does functionally is far harder to deduce. There is no structure to show that is relevant at any one point in time.

A state diagram with 100 states instead of 3 still has states as simple as the 3, and each may be understood in turn.
35 A ladder diagram with 100 rungs is very difficult to understand.

The principles of our loop-state structure are:

- Programs are organised into (two or) more than two separate loops that represent independent sequential control schemes for sections of the machine. A loop can be processed independently of the rest of the program, or can handshake and synchronise with other loops via flags etc.
- Loops are formed of sequences of states. i.e. they are in effect state diagrams representing state machines having outputs which may be conditional/unconditional on the inputs etc.
- Each state machine, i.e. loop, has one and only one state active at any point in time that the loop is active. If the loop is not active then there is no state active in that loop. Active loops can therefore be said to possess a state variable, the value of which is the number/name of the state active at that point in time. This can be accessed and used for handshaking, diagnostics etc. This is also the mechanism that focuses the observer onto a particular part of the program as being relevant for a particular purpose, and EXCLUDES the possibility that any other part is relevant. This exclusion is most important because it says 'look here, and DON'T LOOK ANYWHERE ELSE'.
- A class of outputs is provided as part of the language where no outputs of this type exist from a system, other than those defined in the states active in each loop. This type of output is typically used for activating motors, or valves. A state can therefore also be said to exist in the sense that it produces real world outputs while it exists, and these outputs are functions of the state only, i.e. they are activated conditional on nothing other than the existence of the state. The output setting statement in the language effectively says 'While in this state this output is on and will go off when this state becomes inactive'. It does not say 'set this output on', and does not need any additional statement to turn the output off again.
- Transition conditions are a function of the state they cause to be deactivated. They say 'change the active state from this to another', where the other is given as a parameter of a GOTO statement. They therefore appear in the state block of which they are a part.

- State blocks are contiguous segments of program, starting with a state statement and ending at the next state statement. They contain all the code related to that state, and none related to any other.

We see therefore that

- for one section of a machine, only one loop is relevant;
- for that section of the machine at a particular point in time in the sequencing of the section, only one state in that loop is relevant.

We have therefore focussed down in the program to a segment representing one state, probably e.g. 10 lines out of 2000, and we can be sure that that is the only part that is relevant.

To do this we need only determine the section of the machine associated with the problem or other event of interest

- which is a physical association only, and the point in the sequence that the problem occurs. Trapping and tracing provide the means for determining the point in time in the sequence if this is not obvious from the program. The specifics of trapping and tracing are described later in this specification.

Documentation is made easy by the textual form of the program, as opposed to a graphical form.

Figure 1 depicts a programmable controller and a computer program which form programmed controller 10, for emulating state diagrams. A finite state applications program is stored at 14 in the memory portion 10A of the programmed controller 10. A central processing unit ("CPU") 70 performs mathematical and logical functions for the finite state applications program ("applications program"). A task table or set of pointers 68 keeps track of active program loops. A trace table 82 contains a temporary history of the activity of the applications program. The programmable controller is equipped with a debugging monitor 74 for enabling a user to easily pin point errors in the applications program. The trace table 82, the task table 68, and debug monitor 74 are all stored in memory 10A. Other operating system software for controlling miscellaneous features of the programmable controller are stored at 76. Terminal 80 is provided to enable a user of the controller to program the applications program and debug the system on terminal 80.

Figure 1 shows a programmed controller 10 connected to machine or process such as elevator 12. The programmed controller 10 emulates the applications program at 14, for controlling the operation of the elevator 12. Finite state applications programs are high-level programs representative of state diagrams (See Figure 2a). State diagrams are pictorial representations of the states of operation of the particular machine or process to be controlled. A programmer controls the machine or process represented by the state diagram by first designing a finite state applications program and; second, processing it on the control to emulate the state diagrams and control the order and sequence of occurrence of each task or operation to be performed by the machine or process.

A simplified application of the programmed controller 10 is monitoring, evaluating and controlling the operations of elevator 12. Elevator 12 consists of a reversible motor and brake 16, pulley 20, and an elevator compartment 18. A fundamental principal of the elevator is that only one state of operation of the elevator can be activated at any one point in time. The elevator's states of operation are Stopped, GoingUp, or GoingDown.

Buttons 22 in elevator compartment 18, cause the elevator to change from one state of operation to another state of operation. For example, when the Up button is closed and the Down button is open, the elevator compartment 18 goes up. When the Down button is closed and the Up button is open, the elevator compartment 18 goes down. When the Up button is pressed while GoingDown or when the Down button is pressed while GoingUp the elevator compartment 18 is stopped. Figure 2a is a state diagram or graphical representation of these states of operation and the conditions which effect changes to occur from one state to another. More particularly, a state diagram is a series of nodes 24, 26 and 28 with interconnecting arcs 30, 32, 34 and 36. Each node represents a particular state of operation and each connecting arc represents a condition or transition function which causes the status of a particular state to change.

Specifically, the state diagram in Figure 2a has nodes 24, 26 and 28 which respectively represent the states of operation GoingUp, Stopped, and GoingDown. The conditions which effect the status of the states are the arc connectors 30, 32, 34 and 36 between the nodes. Each arc defines its own set of transition conditions, origin and destination state. Arc 32 has switch condition and action 42:

If the Up button is on and the Down button is off then state of the elevator 18 changes to GoingUp. Arc 30 has switch condition and action 38:

If the Up button is off or the down button is on then the state of the elevator 18 changes to Stopped The rest of the conditions and actions can be discerned from Figure 2a.

An applications program representative of the state diagram in Figure 2a is:

```
GoingUp UpMotor ON IF UpButton OFF OR DownButton ON GOTO Stopped
Stopped Brake ON IF UpButton ON AND DownButton OFF GOTO GoingUp
```

If UpButton OFF AND DownButton ON GOTO GoingDown
 GoingDown DownMotor ON IF DownButton OFF OR UpButton ON GOTO Stopped

The three blocks of statements above are called program states and each block corresponds to both a particular state of operation of the elevator 12 and to a particular node in the state diagram (Figure 2a). Each of the program states consists of a state delimiting statement. GoingUp, Stopped, or GoingDown, which signifies to the controller the end of one program state and the beginning of a new program state. The state delimiting statement is followed by a series of statements consisting of none, one or more compound statements which can cause actions or cause transitions to occur from a currently processing or active program state to another program state. A compound statement consists of a condition part and an action part. If the condition part is satisfied, then the action part is executed. If the action part contains a GOTO, GOSUB or RETURN statement a transition to a new program state occurs. If the action part contains a START statement then a new program loop is initiated with an additional newly active state. Stated differently, these statements determine what the program state changes or initiations will be and the type of actions to be taken by the machine or process during the currently active state of operation. All three program states above have compound statements which include GOTO which cause the program states to shift processing to another program state in the same state diagram.

By gathering all statements which are associated with a particular state of operation into only one contiguous segment of code, the system can scan only these statements to initiate actions required by the state of operation of the machine or process and to determine if a particular set of conditions have been satisfied in order to initiate a new state of operation. Whereas, the Drummond system evaluates every statement in the applications program to determine whether to change state activity or carry out an action, and in the course of this has to evaluate many state variables. The structural difference in the program enables the present invention to operate much more efficiently by eliminating the need to evaluate state variables.

Figures 2b are pictorials of the program states. Block 48 refers to the sequence of code for the program state GoingUp; block 50 represents the sequence of code for the program state Stopped, and block 52 represents the sequence of code for the program state GoingDown. Arcs 54, 56, 58 and 60 (Figures 2b and 2c) correspond to the arcs 30, 32, 34 and 36 of the state diagram (Figure 2a). Arcs 54, 56, 58 and 60 show how the flow of activity in the applications program can be affected by the compound statements.

Arc 58 of Figure 2b demonstrates that when the condition part of compound statement:

"IF Up Button OFF OR Down Button ON GOTO Stopped" is true, then a transition from the GoingUp program state to the Stopped program state occurs. Likewise, arc 60 demonstrates that when the condition part of compound statement: "IF Up Button OFF AND Down Button ON GOTO GoingDown" is true, then a transition from the Stopped program state to the Going Down program state occurs.

The connections between the program states Going Up Stopped and Going Down logically create a program loop 46. The program loop 46 controls the sequence of operations to be performed by the elevator 12. In the preferred embodiment, a program loop can only have one active program state at any one time. In this way, the controller can keep track of the sequential processing of an individual scheme of tasks. The program loop 46 in Figure 2b exhibits the characteristic of having only one program state active at any one time in order to make the algorithms involved in program execution and debugging more efficient and to reduce ambiguity in the system.

In more complex machines, a single program loop will only represent the control of a particular independent subpart of the machine or process. Usually, multiple program loops are required to independently control several subparts of a complex machine or process. Figure 2d, shows a more complex machine represented by a plurality of state diagrams 62, 64 and 66. Each state diagram represents a program loop depicting the sequential control scheme for a section of the machine or process. The present invention emulates these diagrams by processing program loops which control three hypothetical subparts of a machine asynchronously (also called multitasking). A particular program loop can be processed independently of the rest of the program loops or it can "hand shake" and synchronize with the other program loops using some convenient mechanism such as flags or a statement of the type "IF LOOP N AT STATE M" which tests the value of the active state in a different loop. This multitasking capability of the controller is largely transparent to the programmer, and the operating system and language provide a simple means to implement it.

To facilitate the execution of program states within a program loop, the preferred embodiment provides a program loop-record for each of the active program loops. Referring to Figure 3a, the program loop-record 90 has fields 92 and 94 which hold simple variables. The first simple variable 92 is an identifier of the program loop. The second simple variable 94 is an identifier of the active program state or an address in memory of the start of the active program state. Both the address or the identifier of the program state can be easily converted to the other. The remaining portion at 96 of the pointer 90 contains other less important status information about the program loop. The real purpose for the program loop record 90 is to keep track of which program state within a particular program loop is active at any one point in time.

When a particular program loop becomes active, a program loop record 90 is associated with the loop. As various

program states become active within the program-loop, simple variable 94 is overwritten with the new program state address or identifier. The program loop record 90 enables one program loop to handshake by interrogating a second program loop to determine which state is active in the second program loop. Furthermore, the program loop record 90 can be accessed by terminal 80 to allow programmers to display or modify the active program states.

In the preferred embodiment, the program loop records are all stored in a task table Figure 3b which is stored in memory 10A (Figure 1). The task table functions primarily to enable processing of several program loops all at the same time. The program loop records are packed in the task table so that all non-empty records are at the start of the table and they are arranged in the numerical order of their respective program loop identifiers (Figure 3a at 92). Referring to Figure 3b, the first program loop record 100 in the task table has the lowest loop identifier (e.g., 1) and therefore, is first to be scanned during a program scan because scanning occurs in numerical order of the program loops, scanning and processing the active state only in each program loop, scanning and processing each statement in each active state in the order it appears in the state block. Loop record 102 has the second lowest loop identifier in the table and it will be processed second. Loop record 104 has the third lowest loop identifier e.g., 3) associated with it and it will be processed third. Loop record 106 has the next lowest identifier (e.g. 34) and it will be processed fourth.

Also associated with the task table is a task table register 112 (Figure 3c) which points to the program loop record which is currently controlling the execution of an active program loop. To scan and process the program loops in the task table, the controller writes the address of the active program loop record 90 into the task table register 112.

When the controller executes the program loop associated with record 100, only one block of statements in the program state associated with program loop 100 is executed or activated. Then, the controller will process the next program loop (e.g. 2) and only one program state associated with it will be processed. The controller repeats this procedure for every active program loop. When the task table register points to program loop record 108 (Figure 3b) which has a zero associated with the loop identifier, the controller is alerted that it has reached an empty program loop record. When this happens, the operating system of the controller points the task table register back to the top of the task table and executes the next program state of the first program loop in the table. This procedure is an endless process unless a terminal interrupt or the like occurs.

Because only a few lines of program code are processed for each program state, the controller can jump very quickly from one program loop to the next. In this way, the controller appears as if it is processing multiple program loops or tasks all at the same time.

In order to show that a program loop has been activated, the controller will assign a program loop record to the activated program loop and insert the new program loop record (Figure 3a at 90) into the task table. The system evaluates the numerical value of the new program loop identifier (Figure 3a at 92) and fits the program loop record into position in numerical order within the task table corresponding to the numerical value. For example, if the numerical value is greater than 34, the new record will replace record 108 in the task table (Figure 3b). However, if the new program loop identifier is between 3 and 34, then the program loop record 108 will be swapped with the record at 106 and the new program loop record will be inserted at 106. Because the present embodiment of the task table can have a maximum of 31 program loop records in the task table, the contents of the task table must be evaluated to ensure that there is space available for the new record. The controller will alert the programmer when the task table has reached capacity or when an overflow condition in the task table has occurred.

Referring to Figure 4, a logical representation of a trace table 82, which is stored in the memory portion 10A of the controller (Figure 1). The trace table is a linear table in memory which is made to function as a wrap around "endless tape" storage buffer. The trace table 82 works quite similarly to an aircraft flight recorder. An aircraft flight recorder provides a "playback" of what happened to the airplane just prior to when the airplane "crashes". Likewise, the trace table records what happens to the states of a machine or process just prior to an event requiring investigation. Stated differently, the playback can be used to find out exactly what decisions were made by the controller just prior to that time.

A trace table register 84 stores the address of the last recorded location in the trace table 84. Figure 4 shows register 84 pointing to record 74 as the last location to be filled in the trace table 82. When the trace table has been filled at 86, the system will continue to store new data and overwrite old data.

The following is a detailed description of the preferred embodiment for processing finite state applications programs and for facilitating diagnostics of unexpected conditions in the finite state applications program.

Figures 5, 6, 7, 8, 9, 10, 11, 12 and 13 are flow diagrams that represent routines which together with the programmable controller control the processing of an applications program. Particularly, in Figure 5, routine START PROGRAM initializes the various pointers and tables of the programmable controller. In Figure 6, routine FIRST OPERATOR evaluates the first statement in a program state. In Figure 7 ADVANCE P-CODE POINTER controls a standard applications program pointer (p-code pointer). This pointer points to the address in memory of the current statement being executed by the controller. ADVANCE TASK TABLE REGISTER in Figure 8 effects which record the task table register points to. In Figure 9 routine NEXT OPERATOR evaluates all statements other than the first statement of the program state.

GO TO in Figure 10 changes which state is active in the program loop it occurs in. START initiates a change of state, or establishes an active state in the loop specified in the start statement. GOSUB in Figure 12 and RETURN in

Figure 13 provide a subroutine call and return facility which operates at the state level.

Referring to Figure 5, a flow block diagram for START PROGRAM is shown. This routine initializes the program controller. Specifically, at block 122, the controller initialises the task table (Figure 4b), the task table register (loop pointer) (Figure 3C), and the p-code pointer. The task table is initialised with a program loop record that points in memory to the first active program loop. Also, the program loop record is initialised to the first program state in that program loop. The p-code pointer is initialised with the start address of the first active program state i.e. the first statement. The system is now ready to process the first active loop. In block 124, the system calls subroutine FIRST OPERATOR to evaluate the first statement pointed to by the p-code pointer.

The block diagram in Figure 6 depicts routine FIRST OPERATOR. At block 128, the controller analyzes the type of statement pointed to by the p-code pointer. The first statement encountered can only be a state type statement either: a STATE statement, a PAUSESTATE statement, or a BREAKSTATE statement. A STATE type statement is a program state delimiter which symbolizes the beginning of a program state. A PAUSE statement also causes its own program loop to pause but allows processing of other active program loops. A BREAKSTATE statement also causes the entire system to stop running and return to the monitor in order to alert the programmer that the controller has discontinued processing. If a STATE statement is encountered at 130, the controller will advance the p-code pointer to the next statement address in the program state by entering routine ADVANCE P-CODE POINTER at 132. If the statement encountered in the program state is a PAUSESTATE statement at 134, the system will enter ROUTINE ADVANCE TASK TABLE REGISTER at 136 to advance processing to the next active program loop. If the statement encountered is a BREAKSTATE type, routine GO TO MONITOR at 138 is entered.

GO TO MONITOR places a message on the terminal to alert the user of the programmable controller that a BREAK-STATE statement has been reached and that execution of program code has been halted.

Referring to Figure 7, a detailed block diagram of the routine ADVANCE P-CODE POINTER is shown. At block 142 the p-code pointer is incremented to the next statement in the active program state. The controller processes the next statement by entering routine NEXT OPERATOR at 144.

When the PAUSESTATE statement is encountered by FIRST OPERATOR (Figure 6), routine ADVANCE TASK TABLE REGISTER is entered. The purpose of this routine is to ignore the current loop and switch to the processing of the next program loop. Figure 8 shows a detailed block diagram of this routine. At block 148, the task table register is advanced to the address of the next program loop pointer in the task table. Routine FIRST OPERATOR is entered at block 150 to process the active program state of the next program loop in the task table.

When a STATE statement is encountered at 128 (Figure 6) the controller proceeds to routine ADVANCE P-CODE POINTER 140 (Figure 7). As mentioned above, this routine advances the p-code pointer to the next statement in the program state and then enters routine NEXT OPERATOR at 144. The first statement of a program state is always evaluated by FIRST OPERATOR.

Figure 9 is a schematic block diagram of the routine NEXT OPERATOR. The purpose of this routine is to evaluate any type of simple statement in a program state, other than the first statement. At block 154, the controller reads the next statement pointed to by the p-code pointer. If the next statement encountered is either a STATE statement, a PAUSESTATE state statement, or a BREAKSTATE statement, the controller will enter routine ADVANCE TASK TABLE REGISTER at 158 (Figure 9) because the end of the state block has been reached. If the statement encountered is not one of the three above, then the controller will evaluate whether the statement is a GOTO statement at 160. If the statement is a GOTO statement, then the routine GO TO is entered at 162. This routine will be discussed in more detail below.

If the statement is not a GOTO statement, then the controller will advance to block 164 and determine whether the statement is a START statement. If the statement is a START statement, the controller will enter routine START at 166. We will discuss this routine shortly.

If the statement is not a START statement, the controller will advance to block 168 and determine whether the statement is a GOSUB statement. If it is a GOSUB statement, the controller will enter routine GO SUB at 170. Assuming that the statement is not a GOSUB statement, the controller will determine whether the statement is a RETURN statement at 172. If it is a RETURN statement, the controller will enter routine RETURN at 174.

If the statement is not a GOTO, START, GOSUB OR RETURN type, then the controller will advance to block 176 and evaluate the statement as a normal control statement. Normal control statements do not affect the status of the program states, or stated differently, the program state continues to process with out changing which state is active in the current or a different program loop. Whereas, the statement types: GOTO, START, GOSUB AND RETURN, all potentially affect the status of the program state and/or the corresponding active program loop. As discussed earlier, these types of statements affect the status of a particular program state if a set of conditions is satisfied. Once the controller has evaluated the normal control statement at 176 (Figure 9), it will enter routine ADVANCE P-CODE POINTER to evaluate the next statement in the active program state.

Figure 10 is a block diagram of the routine called GO TO. The GO TO routine determines if a particular set of conditions are satisfied, and if so changes the active state in the loop currently being processed to the new value des-

igned in the GOTO statement and then advances processing to the next program loop in the task table. GOTO provides the basic state changing function whereby transition conditions for active states which are true cause a new specified state to become active in place of the currently active state. Once a GOTO statement is actioned in a state, then no further statements are processed in that state.

At block 182 of Figure 10, the controller evaluates the condition flag status resulting from the evaluation of the condition part of the compound GOTO statement. If the set of conditions in the compound GOTO statement are not satisfied, the controller will enter routine ADVANCE P-CODE POINTER at 184. If the condition is satisfied, the controller will call subroutine STORE HISTORY at 186. STORE HISTORY saves a "decision point" address and the program loop identifier into the trace table of the current loop being processed (Figure 4).

The decision point is the address of the simple statement in the program at which the controller found the conditional part of the compound statement containing the GOTO was true. At block 188, the current program loop record in the task table is updated so that simple variable 94 (Figure 3a) points to a new program state address. ADVANCE TASK TABLE REGISTER at block 190, is then called to update the task table register so that it points to the next program loop pointer in the task table.

Figure 11 is a block diagram of the subroutine called START. The purpose of START is to initiate processing of any other program loop at a state specified in the START statement. By comparison, the GOTO subroutine can only advance processing to the next state in the current program loop.

At block 194 of Figure 11 the condition flag status resulting from the evaluation of the condition part of the compound START statement is evaluated. If the condition flag does not indicate a true going edge the controller enters routine ADVANCE P-CODE POINTER at 196. If the condition shows a true going edge, then the controller advances to block 198 and processes subroutine STORE HISTORY. STORE HISTORY saves the address of the decision point and the program loop identifier for the current loop. At block 200 the controller determines whether the specified program loop is presently active and stored in the task table. If the program loop is already active, then at block 202, the controller overwrites the address currently stored in simple variable 94 of the program loop record (Figure 3a) with the address of a new program state to be activated. However, if the program loop is not presently active, then the system will slot a new program loop record into the task table at 204. The new program loop record will be placed in numerical order in the task table via the numerical value of its program loop identifier. At block 206, the address of the new program state will be written into the program loop record. At block 208, the controller will enter routine ADVANCE P-CODE POINTER and so go on to the next statement.

Figure 12 is a block diagram for GO SUB. GO SUB affects the processing of the active program loop. This routine enables a new state to be activated in the current program loop and status information to be saved so that a corresponding RETURN statement can return control to the state containing the GOSUB statement at the next statement following the GOSUB statement. This provides a subroutine call and return function at state level.

Referring to block 212 of Figure 12 the condition flag status resulting from the evaluation of the condition part of the compound GOSUB statement is evaluated. If the condition has not been satisfied, then the controller will enter routine ADVANCE P-CODE POINTER at 214. However, if the condition statement is satisfied, then the controller will call subroutine STORE HISTORY at 216. This subroutine will save both the decision point and the program loop identifier in the trace table. At block 218, the controller will temporarily save in memory the present address stored in the p-code pointer, the address of the currently active program state, and other necessary status information for use by a corresponding RETURN statement. At block 220, the address of the new program state is written into simple variable 94 of the program loop record 90 (Figure 3a), and this address will also be stored in the p-code pointer. The controller will initiate activity in the new program state by entering routine FIRST OPERATOR at 222.

Figure 13, is a detailed schematic block diagram of routine RETURN. The purpose of the RETURN statement is to get the controller to return to the program state which contained the GOSUB statement. In other words, once the specified program state or state diagram called by GO SUB (Figure 12) is completed execution, the controller will return control to the old program state which was active when GOSUB was executed. This requires that the status values stored by GO SUB be restored.

Referring to Figure 13 at block 224, the controller evaluates the condition flag status, resulting from the evaluation of the condition part of the compound RETURN statement. If the condition has not been satisfied, then the controller will enter routine ADVANCE P-CODE POINTER at block 226. However, if the condition is satisfied, then the controller will call STORE HISTORY at block 228. This routine saves the address of the decision point and the loop identifier in which it occurred. At block 230, the controller will restore the values saved at block 218 of GO SUB (see Figure 12). ADVANCE P-CODE POINTER at block 232 is then entered to continue processing in the old program state.

Figures 14, 15, 16, 17 and 18 are block diagrams of programs which facilitate debugging of application programs. Specifically, Figure 14 is a block diagram of STORE HISTORY which is a subroutine for updating the trace table (Figure 4) with information and is called by the routines, GO TO, START, GO SUB and RETURN. The remaining figures are essentially diagrams of subroutine utilities which enable a programmer on his own command to view information stored in the trace table, the task table or the active program states. Specifically, Figure 15 is a block diagram of a subroutine

called DISPLAY HISTORY which displays the information stored in the trace table on terminal 80. Figures 16 and 17, respectively are subroutines DISPLAY LINE and DISPLAY PROGRAM STATE. These routines are responsible for displaying the statements of a particular program state and illustrate e.g. where within the program state was the decision point at which the controller decided to execute the GO TO, GO SUB, START or RETURN statement that caused the change of state. Figures 18 and 19 respectively, are DISPLAY LOOP and FOUND LOOP which display the active program states and associated active program loops which are stored in the task table.

STORE HISTORY is called whenever the status of a program state changes. STORE HISTORY is called, in subroutines GO TO (Figure 10 at 186), GO SUB (Figure 12 at 216), START (Figure 11 at 198), and RETURN (Figure 13 at 228). As discussed above, STORE HISTORY essentially saves into the trace table the address of the decision point in the active program state and the loop identifier of the program loop being processed.

At block 236 in Figure 14, the address stored in the trace table pointer 84 (Figure 3) is evaluated. If the address pointed to by the trace table pointer is at the end of the trace table, the trace table pointer will be automatically set to the beginning of the trace table at block 240. This step makes the trace table appear as if it is an endless loop (see Figure 4). Whether the pointer is at the end of the trace table or not, the address of the decision point will be written into the record pointed to by the trace table pointer. Additionally, at block 244, the controller will also save the program loop identifier of the loop being processed into the record. At block 248, the controller will return to the calling routine.

Subroutine DISPLAY HISTORY enables a programmer to evaluate the history of program states which occurred during the processing of an applications program. The programmer initiates this program when he wants to see a display of what was stored by the trace table. This subroutine displays the contents of the trace table in a two axis table. The horizontal axis presents the program loop identifier and the program state corresponding to the GOTO GOSUB RETURN or START statement that made the entry in the trace table. The vertical axis represents the sequential order of the program states and program loops with the newest states at the top. DISPLAY HISTORY is a subroutine that is initiated directly by user at terminal 80 (Figure 1). A typical printout of the DISPLAY HISTORY might be:

DISPLAY HISTORY

ACTIVE PROGRAM STATES	ACTIVE PROGRAM LOOPS
-----------------------------	----------------------------

			; display the history table
			; program states and loop identifiers
0	2	2	; line 0 shows state 2 exited (loop 2)
1	3	1	; line 1 shows state 3 exited (loop 1)
2	2	1	; line 2 shows state 2 exited
3	3	1	; line 3 shows state 3 exited
4	2	1	; line 4 shows state 2 exited
5	200	2	; line 5 shows state 200 exited (loop 2)
6	1	1	; line 6 shows state 1 exited
7	1	1	; line 7 shows loop 2 started (oldest)
8	END		; no more entries in the trace table

Note that the command DISPLAY HISTORY is shown first on the screen. Second, across the top are the words "active program states" and "active program loops". The two columns beneath these statements refer respectively to the active program states and the active program loops. A short comment is provided in this specification on the right-hand side to explain each line. The program state which caused a state change last is shown in line 0. Program state 2 in program loop 2 was the last to make a change. The first program state to cause a change and the program loop in which it resided is shown on line 7. Line 8 shows the end of the trace table.

If an unexpected condition occurred altering the expected flow of active states in the applications program, a pro-

grammer could easily narrow down which program state the unexpected condition occurred in by viewing the sequence of program states that occurred during processing. For example, assume the program state 3 should not have occurred in loop 1 (line 3 in the table) in the applications program. The unexpected condition must have occurred in program state 2 the previous state in loop 1 which caused the status of program state 2 to change to program state 3. The programmer can more closely analyze program state 2 and the conditions which caused program state 2 to change to program state 3 by calling DISPLAY LINE (Figure 16). This program will be discussed shortly.

Referring to DISPLAY HISTORY in Figure 15 and at block 252, the controller prints the header which appears at the top of the screen as shown above. Next, at block 254, a copy of the trace table and a copy of the trace table pointer are stored into a buffer. By placing the trace table and trace table pointer in a buffer, a snapshot of what was stored in the trace table when the request for DISPLAY HISTORY was made is acquired and remains unchanged for as long as needed while allowing continued updating of the main trace table. The reason for this step is that the trace table is a dynamically changing table and it is constantly being updated. Only the status of the table at the time the request is made is required for purposes of analysis. At block 256, the system will get the address stored in the trace table pointer of the last record stored. At block 258, the controller will evaluate this address to determine whether the buffer is empty. If the buffer is empty, an end statement is printed and the controller returns to its original processing at block 260.

If the buffer is not empty, the trace table is evaluated to determine if it points to the end of the buffer at block 262. If the pointer is at the end of the buffer, then the trace table pointer is set to the beginning of the buffer at block 264. In this way, the controller processes the contents of the trace table as if the trace table were an endless loop. Regardless of whether or not the trace table pointer is at the end of the buffer, the loop identifier and the state address are obtained from the buffer at block 266. The decision point address will be used as a starting point in locating the program state number which has been previously stored in the state delimiting statement of the program state. At block 268, the program state numbers and the program loop identifiers will be printed in their respective columns as shown above. At block 270, the trace table pointer is incremented to the next record in the buffer and the routine will continue processing at block 258. The program will iterate loop 271 until all of the program states, and the corresponding program loops, which were stored in the trace table, are printed.

In our example above, the programmer knew that program state 3 should not have occurred. He also knows that an unexpected condition must have occurred during program state 2 on line 4 of DISPLAY HISTORY. The programmer can analyze more closely what occurred in program state 2 and determine exactly what caused the program state to change to program state 3, by calling the subroutine DISPLAY LINE. DISPLAY LINE generates a listing of the statements of the program state and indicates at which statement within the program state a decision was taken which caused the status of the program state to change.

Referring to Figure 16, a detailed block diagram for the subroutine DISPLAY LINE is shown. This program is initiated directly by the programmer via a call from the terminal 80 (Figure 1). At block 274, the programmer will designate which line of the printout for DISPLAY HISTORY he wants to have expanded. For example because he is interested in what occurred from line 4 to line 3, the programmer will expand line 4 via the DISPLAY LINE. At block 274, the desired line number of DISPLAY HISTORY is set equal to the variable "Line Count". At block 276, the controller will determine if the line number chosen falls within the valid range of numbers. If it does not, then the controller will return to its normal processing at block 278. Assuming that the line number is within a valid range, the last date entry in the trace table which contains data for line 0 will be pointed to by the trace table pointer at block 280. Next, the trace table pointer is incremented by one at block 290. At block 292, the controller will determine whether the trace table pointer points to the end of the trace table buffer. If it does, the trace table pointer will be set to the beginning of the trace table at block 294. Regardless of whether or not the trace table pointer points to the end of the buffer, the controller will evaluate the "line count" to determine if it is zero at block 296. If "line count" is equal to zero, then the desired program state has been uncovered, and DISPLAY PROGRAM STATE will be entered at block 298. If "line count" is not equal to zero, "line count" is decremented by one and the trace table pointer is incremented by one at block 290. Loop 301 iterates until "line count" is equal to zero at block 236, and the designated line is found.

Figure 17 shows a detailed block diagram of the routine DISPLAY PROGRAM STATE. At block 304, the controller will determine if the record pointed to by the trace table pointer is empty. If the record is empty, then the system will print a statement on terminal 80 that the line requested is invalid and return to its normal processing at 306. Assuming that the record is not empty, the controller will extract the address of the decision point from the trace table and determine the starting address for the program state which encompasses the decision point at 308. In block 310, the controller will print the text for one simple statement of the program state. In block 312 the controller will check to see if it has reached the decision point in the program by comparing the address of the statement printed to the decision point address stored in the trace table. If the decision point is reached, then "***DECISION POINT***" will be printed at block 314. The controller will then step to the next statement in the program state at block 316. At block 318, the controller will then determine whether the next statement is a state delimiting statement; STATE statement, PAUSESTATE statement or BREAKSTATE statement. If any one of these three statements are encountered, the controller will return to its normal processing at block 320 because the end of the state block has been reached. If none of the three statements are encountered,

processing will continue via loop 322 until all the statements of the program state are printed.

Figure 18, is a block diagram for DISPLAY LOOP. DISPLAY LOOP is a subroutine for displaying the contents of the task table. Referring to a typical screen display below, loop identifiers are presented in a horizontal axis and the active program state within each of the program loops is designated vertically beneath each of the program loop identifiers. The following is a typical screen display of an output for DISPLAY LOOP.

Program Loop Identifiers	1	2	3	4	5	...	30	31
Program State Identifiers	201	19	1	0	0	...	0	0

At block 324 of Figure 18, the controller is commanded to print the loop identifiers as shown in the first line of the screen display, as shown above. Block 326 sets the task table register to the first program loop pointer in the task table. A variable called LOOPCOUNT is set equal to zero at block 328. LOOPCOUNT is incremented by one at block 330. The controller then determines whether LOOPCOUNT is equal to the maximum number of program loops plus one that can be stored in the task table. If this condition is true, then the controller will return to its original processing at block 334. If LOOPCOUNT is not equal to the maximum number of program loops plus one, then the controller will get the loop identifier from the relevant task table record. At block 338, the loop identifier is checked to determine if it is equal to zero. If the program loop identifier is equal to zero, then the system will return to its normal processing at block 340 whether the loop identifier is equal to the LOOPCOUNT, then subroutine LOOP FOUND is called at block 344. If the program loop identifier is not equal to LOOPCOUNT, then the controller will tab across the screen to the next loop position at 348. LOOPCOUNT is incremented by one at block 350, and the blocks after block 328 will be iterated until no more active program loops are in the task table.

Figure 19 is a detailed block diagram of subroutine FOUND LOOP. At block 354, the start address of the active program state stored in the program pointer will be accessed. The program state number is extracted from a prestored value in the state delimiting statement in the program state. At block 356, the task table pointer is incremented to the next program loop pointer and the controller returns to the routine DISPLAY LOOP at 346 of Figure 18.

The invention has been described in an exemplary and preferred embodiment, but it is not limited thereto. Those skilled in the art will recognize that a number of additional modifications and improvements can be made to the invention without departure from the essential spirit and scope. For example, a number of different software techniques and any number of different software languages would be suitable for implementing the disclosed invention.

However by way of explanation practical applications of the invention may be useful. To put the invention into effect an interpreter now to be described is used to execute an intermediate or "pseudo" code (P code). The interpreter enables an applications program to be written essentially in the form of one or more state diagrams each of which defines operations of a part of a machine or process whereby any individual state defines the output signals and other data transmitted to the machine or process for a finite period of time.

The following interpreter used an RCA 1802 microprocessor and is assembled using a cross assembler from 2500 A.D. Inc.

The interpreter is used to execute the applications program in which there are:-

- 1) a task table i.e. a table indicating the states active in each loop, and containing the loop variables
- 2) a trace table which when displayed enables the state changes in the program run to be traced.
- 3) in the following listings and examples a semicolon indicates that the rest of the line is commentary only.

Details of the interpreter are as follows:-

;REGISTER ASSIGNMENTS FOR CCSINT.

;R0 not used

;R1 interrupt

;R2 D stack ptr

;R3 Program Counter

;R4 Call instruction ptr

;R5 Return instruction ptr

;R6 link for call

;R7 POP instruction ptr

```

;R8      Macr call ptr
;R9      Return address stack ptr
;RF      Points to P code
;RE      Points to the task table
;RD      Points to the data page
;RC      Decision point address
;RB      Points to the trace table
;RA      Can point to the data page
;X       is normally set to RF
;DF      is used as the condition flag
;X,DF    must be protected to preserve normal values
;        if affected by other actions than specified above

RAMSIZE  EQU    2000H
NCTRACEMSK EQU    80H
OPMAPSZ  EQU    32
IPMAPSZ  EQU    32
NUMOFL   EQU    21    ; the number of loops

;The order of the data in the task table for each loop is:
TTLOOP   EQU    0
TTSTATE  EQU    TTLOOP+1
TTONCE   EQU    TTSTATE+2
TTACCUM  EQU    TTONCE+1
TTXTEN   EQU    TTACCUM+1
TTSTATESTR EQU    TTXTEN+1
TTONCESTR EQU    TTSTATESTR+2
TTRTNAD  EQU    TTONCESTR+1
TTLENGTH EQU    TTRTNAD+2

```

```

;macros
CALL      MACRO ADDRESS
          MACLIST ON
          SEP R4
          DW ADDRESS
          MACLIST OFF

          ENDM

          RETN      MACRO
          MACLIST ON
          SEP R5
          MACLIST OFF

          ENDM

          LXI      MACRO REG, ADDRESS
          MACLIST ON
          LDI >ADDRESS
          PHI REG
          LDI <ADDRESS
          PLO REG
          MACLIST OFF

          ENDM

          PAGE.    MACRO NUMFROMEND
          IFMA 1
          DS      <<(100H-<NUMFROMEND-<$)
          ELSE
          DS      <<(100H-<$)
          ENDIF

```

ENDM

;ROUTINES NOT LISTED

```

5 ;CALL          is the RCA standard call routine modified
;               to preserve registers
10 ;RTN          is the RCA standard return routine modified
;               as CALL
;CALL          PRINT hardware dependant.- prints on the terminal
15 ;               the text string pointed to by the following
;               2 bytes
;CALL          TYPE2D hardware dependant.- prints the decimal
20 ;               value of the accumulator D
;CALL          TYPE4I hardware dependant.- prints the decimal
25 ;               value of the 2 bytes pointed to by register
;               RF
;CALL          COMPT hardware dependant.- prints out the
30 ;               instruction pointed to by RD
;               RD returns pointing to the next P-code
35 ;               instruction.
;               RB is corrupted

```

;The following are routines that are called by the macro MACR

```

40 ;MACR          BMIN1 does the 16 bit subtraction RB=RB-RA
;MACR          IFBIS1 returns DF set if register RB=0
;MACR          STEPON is entered with RD pointing to a P-code
45 ;               instruction (A) and returns the following:
;               RD pointing at the next P-code instruction
50 ;               (B).
;               RB pointing to the p-code token of (A)
55

```

```

;           DF set if the P-code instruction (A)
;           requires compiling
5 ;           D containing the instruction (A) length
;           RC points to 2 bytes containing flags that
10 ;           describe the instruction type
;MACR      SDIR   is a store direct routine
;PUSH,POPR push a register to the stack and
15 ;Pop it off

```

```

INCT PROCESSES STATE

```

```

;multi-loop overhead

```

```

INCT:

```

```

25         GLO    RE
          ADI     TTLENGTH-TTONCE+TTSTATE
          PLO     RE

```

```

30 GETSTP:

```

```

          LDA     RE
          BNZ     MORE ;Loop number greater than 0

```

```

35 GETSTR1:

```

```

          ;finished all active loops so check if running
          ;under monitor control

```

```

          LXI     RA,MONFLAG

```

```

          LDN     RA

```

```

45         LBNZ   MONITOR1 ;go to the monitor

```

```

STARTINTER: ;interpreter only running

```

```

          CALL    IOSCAN ;update all IO

```

```

          SDF     OFFLINEX1

```

;PUT FIXED VALUES BACK IN RD

LDI >OPMAP

PHI RD

LXI RC, INPOUT ;check for a terminal interrupt

LDA RC

ST R2

LDN RC ;get input buffer pointers

XOR ;and compare them (=means
;empty)

BZ CSRENT

OFFLINEX:

LDI 3 ;terminal interrupt

OFFLINEX1;

MACR SDIR

DW MONFLAG

LBR MONITOR1

CSRENT:

;set up RB

LXI RB, TRCTAB

;point to first loop

LXI RE, (TT+TTSTATE)

SEX RF

LDA RE

;SET UP P CODE POINTER FROM TASK TABLE

MORE:

PHI RF

PHI RC

```

      LDA    RE
      PLO    RF
5      PLO    RC
      INC    RC          ;set up trace initial decision
                          ;point
10
;THE FIRST OPERATOR MUST BE STATE, PAUSE STATE
;OR BREAK STATE AND THESE REQUIRE SPECIAL TREATMENT.
15      SMI    00          ;set DF for first time
      LXI    RA, ONCEMASK
      LDI    1
20      STR    RA          ;initialize the once mask
      LDA    RF
25      ;ONLY 3 OPERATORS ARE LEGAL HERE.
      PLO    RA
      XRI    <STATE
30      LBZ    STATEII
      GLO    RA
      XRI    <PSST
35      LBZ    INCT          ;PAUSE STATE JUST OMITS
                          ;PROCESSING THIS LOOP.
40      ;PROCESS A BREAKSTATE INSTRUCTION
      TRPST:
      LXI    RA, STATESTORE
45      LDA    RF          ;PUT STATE NUMBER IN
                          ;STATESTORE
      STR    RA
50      INC    RA
55

```

```

LDN    RF
STR    RA
5      DEC    RF
DEC    RF    ;DECR TO BREAK STATE OPERATOR
10     LXI    RA, MONFLAG
LDI    2
STR    RA    ;MONFLAG
15     LBR    MONITOR1    ;GO TO MONITOR

20     ;Exit routines from each function

I3:    INC    RF
25     STATEII:
I2:    INC    RF
I1:    INC    RF
30     NXTOPR:    LDA    RF    ;get next TOKEN from the P
                    ;Code
35     LBR    VBR1
PAGE.
VBR1:    PLO    R3    ;jump to the interpreter
40     ;service routine

;INTERPRETER TABLE

ITEPTTBL:
45     AIFO:    LBR    AIFOI    ;AND IP/FLAG ON
OIFO:    LBR    OIFOI    ;OR IP/FLAG ON
50     IIFO    LBR    IIFOI    ;IF IP/FLAG ON (1)
SOF:    LBR    SOFI    ;SET OP/FLAG
55

```

```

GOTO:      LBR    GOTOI      ;GOTO
STATE:     LBR    STATEI     ;STATE
5
PSST:      LBR    PSSTI      ;PAUSE STATE
BRST:      LBR    BRSTI      ;BREAK STATE
10
REM:       LBR    REMI       ;REMARK
HALT:      LBR    HALTI      ;HALT INSTRUCTION
START:     LBR    STARTI     ;START INSTRUCTION
15
PAGE.
;SKIP TABLE
;(this must be in the same order as the
;interpretor table)
20
SKIP:      DB      0
25
BR        SKIP2      ;AND IP/FLAG ON (3 byte
;instruction)
;(skip over the next 2 bytes)
30
NOP
BR        SKIP2      ;OR IP/FLAG ON
35
NOP
LBR       VBR1      ;IF IP/FLAG ON
LBR       VBR1      ;SET OP/FLAG
40
LBR       VBR1      ;GO TO
LBR       VBR1      ;STATE
45
LBR       VBR1      ;PAUSE STATE
LBR       VBR1      ;BREAK STATE
LBR       VBR1      ;REMARK
50
LBR       VBR1      ;HALT
LBR       VBR1      ;START
55

```

SKIP2: INC RF

SKIP1: INC RF

SKIP0: LDA RF

PLO R3

;--SET OUTPUT, FLAG--

SOFI: BNF I2

LDA RF

PLO RD

LDN RD

OR

STR RD

BR I1

;AND, OR, IF --IF--, --FLAG-- "ON"

AIFOI: BDF IIFOC ;'AND' ENTRY;

BR I2 ;else exit

OIFOI: LBDF SKIP2 ;'OR' ENTRY:

IIFOI: SMI 00 ;'IF' ENTRY: set condition

;flag

IIFOC: GHI RF ;store decision point

PHI RC

GLO RF

PLO RC

LDA RF ;set up input address

PLO RD

LDN RD

AND ;do the test

```

      BNZ      I1          ;exit if input etc = on
                               ;(flag=1)
5
      ADI      00          ;else set flag = 0
      BR       I1          ;then exit
10
HALTI:
      LBNF     NXTOPR      ;ignore the following code if
                               ;the condition is true
15
;THIS IS THE END OF THE STATE, DO THE NEXT LOOP
;----STATE----
20
STATEI:
;----PAUSE STATE----
PSSTI:
25
;----BREAK STATE----
BRSTI:
30
      LBR      INCT
;--GO TO COMMAND
GOTOI:      LBNF      I2
35
      CALL     STOREHIST
      DEC      RE
40
      DEC      RE          ;point to P CODE address
      LDA      RF
      STR      RE          ;change to new state
45
      INC      RE
      LDA      RF
50
      STR      RE
      INC      RE
55

```

```

5          LDI      0
          STR      RE          ;clear ONCE flag
          LBR      INCT

STOREHIST:
10          DEC      RE
          DEC      RE
          DEC      RE
15          LDA      RE          ;Loop number
          INC      RE
          INC      RE
          STR      R2          ;save loop number
          ANI      NOTRACEMSK  ;enabled?
          BNZ      STOREHISTX  ;no
          SEX      RB
          LDN      RB          ;trace table pointer
          BNZ      RSTTPT6     ;reset at the end of table
          LDI      <<TRCTAB-0FFH>> ;wrap around in trace table
35
RSTTPT6:
          PLO      RB
40          ;FIRST SAVE PCODE POINTER
          ;THEN LOOP NUMBER INFO into historical trace table
          GLO      RC          ;decision point
          STXD
          GHI      RC
50          STXD
          LDN      R2
          STXD
55

```

```

      GLO    RB
      STR    R2
5      LDI    <TRCTAB
      PLO    RB
10     LDN    R2
      STR    RB          ;save current trace table
                          ;pointer
15
STOREHISTX:
      SEX    RF
20     RTN
;-----REMARK COMMAND-----
25     REMI:   LDA    RF
      STR    R2
      LDA    RF
30     PLO    RF
      LDN    R2
35     PHI    RF
      LBR    NXTOPR      ;JUMP OVER PRINT DATA BY
                          ;READING ADDRESS OF NEXT OPR
40                          ;STORED IN P CODE

45     ;START COMMAND IS:-
      ;START, LOOP NUMBER, NEW STATE ADDR, NEXTOPR
STARTI:
50     STTS:
      CALL   ONCETEST
55     LBZ    I3

```

```

CALL    STOREHIST
;fix up RE if necessary
5      DEC    RE
      DEC    RE
      DEC    RE
10     LDA    RE          ;get loop number
      INC    RE
15     INC    RE
      ANI    .NOT. NOTRACEMSK
      SD                      ;compare with loop being
                              ;started
      BDF    STARTOK
25     GLO    RE          ;fix up RE
      ADI    <TTLENGTH
      PLO    RE
30     STARTOK:
      CALL    SEARCHTT
35     LDI    >OPMAP
      PHI    RD
40     SMI    0
      SEX    RF
      LBR    NXTOPR      ;FINISH THIS STEP
45     SEARCHTT:          ;search through task table for loop number
                              ;and set up a new loop
                              ;use RA, RB, RD, and RF, sex RF
50     SEX    RF
      LDI    NUMOFL+1
55

```

LPFD:

```

PLO    RB
LDI    >TT
PHI    RA
PHI    RD
LDI    <TT
PLO    RA
LDI    <TTEND
PLO    RD
DEC    RB
GLO    RB
LBZ    NOVAD
LDN    RA
BZ     LEQZ
ANI    .NOT.NO TRACEMSK
SD                                     ;compare this loop number with
                                     ;[RF]
BZ     LEQZ
BNF    SHLP
GLO    RA
ADI    <TTLENGTH
PLO    RA
      LPFD
SHLP:  SEX    RD
GLO    RD
SMI    <((TTLENGTH-1)
PLO    RA

```

		GLO	RB	
		STR	R2	;Store count
5	SLPA:	LDN	R2	
		SMI	1	
10		BZ	LEQZ	
		STR	R2	
15		LDI	TLENGTH	
		PLO	RB	
20	SLPA2:			
		DEC	RA	
		LDN	RA	
25		STXD		
		DEC	RB	
30		GLO	RB	
		BNZ	SLPA2	
		BR	SLPA	
35	LEQZ:	LDA	RF	
		STR	RA	
40		INC	RA	
		LDA	RF	
		STR	RA	
45		INC	RA	
		LDA	RF	
50		STR	RA	
		GLO	RA	
55		ADI	<(TLENGTH-TTENCE)	
		PLO	RA	

```

      LDI    0
      SEX    RA
5       STXD
      STXD
10      STXD
      STXD
      STXD
15      STXD

```

NOVAD:

```

      LDI    0
      PLO    RB
      RTN

```

ONCETEST:

```

      ;this routine called by the routines
      ;that are only to occur ONCE after
      ;the condition is true
      ;EXIT with nonzero if routine is to be run.

```

```

35      ;a ONCE MASK points to the current bit in the loops
      ;ONCE FLAG for the particular routine that called this routine.
      ;this mask is shifted left one bit each time this routine
40      ;is called : therefore for practical reasons in the herein
      ;described form a maximum of only 8 routines that use this
      ;function are allowed in any STATE.

```

;logic:

```

      ;SET ZERO      := 0
50      ;ONCEBIT      :=ONCEMASK .AND. ONCEFLAG
      ;IF (ONCEBIT = 0 AND DF = 1)

```

```

;          THEN
;          ONCEBIT:=1
5          ZERO    :=1
;IF (ONCEBIT = 1 AND DF = 0)
10          THEN
;          ONCEBIT:=0
;ONCEMASK' :=ONCEMASK . SHIFTED LEFT.
15          LXI     RA,ONCEMASK ;point to once mask
          SEX      RA
          LDN      RE          ;get once flag
          AND              ;test
25          BNZ     ONCE1
          BNF      ONCE2
          ;only time to do it
          LDN      RE          ;once = 1
          OR
35          STR     RE          ;replace once byte
          LDI      1
          BR       ONCE3        ;clear zero flag
40          ONCE1:
          BDF      ONCE2
          ;reset onceflag to zero
          LDN      RE
          XOR              ;once = 0
50          STR     RE
          ONCE2:
          LDI      0
55

```

ONCE3:

```

5          SHLC                      ;save zero and DF
          PLO      RD
          LDN      RA
10         SHL                      ;shift mask for next time
          STR      RA
          GLO      RD
15         SHR                      ;restore zero and DF
          SEX      RF
          RTN

```

IOSCAN:

```

25         ;This routine is dependant on the hardware
          ;configuration of the microprocessor and is
          ;therefore left to the reader to implement
          ;the routine transfers the OPMAP contents to the
          ;real world outputs and transfers the real world
30         ;inputs to the IPMAP
          ;The OPMAP and IPMAP are two contiguous blocks of
          ;32 bytes within one 256 byte page structured as
35         ;follows:
          ;Input bit 0 is the least significant bit
40         ;(representing 1 Hex) in the least significant
          ;byte of the IPMAP.
          ;Input bit 1 is the next least significant bit
45         ;(representing 2 Hex) in the least significant
          ;byte of the IPMAP, and so on up to
          ;Input bit 255 is the most significant bit
50         ;(representing 80 Hex) in the 32nd byte of the
          ;IPMAP.
55

```

;Similarly for the output bits

RTN

MONITOR1:

;a system monitor program that can either run the interpreter or
;CALL any of the following routines:

;DISPHIST ;Display a table of the state changes in each loop
;in the order that they occurred.

;DISPLINE ;Display one line from the above table expanded by
;printing the relevant program state and
;indicating the position that the transfer
;equation became true

;DISPLOOP ;Display the current states in each loop

DISPHIST:

;PRINT THE CHANGES THAT HAVE BEEN STORED IN THE TRCTAB

CALL PRNTHD; NOW PRINT HEADINGS

;NEXT THE CHANGES AS STORED IN THE TRACE TABLE.

LXI RD,TRCTAB

LDN RD

PLO RD

LDI 0

PLO RC ;line number counter

DISPHIST1:

CALL BREAK ;check interruption from
;terminal

BNZ DISPHISTR ;exit

GLO RC

```

SMI      (256/3)      ;number of steps in trace
                                ;table
5
BDF      DISPHISTX    ;PRINT LINE NUMBER
GLO      RC
10
CALL     TYPE2D
CALL     PRINT
DW       SPACMSG
15
INC      RD
GLO      RD
20
BNZ      DISPHIST2
DEC      RD            ;reset the pointer after wrap
                                ;around
25
LDI      1
PLO      RD
30
DISPHIST2:
LDN      RD
XRI      0FFH          ;empty ?
35
BZ       DISPHISTX
LDA      RD            ;GET LOOP NUMBER FROM TRACE
                                ;TABLE
40
PSH      ;save
;TYPE STATE NUMBER.
45
DISPHIST4:
GLO      RC
PSH
50
CALL     STPADR
POP
55

```

```

PLO    RC
CALL   TYPE4I      ;type STATE number
;print a dot every 5 columns across the page
;for the same number of columns as the loop
;number
POP     ;get loop number
PSH
ADI     1           ;extra space
PLO     RA          ;loop counter
LDI     6
DISPHIST5:
PLO     RS          ;dot counter
DISPHIST6:
DEC     RA
GLO     RA
BZ      DISPHIST8    ;found the right loop
DEC     RS
GLO     RS
BZ      DISPHIST7
CALL    PRINT
DW      SPACMSG
BR      DISPHIST6
DISPHIST7:
CALL    PRINT
DW      DOTMSG
LDI     5
BR      DISPHIST5

```

DISPHIST8:

POP

5

CALL TYPE2D ;now print the loop number

CALL PRINT

10

DW CRMSG

INC RC

15

BR DISPHIST1

DISPHISTX:

CALL PRINT

20

DW ENDMSG

DISPHISTR: ;EXIT FROM THE PRINT TRACE ROUTINE

25

RTN

PRNTHHEAD:

30

CALL PRINT ;print the string whose
;pointer is following

DW CRMSG ;points to CR , 0

35

CALL PRINT

DW SPACE7MSG ;7 spaces

40

CALL PRINT

DW ONMSG ;1

45

LDI 5 ;LOAD FIRST LOOP NUMBER FOR
;HEADING

PLO RD

PRNTHHEAD1:

50

GLO RD

CALL TYPE2D ;CONVERT TO DECIMAL AND TYPE
;OUT

55

```

CALL    PRINT
DW      SPACE2MSG    ;2 spaces
5      GLO    RD
      ADI    5
10     PLO    RD
      SMI    NUMOFL    ;maximum number of loops
      BNF    PRNTHAD1

```

```

15 PRNTHAD1:

```

```

      CALL    PRINT
      DW      CRMSG
      RTN

```

```

25
;STPADR IS A CALLABLE ROUTINE TO EXTRACT THE NUMBER OF THE STATE
;WHICH APPLIES TO ANY BUFFER POINT VALUE.
30 ;ENTRY WHERE THE SEARCH ADDR IS OBTAINED FROM
;      (RD).

```

```

35 ;EXIT IS WITH RF POINTING TO THE STATE NUMBER

```

```

STPADR:

```

```

40      LDA    RD
      PHI    RA
      LDN    RD
45     PLO    RA
      DEC    RA
      PUSH   RD
50     LXI    RD,PGMSTT

```

```

STPADR3:

```

```

5      MACR    STEPON      ;TEST FOR STATE, BREAKSTATE 01
                                ;PAUSESTATE

      BNF      STPADR4      ;not compiled type

      INC      RC

10     LDN      RC

      ANI      <STATEMASK

15     BZ       STPADR4      ;not a state type

;NEXT SAVE THE ADDR OF ANY STATE FOUND

      GLO      RB

20     PLO      RF

      GHI      RB-

25     PHI      RF

STPADR4:

      MACR      BMINA      ;compare the addresses
                                ;RB=RB-RA

      BNF      STPADR3      ;not past the target address

      INC      RF

35     POPR     RD

      RTN

40     ;PRINT THE REASON WHY A CHANGE OF STATE OCCURRED

DISPLINE:

      CALL      GETPARM1      ;get required line number in
                                ;RA

45     GLO      RA

      SMI      84

50     LBDF     DISPLINER      ;IF MORE THAN LINE 84 ASKED
                                ;FOR PRINT QUERY

55

```

```

                    LXI    RD, TRCTAB
                    LDN     RD
5
                    PLO     RD
                    ;RD IS NOW POINTER TO TRACE TABLE
10
DISPLINE1:
                    INC     RD
                    GLO     RD
15
                    BNZ     DISPLINE2    ;@ END OF TRACE TABLE?
                    DEC     RD
                    LDI     1
                    PLO     RD
20
25
DISPLINE2:
                    GLO     RA
                    BZ      DISPLINE3
                    DEC     RA
                    INC     RD
30
                    INC     RD
35
                    BR      DISPLINE1

;HAVE GOT TO THE LINE REQUESTED IN THE TRACE TABLE
;NEXT CHECK IF IT CONTAINS AN ØFFH EMPTY INDICATOR.
;IF IT DOES PRINT QUERY
40
;IF NOT THEN CAN PRINT STATE CONTENTS.
45
DISPLINE3:
50
                    LDA     RD            ;table entry empty?
                    XRI     ØFFH
                    LBZ     DISPLINER
55

```

CALL STPADR ;TRACED VALUE OF P CODE IS IN
;RA, STATE NUMBER IS POINTED
;TO BY RF

GLO RF

PLO RD

PLO RB

GHI RF

PHT RD

PHI RB

DEC RB

DEC RD

DISPLINE4:

CALL BREAK

LBNZ DISPLINEX

PUSH RA

PUSH RB

CALL COMPRT ;print out the instruction

POPR RB

POPR RA

MACR HMDMA. ;have we passed the decision
;point?

MACR LEFTSO ;test RB=0

BDF DISPLINE5 ;no

CALL PRINT ;yes so print so

DW DECISMSG

DISPLINE5:

```

5      PUSH    RD
      MACR     STEPON
      POPR     RD
      INC      RC          ;are we at the end (
10     LDN      RC
      ANI      <STATEMASK
      BNZ      DISPLINEX   ;yes so finished
15
DISPLINE6:
      LBR      DISPLINE6
20
DISPLINER:          ;PRINT QUERY
      CALL     PRINT
25     DW       NVALIDMSG
      DISPLINEX:    ;EXIT FROM TRACE LINE ROUTINE
      RTN
30
DISPLINX1:
      CALL     PRINT
35     DW       HISTMSG
      RTN
40
DISPLOOP:          ;DISPLAY LOOP ROUTINE.
      ;PRINT EXISTING STATE NUMBERS
45     CALL     PRINT
      DW       SPACE3MSG
      LXI      RD, TT
50     LDI      0
      PLO      RB
      GETINT:      INC     RB
55

```

```

      GLO    RB
      SMI    NUMOFL+1
5      BZ     PRSEX
      LDN    RD          ;get-loop number from task
                          ;table
10     BZ    - PRSEX

NEXE:
15     SEX    RD
      GLO    RB
      XOR          ;compare current loop with
                          ;desired
20     BZ     FDOK
25     CALL   PRINT
      DW     SPACE4MSG
      INC    RB
30     BR     NEXE      ;TRY NEXT ONE
      INC    RD          ;found so print loop number
      LDA    RD
      PHI    RF
      LDA    RD
40     PLO    RF
      INC    RF
45     ;get first 2 digits
      CALL   TYPE4I
      GLO    RD
50     ADI    <(TTLENGTH-3);already incremented 3 times
      PLO    RD          ;move on to next task in table
55

```

```

BR      GETNT      ;go look for next,
PRSEX:  SEX      R2
5        CALL     PRINT
        DW      CRMSG
10       RTN

```

```

;text

```

```

15 DOTMSG      DB      ' ',0
    SPACE9MSG  DB      ' '
    SPACE8MSG  DB      ' '
20    SPACE7MSG DB      ' '
    SPACE6MSG  DB      ' '
    SPACE5MSG  DB      ' '
25    SPACE4MSG DB      ' '
    SPACE3MSG  DB      ' '
    SPACE2MSG  DB      ' '
30    SPACMSG   DB      ' ',0
    NTRACMSG   DB      'NO TRACE' ØDH
35    DECISMSG  DB      '***DECISION POINT***',8DH,ØDH
    ENDMSG     DB      'END'
40    CRMSG     DB      ØDH

```

DATA

```

;The Task Table

```

```

50 TT:          DS      TTLENGTH*NUMOFL-1 ;THE FIRST PART OF THE
                                     ;TASK
    TTEND:       DS      1                ;THE LAST TASK TABLE ENTRY
55 ;THE ORDER OF THE DATA IN THE TASK TABLE FOR EACH LOOP IS

```

```

;TTLOOP      DB      0
;TTSTATE     DB      0,0
;TTONCE      DB      0
;TTACCUM     DB      0
;TTEXTEN     DB      0
;TTSTATESTR  DB      0,0
;TTONCESTR   DB      0
;TTRTNAD     DB      0,0
ONCEMASK     DB      0          ;used for the ONCE operation

```

PAGE.

;THE TRACE TABLE

```

TRCTAB:      ;The first byte is the low byte of the
              ;pointer to the current position in the
              ;table where the storing of the
              ;historical date is taking place

```

;The format of the table is:

;each entry is 3 bytes long

```

;the table is filled from the top (TRCTABF) down towards the
;bottom (TRCTAB+1). When it reaches the bottom, it is wrapped
;around again to the top

```

;the three bytes are stored in the following order:

;the low byte of the decision point

;the high byte of the decision point

;the loop number

;NOTE THAT THE TRACE TABLE IS 255 BYTES LONG PLUS IT'S POINTER

PAGE. 1

TRCTABF:

PAGE.

;THE FOLLOWING AREA IS USED BY THE MONITOR.

```

5  MONFLAG:      DS      1           ;0= running interpreter
                                   ;1= running in monitor or from
10                                   ;a pwr up
                                   ;2= from break state
                                   ;3= from a terminal halt
15  STATESTORE:  DS      2           ;state number from a
                                   ;BREAKSTATE

```

;the input buffer and pointers from the terminal

```

20  INPUT        DB      0
    INPIN        DB      0
25  ECHOOUT       DB      0
    INPBUF       DS      INPBUFSZ
    INPBUFND     DB      0
30  RATOP:        DS      0AFH-<$
    RASTK:        DS      1           ;RETURN ADDRESS STACK - SUBS
                                   ;CAN BE 40 DEEP
35  S2TOP:        DS      1           ; -80 BYTES
                                   PAGE. 1
40  STK2:         DS      1           ;THE NORMAL STACK WITH REG 2
                                   ;AS POINTER
45

```

PAGE.

;THE OUTPUT MAP STARTS ON A PAGE BOUNDARY

;OUTPUT MAP

```

50  OPMAP:        DS      OPMAPSZ-1   ;Output 0 is bit 0 here
55  OPMAPN:       DS      1           ;Output 255 is bit 7 here

```

; INPUT MAP

5 IPMAP: DS IPMAPSZ-1

IPMAPN: DS 1

FGMAP: DS IPMAPSZ-1

10 FGMAPN: DS 1

; THE P CODE AREA, THE INTERPRETER PROGRAM

15 MEMORYSTT:

PROGEND: DS 2 ; stores the end of the
; program

; (size info)

CHKSM: DS 2 ; stores the checksum of the
; program

25 PROCNUM: DS 1 ; PROCESSOR/PROGRAM NUMBER.

COMFLAG: DS 1 ; FLAGS P CODE IS IN THE
; COMPILED OR UNCOMPILED
; STATE

; 0= compiled run

; 1= uncompiled

35 PGMSTT: ; the P CODE starts here

40 BUFTCP:

INTPGM: DS RAMSIZE-\$-1

45 BUFBCT: DS 1

END

50 Using the above interpreter code, an applications program code is executed.
The invention enables the following to be effected:-

1. Being able to use BREAKSTATE, PAUSESTATE or HALT statements in the program, and being able to run, pause or stop the program from the monitor.

55 The control program written in a finite state type language can be run or stopped from the Monitor program. The control program, or part (s) of it, can also be paused, halted or stopped by instructions within the control program itself. The instructions that do these functions in the following examples are:

PAUSESTATE causes that particular loop to pause or halt when control is passed to that particular state

BREAKSTATE causes the whole program to stop running and return to the monitor in either the "online" or the "offline" modes when control is passed to a particular state

HALT is a conditional instruction that causes all instructions after it in its state to be ignored

2. Being able to selectively or otherwise record the history of state changes of the program and being able to display these:

The program is such that in the step of storing of the history:

During the execution of each instruction that could influence the decision to change states, (any part of the transition function), a temporary record is kept of its location in the program. At such time that the transition function becomes true and control is passed to another state, the temporary information and information identifying the particular loop, is transferred to the historical changes table (the Trace Table). An example of this is shown in the interpreter code at MORE, IIFOC and STOREHEST.

3. Displaying the historical changes and the action that caused the change:

This simply requires the contents of the Trace Table or Task Table to be printed out or displayed in a convenient form. The examples below show three such methods.

The first method displays the identifier of the currently active state in all of the active loops by deriving the required information from the task table.

The second method displays the contents of the Trace Table as a two axis table. The horizontal axis represents the loop that the change took place in, while the vertical axis represents the time order of the changes.

The third method displays the change that occurred in any selected line of the table from the second method, and the event or events that caused the change, by listing out the relevant part of the program and indicating the point within the listing where the decision to change states was finalized.

Example 1

It is to be expressly understood, however, that the following listings are for illustration only and are not to be construed as defining the limits of the invention.

STATE 1

; Program start

! SAMPLE PROGRAM TO ILLUSTRATE

! DEBUGGING USING

! HISTORICAL RECORDING

START LOOP 2 WITH 200 ; start the second loop at

; STATE 200

GOTO 2

; unconditional state change

STATE 2

; Program structure

SET OP 2

; Control outputs (OP means

; output)

IF IP 3 ON

; Transition equation (IP means

; input)

OR IP 7 ON

GOTO 3

; Destination

```

STATE 3                                ; Next finite state
SET OP 3
5 IF IP . 5 ON                        ; 2nd Transition equation
GOTO 2
10 IF IP 4 ON                         ; 3rd Transition equation
GOTO 2                                ; 3rd Destination
STATE 200 GOTO 201                    ; Loop 2
15 ; Unconditional transition

```

```
STATE 201
```

```

IF IP 5 ON
GOTO 202

```

```
BREAKSTATE 202 ; Break point in the program
```

```
END
```

```
; DISPLAY OF THE HISTORICAL TRACE FOR THE ABOVE PROGRAM
```

```
; (>) is the prompt from the monitor program)
```

```
> GO ; start running the program
```

```
Psc Monitor - BREAK AT STATE 202
```

```
; program has been stopped at
```

```
; breakpoint
```

```
> OI LOOP ; monitor command requesting
```

```
the display of the current
```

```
; state in each loop
```

```
2 202 ; this is the response showing
```

```
; loop 1 currently at state 2
```

```
; and loop 2 at state 202
```

```

>DI HI                                     ;display the history table
LI STATE      1 5 10 15 20                ;Line, State and Loop numbers
                                                (newest date at the
5                                                ;top) of table

0    201      2                            ;line 0 shows state 201 exited
                                           ;loop 2
10
1    3         1                            ;line 1 shows state 3 exited
2    2         1                            ;line 2 shows state 2 exited
15
3    3         1                            ;line 3 shows state 3 exited
4    2         1                            ;line 4 shows state 2 exited
5    200      2                            ;line 5 shows state 200 exited
                                           ;loop 2
20
6    1         1                            ;line 6 shows state 1 exited
25
7    1         1                            ;line 7 shows loop 2 started
                                           ; oldest data at the bottom

3    END                                    ; NO more entries in the table
DL LI 7                                     ;display line 7 expanded
STATE                                       ;unconditional decision

**DECISION POINT**
X! SAMPLE PROGRAM TO ILLUSTRATE
35 X! DEBUGGING USING
X! HISTORICAL RECORDING
40 START LOOP 2 WITH 200
GOTO 200

>DI LI 6                                   ; display line 6 expanded
45

STATE 1
50 ** DECISION POINT **                   ; unconditional decision
X! SAMPLE PROGRAM TO ILLUSTRATE
X! DEBUGGING USING
55

```

X! HISTORICAL RECORDING
 START LOOP 2 WITH 200
 GOTO 2

>DI LI 5 ; display line 5 expanded
 STATE 200 ; LOOP2
 ** DECISION POINT ** ; unconditional decision
 GOTO 201 ; unconditional transition
 >DI LI 4 ; display line 4 expanded

STATE 2
 SET OP2
 IF IP 3 ON
 OR IP 7 ON

** DECISION POINT **

; input 7 on caused
 ; program

GOTO 3

; to change to state 3

>DI LI 3

; display line 3 expanded

STATE 3
 SET OP3
 IF IP 5 ON

** DECISION POINT **

; input 5 coming on caused
 ; program

GOTO 2

; to change to state 2

IF IP 4 ON

; This Transition equation
 ; ignored

GOTO 2

>DI LI 2

; display line

; expanded

STATE 2

SET OP2

IF IP 3 ON

** DECISION POINT **

; input 3 coming on caused the
 ; program

```

      ORX IP 7 ON          ; to change to state 3
5      GOTO 3
      >DI LI 1             ; display line 1 expanded
      STATE 3
10     SET OP 3
      IF IP 5 ON          ; This Transition equation
                          ; false
15     GOTO 2
      IF IP 4 ON
      ** DECISION POINT **      ; input 4 caused the
                          ; program
      GOTO 2                  ; to change to state 2
25     >DI LI 0             ; display line 0 expanded
      STATE 201
      IF IP 5 ON
      ** DECISION POINT **      ; input 5 on caused the
                          ; program to change to state 202
      GO TO 202

```

CODE GENERATED FROM THE ABOVE PROGRAM THAT IS INTERPRETED BY THE
MULTITASKING CONTROL PROGRAM INTERPRETER UNDER SUPERVISION OF
THE MONITOR

The left-hand column shows the memory address where the data on
the right is stored in the controller's memory. We shall refer
to the following code as P-CODE (Pseudo Code).

(For added clarity, the program is repeated as comments near the
corresponding data).

```

X000      10 00 01          ; STATE 1
50 X003      19 X0 1F 53414D504C ; SAMPLE PROGRAM TO
                          ; ILLUSTRATE

```

4520505241

EP 0 249 384 B1

```

                                4020544F20
                                494C4C5553
5                                5452415445
X01F      19 X0 31 4445425547 ; ! DEBUGGING USING
                                47494E4720
10                                5553494247
X031      19 X0 48 484953544F ; ! HISTORICAL RECORDING
                                524943414C
15                                205245434F
                                5244494E47

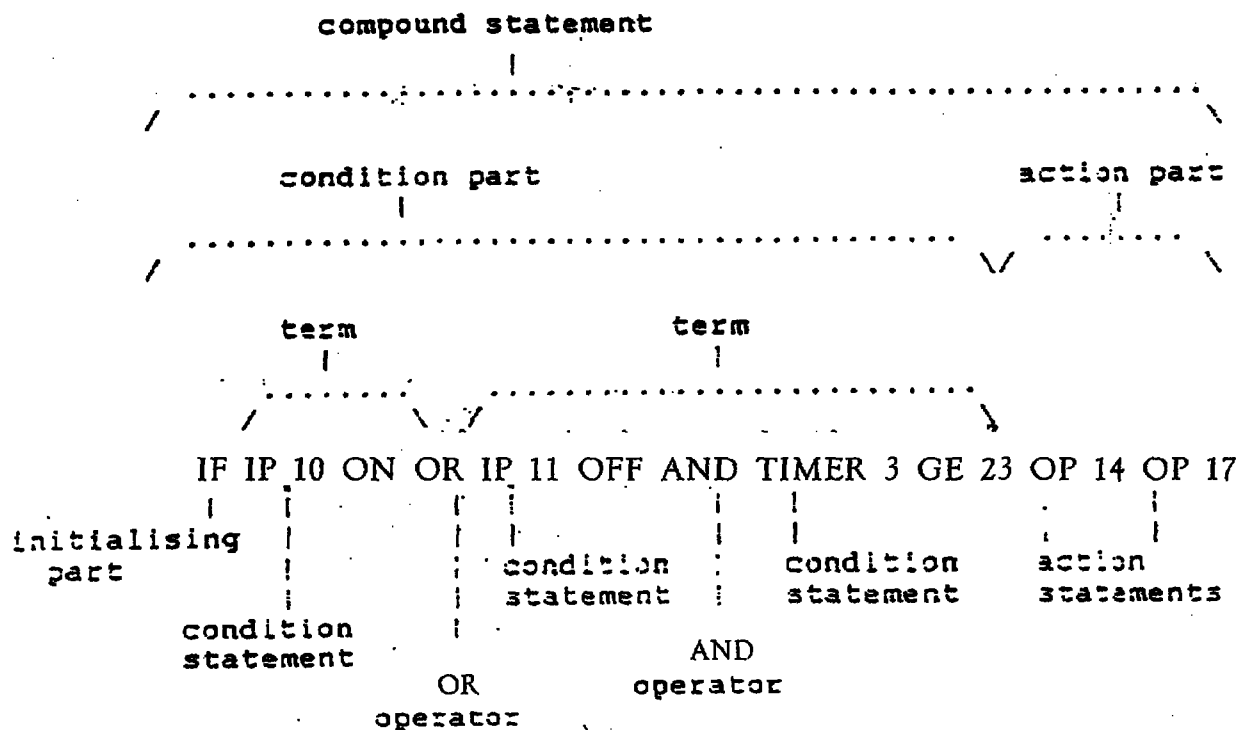
X048      1F 02 X0 70          ; START LOOP 2 WITH 200
X04C      00 X0 4F            ; GOTO 2
X04F      10 00 02            ; STATE 2
25 X052      01 00 04          ; SET OP 2
X055      07 20 08            ; IF IP 3 ON
X058      04 00 30            ; OR IP 7 ON
30 X05B      00 X0 5E          ; GOTO 3
X05E      10 00 03            ; STATE 3
X061      01 00 08            ; SET OP 3
35 X064      07 20 20          ; IF IP 5 ON
X067      00 X0 4F            ; GOTO 2
40 X06A      07 20 10          ; IF IP 4 ON
X06D      00 X0 4F            ; GOTO 2
X070      10 00 C3            ; STATE 200
45 X073      00 X0 76          ; GOTO 201
X076      10 00 C9            ; STATE 201
X079      07 20 40            ; IF IP 6 ON
50 X07C      00 X0 7F          ; GOTO 202
X07F      16 00 CA            ; BREAK STATE 202
55 X082      15 00 00          ; PROGRAM END
                                ; BREAK STATE 0

```

The following section shows how we structure a program written in our language.

The program functional structure can be defined as follows -

- a program consists of one or more program (logical) loops
- 5 - a loop consists of one or more states, only one of which can be active at one time. The set of states making up a loop is implied by the START, RESTART etc. loop initialising statements in the program and the GOTO, GOSUB, state changing statements in those states
- a state consists of a state delimiting statement, followed by a statement list
- 10 - a statement list consists of zero, one or more compound statements
- a compound statement consists of an optional condition part and an action part. The absence of a condition part implies a true condition and the action part would then be unconditionally executed.
- 15 - an condition part is a Boolean expression consisting of an initialising part, one term (or more terms connected by OR operators), and terminates at the next action part
- an initialising part provides the functional equivalent to the connection to the power bus in a ladder diagram, and in our language is an IF statement.
- 20 - a term consists of one condition statement (or more condition statements connected by AND operators)
- an action part consists of one or more action statements The following diagram shows the structure of a compound statement. The meaning of the statement in the diagram is 'If input 10 is on or input 11 is off and timer 3 has timed for greater or equal to 23 time units, output 14 is on, and op 17 is on'.
- 25



The method of evaluation of the condition part and execution of the action part is as follows:

- 55 - the statement is evaluated in the same order as it is read;
- IF sets a boolean condition flag to 1 (true) and this flag is used as a running record of the evaluation as it proceeds, and as a record of the result when it has been fully processed;
- a condition statement is evaluated and if 0 (false) the flag is set to 0;

- an AND operator just proceeds to the next evaluation of the next condition statement;
- an OR operator looks at the condition flag;
 - if the flag is 1, the whole condition part must be true and so the remainder of the evaluation is skipped
 - if the flag is 0, the proceeding terms must all be untrue, so the flag is set to 1 and the evaluation proceeds with the next term;
- an action statement executes and carries out it's function if the condition flag is true when the action statement is read.

The method does not allow the use of brackets in the condition part, and a condition part is equivalent to multiple rungs on a ladder diagram, each consisting of any number or series connected input variables, all connected in parallel at the point where the rungs connect to the actuator, but with no parallel connections at any other point. This allows any necessary condition to be evaluated, although the parallel connections would have allowed this with less evaluations of input variables. However the method provides a very simply implemented way of recording which inputs resulted in a true evaluation and thus caused the decision to execute the action list.

This is achieved by maintaining a pointer to the Pseudo P code representing the program that is being processed, and this is necessary anyway. Any OR operator which detects a 1 condition flag saves the value of that pointer before modifying it to skip to the action part. As we only have complete 'rungs' with no alternative or ambiguous paths in the condition part, the saved pointer value must point to the end of the first rung found to evaluate true. This is termed the 'decision point', and is used in debugging. Should the evaluation proceed to the end of the last term, and thus past the last OR operator, this position becomes the decision point provided the last term yields a true condition, or the position of the action statement can serve equally well in this case.

Broadly a compound statement is equivalent to a rung on a ladder diagram which evaluates a condition, and if the condition is found true, carries out some action. The detail of this part of the structure as defined above is of significance mainly from the point of view of easy recording and print out of the decision point in a trace of any one particular state change. The Program-loop-state structure is the thing of major significance because it is this structure that yields the greatest benefits in terms of easy designing of error free programs, and of understanding those programs.

It is also true from another viewpoint that a program consists of one active state diagram (logical loop), which automatically starts with state 1 active, together with one or more potentially active state diagrams which can be activated from the first, and subsequently any state diagram can activate or deactivate any other. Code describing all potentially active state diagrams which can be activated from the first and subsequently any st diagram can activate or deactivate any other. Code exists within the body of the program at all times, but a state diagram is not executed and is therefore functionally inactive until the state diagram is activated by the assignment of one of its states to a loop variable which subsequently keeps track of which state is active in that state diagram at any one point in time. The systems software has code and variables which enable it to determine whether an assignment has been made, or whether the loop has been deactivated.

An example of a program using numeric (rather than symbolic) variable names is shown below. The program does not do anything, and is for the purpose of illustrating structure only.

Listing 1.

Program	Commentary
STATE 1	;first state delimiting statement.
OP 20 ON	;the only compulsory state statement.
	;automatic start of loop 1.
	;start of first statement list.
	;output 20 is on while in state 1.
	;unconditional action statement.
	;can also be considered a compound
	;statement with no condition part
	;and 3 action statements in the
	;action part.
START TIMER 3	;unconditional action statement.
START LOOP 2 WITH 200	;starts a timer to count up from 0.
	;defines state 200 as start (or restart)
	;state of loop 2.
IF IP 2 ON	;unconditional action statement.
	;start of first compound statement.
	;initialisation part ('if'), and first
	;and only condition statement in first
	;term.
OR IP 3 ON	;ip means input.
AND IP 5 OFF	;start of second term.
AND TIMER 3 GE 34	
GOTO 2	;last statement in second term
	;and in condition part.
	;action statement conditional on
	;result of evaluation of
	;previous condition part.
	;if actioned, causes loop one's active
	;state to change from 1 to 2.
	;start of next evaluation.
if ip 6 on	;if true change active state to
goto 20	;state 20 else leave as state 1
	;end of first state and first
	;statement list.
STATE 2	;second state delimiting statement.
	;start of second state in program.
	;Processing order controlled by
	;scanning order and by
	;GOTOs, GOSUBs etc not order in listing

```

STATE 3                                ;end of state 2.
                                        ;start of state 3.
.
.
5
.
STATE 4
.
.
.
10
.

STATE 20
PRINT "Hold state entered at "
GOSUB 30                                ;to print the time.
.
15
.

                                        ;more states

STATE 30                                ;start of a single state subroutine
READ CLOCK                             ;(but could be more than one state).
                                        ;read the hours and mins etc shown on
25                                ;the clock into the variables called
                                        ;hour and min.

GET hour
PRINT "@ Hrs. "                        ;'@' is a special symbol for the imple
                                        ;variable discussed elsewhere. I.e.
                                        ;what the hour was 'got' into.

30
GET min
PRINT "@ mins."                        ;\ is a special symbol for 'new line
RETURN                                ;unconditional return to calling stat

STATE 200                              ;the state at which loop 2 is
35                                ;started.

STATE 201
.
.
40
                                        ;more states

.
.
(last statement)                      ;end of program.

```

45 In the program above, there is an interesting example of the time sequencing capabilities of the system stemming from the way all statements in a state are processed sequentially in one contiguous block. Part of a message is being printed in state 20, and the remainder is being printed in state 30. This is a suitable technique where there is no random printing by other loops than this one. However, if other loops have states that could possibly print between "Hold state entered at" and "@ Hrs.", this technique would cause a problem. The way round the problem is simple, just place all

50 associated print statements together in a single state and the operating system interpreter will guarantee to process all together, avoiding hold-ups in the control functions by placing characters in a print buffer to be down loaded to the printer one at a time by an interrupt routine as the printer becomes ready.

It should be noted at this point that the functional structure is different to the structure as apparent from a listing of a program. A listing does not directly show the presence of loops, and a program listing consists of one or more state listings only, including the finer detail of compound statements ... action statements. The loop information is of course

55 there, and can be found in the START statements, and the GOTO, GOSUB statements etc.

The third statement in the example program, START LOOP 2 WITH 200, defines the state to become active in loop 2 on the next scan of that loop after the execution of the START statement. Other START statements would normally

be used to define initial states for all other loops used in the program.

It should be noted here that there are two different meanings of the word loop in our system, and the word loop could be qualified and shown as 'logical loop' and 'physical loop'. Broadly the difference is the same as for instance 'logical devices', and 'physical devices' in MSDOS (TM of Microsoft), i.e. a program can talk of printing on a 'theoretical' line printer without reference to where that line printer is, but at some stage the overall software must make the link between the theoretical line printer and the fact that such a device actually exists at some particular address or location in the hardware of the system running the program. Similarly, our program is designed in terms of (logical) loops which are effectively state diagrams and are termed loops because in most practical systems the state that is first active in the diagram will become active again each time the machine or process cycles. The process activity therefore typically cycles through a series of states arriving back where it started and therefore traverses a 'loop'.

In order to enable the processor to run the program in the way desired, we must assign a logical loop to a physical loop. This consists of using a start statement, START, LOOP N WITH STATE M which associates state m with loop variable n, either directly storing M in loop variable N, or storing a pointer or other indirect reference. After executing one or more START statements we have one or more physical loop variables that are initialised with state references and the system software can read these in turn process the states, changing the references in response to GOTO, GOSUB, statements etc, and performing the necessary activities to control the machine/process. As noted elsewhere, which state in the program forms part of which logical loop(s) is determined by the START, GOTO, GOSUB, statements etc, and not by any means such as associating state blocks with larger subprogram blocks.

It would of course be possible to define 'Loop Blocks' which initial states, and use a statement such as 'start loop block 3'. It is considered that the current method provides good flexibility and ease of use.

We should also note that it is often desirable to have different logical loops associated with the same physical loop (one at a time) when performing different alternative control tasks.

Normally the exact meaning of the word loop will be apparent from the context and so the word is not normally qualified with 'logical' or 'physical'.

The details of which states form parts of which loops, and the interconnections between the various states in a loop can be readily determined by following the network through via the state changing instructions in the states, i.e. the GOTOs or GOSUB/RETURNS.

In our example we can see that loop 1 starting at state 1 also has states 2 and 20 in it because state 1 contains GOTO 2, and GOTO 20 statements. We can follow through states 2 and 20 via their GOTO statements etc and find the complete network. Normally the programmer would define the structure of the network before the details of the states, and so this would be predefined. A utility program would normally be used to list any loop network structure should this be necessary, rather than manual inspection.

It can be seen from the above that the functional structure of the loops is built into the program but does not have any delimiting statements to define a contiguous block of code associated with any one loop. It would however be considered good programming practice to code loops using contiguous blocks of states, i.e. perhaps states from the numerical range 200-299 for loop 2, 300-399 for loop 3, 400-499 for loop 4 etc.

It should also be noted that assigning logical loops to physical loops provides a mechanism whereby a physical loop may control different processes at different points in time, in the sense that a process is represented by a network of states. This might occur where a section of a machine can make one of a number of different parts depending on requirements. When a particular part is required a particular loop is started at the start state for the state network which controls the making of that part. If a different part is required, the same physical loop is started at the start state for the different state network that controls the making of that different part. In this way a program can be structured so that different subclasses of activities are controlled by different state networks, thus modularising the program and minimising the extent of the program code that is relevant under any particular circumstances, and simplifying the comprehension of the operation of the machine.

This illustrates the principle that of all the combinations of input variables and states that exist in a system, only a very few simplified combinations are significant to the real operation of the machine, and at any one point in time, only a few of those few are significant.

It is also relevant to note that normal programming in this language, particularly of sequencing operations, often leads to all action statements (other than state changing statements, such as GOTO, GOSUB, etc), being unconditional, and the decisions made in the system thus are mainly related to changes of state. The lack of conditions qualifying actions such as turning on outputs etc. leads to a high degree of certainty as to what has or will happen when the system is in a particular state, and the system is therefore predictable. The user does not have to wonder what happened as he would if outputs were conditioned by inputs whose values were not known and are transient and not measurable later.

It should also be noted that any one state can be said to represent a ladder diagram because the context of the state is a list of compound statements, each of which consists of a condition part and an action part. From this it follows that state changes actually represent the deactivation of a ladder diagram equivalent to the state being deactivated, and the activation of the ladder diagram equivalent to the state being activated. However it should also be noted that this is

done in a high level way using abstract concepts such as states and conditional GOTO statements, and does not involve the manipulation of any extraneous elements such as enabling or master relay contacts or connections from such contacts to the various rungs of the dependent ladders, and in addition the activation of one state by a GOTO statement precludes, at the system level, the possibility of the state containing the GOTO statement still being active, because this action involves the overwriting of the value held in the loop variable for that loop.

It also follows from this that a program, only containing STATE 1 (and thus not having any state changing action statements) and where STATE 1 contains a list of compound statements, is in fact the functional equivalent of a ladder diagram program as used with a normal ladder diagram type programmable controller. Although this method of programming would not normally be used when the more powerful state-loop structure is available, it is relevant to note the equivalence because it indicates the relationship of ladder diagrams to state machines of the type that have their outputs conditioned by system inputs.

It follows that ladder diagram functionality is a sub-function of a state diagram, which is itself a subfunction of a program consisting of multiple state diagrams, and in our case where these state diagrams may be active or inactive depending on whether they have been started by a 'START LOOP .. WITH ..' type of statement. (In our system, they may also be overridden, paused, stopped, etc). It is also worth noting that the loops could also be started by means other than statements in the program itself, by preallocation of loop starting state numbers, or a table driven system, from the system monitor, etc.

It also follows that physical loops are not directly equivalent to particular state diagrams. They are system variables, members of a list, that hold the value of an active state, and by virtue of the structure of the various state diagrams emulated by the program, define which state diagram is being controlled via that loop. Which particular state diagram is being controlled via a particular loop depends on the START statement that started it, (or the subsequent GOSUB statements that swap in different 'sub state diagrams' until a 'RETURN' statement is encountered, etc). Loop variables may be statically allocated in memory or dynamically allocated.

In the above sense 'loop 2' refers to physical loop 2 or the value of the active state stored in loop variable 2. GET LOOP 2 in our language means get the number of the active state in the state diagram currently being controlled by physical loop 2 using loop variable 2. The word loop has also been loosely used to mean state diagrams, i.e. logical loops, and we have just seen that any one physical loop can be used to control different state diagram emulations at different points in time. The precise meaning should be determined from the context. The reason the same term has been used for two different but related concepts is that this assists in the intuitive understanding of program functionality.

As we have seen, each active state is processed each scan. Statements such as 'START TIMER' do not need processing each scan. Depending on the approach to programming, it is necessary to process such statements

- either, in the first scan after activation of the state,
- or, in the first scan after each time that the condition part of the compound statement involved becomes true, including the first scan through the state that it is true.

Such processing is similar to the equivalent ladder processing together with the additional state qualification, and avoids timers being started repetitively each scan, thus being held at 0, and continual repetition of messages being printed each scan, etc. Most action statements require this type of processing, except for statements such as 'OP 22 ON', which means 'output 22 is on in this state'. This needs repetitive insertion into a pre-zeroed output array each scan so that the output will automatically deactivate when the state is deactivated.

Symbolic Variables.

A program using symbolic variables but otherwise identical to Listing 1 is shown below in Listing 2.

Note that state 1 now has a list of define statements for loops, states etc. These may be manually inserted or automatically inserted by the operating system when a new variable name of a particular type is encountered during program entry, automatically allocating the next free number for the particular type of variable. These define statements could be in state 1 or not (they could be in a separate block), hidden or explicit (as shown).

The main point of note is that the program now has a great deal of user friendliness, and to a large extent is self explanatory and self commenting providing only that the reader has an understanding of the machine or process itself.

The indenting of the text should also be noted as this conveys information indicating the positions of IF, OR, AND operators, compound statements, condition parts, action statements etc, and breaks the program up into groups of statements that are functionally related, again easing the task of comprehension.

Listing 2.

	Program	Commentary
5	Initialise:	
	DEFINE LOOP	2 = Cutter_operation
	DEFINE STATE	2 = Infeed_active
		3 = Infeed_counting
		4 = Infeed_slowdown
10		.
		.
		20 = Infeed_hold
		.
		.
15		200 = Cutter_wait
		201 = Cutter_down
	DEFINE IP	2 = Infeed_start_button
		3 = Auto_switch
		5 = Pause_switch
20		6 = Hold_button
	DEFINE OP	20 = Initialise_indicator
		Initialise_indicator ON
		START TIMER 3
		START Cutter_operation WITH Cutter_wait
25		IF Infeed_start_button ON
		OR Auto_switch ON
		AND Pause_switch OFF
		AND TIMER 3 GE 34
		GOTO Infeed_active
30		IF Hold_button ON
		GOTO Hold_infeed
	Infeed_active:	
35		.
		.
	Infeed_counting:	
		.
40		.
	Infeed_slowdown:	
		.
		.
45	Hold infeed:	
		PRINT: "Hold state entered at "
		GOSUB Print_Time
		.
		.
50	Print_time:	
		READ CLOCK
55		

```

      GET HOUR
      PRINT "@ Hrs. "
      GET min
      PRINT "@ mins.\n"
      RETURN

```

```

      Cutter_wait:

```

```

      Cutter_down:

```

```

      (last statement)

```

Program Scanning.

Our program is processed by the processor in a similar way to that of a standard programmable controller in the sense that it scans repetitively through the relevant statements, evaluating and actioning. In detail it differs. The scan only involves the statements contained in the active states. Which states these are is determined by reference to the loop variables. The processor reads the first loop variable in the list, determines from this where the beginning of the active state for that loop is in the program. It goes to that state and processes the statements in the state until it reaches the next state delimiting statement. This is the beginning of the next state in the program listing and marks the end of the current state being processed. The processor then goes to the next loop variable in turn and repeats the procedure, and ditto through the complete list of loop variables having active states. This activity together with various overhead activities common to programmable controllers makes up one scan. The scan repeats continually.

GOTO, RESTART, RETURN and GOSUB statements change the value of the loop variables immediately when processed and processing of the statements in that state ceases when this occurs and the processor transfers to process the next state in the list. I.e. statements past the GOTO statement in that state are not processed in that scan. On the next scan it of course processes the statements in the new state that has been swapped in, and the old state which was overwritten is not active or processed. Order of processing statements in any state is in the same order as the program listing, and loops are scanned from 1 upwards in numerical order.

Multitasking operating systems (MTOS) used as the basis for real time software are well known. They function in a similar way to our system in that they share processor time around the different tasks requiring attention, and we do the same with the different active states in the different loops. MTOS however do not necessarily impose the rigid order of processing on their tasks that is imposed by our order of scanning loops, processing each active state completely before going on to the next. MTOS normally can swap the processor from one task to another on the basis of task time-out, ready task priority, interrupt, etc. This means that the actual order of processing of tasks and even statements is not necessarily known to the programmer who writes the applications program although there are mechanisms provided to allow the programmer to control sensitive parts of the program or hardware system as deemed necessary.

In a machine control or process control environment, it can be very advantageous to know the exact order of processing, or to be able to predict or ensure that the response time of the processor to system inputs will be adequate. Machines particularly, require fast response, and with programmable controllers it is normal to specify the time for one full scan of the program, and this may be of the order of 1 to 20 milli-seconds. Our system provides a means of ensuring that all necessary activities are always carried out sufficiently quickly.

The structure of our language thus establishes that one scan will take the time needed to process all the active states once, (but not any of the inactive states), plus some additional system overhead that is predictable. This provides for predictable scan times and because inactive states are not processed, scan times are short, or looked at conversely, a particular scan time can be achieved using our system and a lower power and cost processor, than can be achieved with a system such as ladder logic in which all statements are scanned every scan.

It should be noted that there are ways for standard programmable controllers of avoiding scanning all ladder diagram statements every scan, but these are not as convenient as those provided with our system.

The swapping in and out of active states together with the loop structure provides the programmer with facilities similar to an MTOS, but which are totally predictable. The programmer can control response time by ensuring that states

are coded so that no state takes too long for processing, or by using the loop variables to examine the other loops to ensure no long-winded processing can occur in one loop while a sensitive state is active. The system provides the necessary facilities in a very simple form.

In addition, as the statement list in each active state is processed in the order of the statements in the list, and all statements are processed before going on to the next loop, the programmer has a very simple way of guaranteeing that all necessary operations have been carried out once a list of sensitive operations has been started, and that no operations associated with some other state will intervene. Examples of when this might be important could be when using global variables or writing blocks of text to a terminal or when handshaking operations are involved. While this can be achieved using other techniques such as an MTOS, our structure which is all built in to a high level language provides a particularly simple way of achieving this. As well as being simple from the point of view of the programmer, it also is simple from the point of view of the systems software and hardware, and this translates into more economical means for controlling machines or processes.

The applications program is stored as intermediates code or 'p code'. An interpreter program, which is part of the systems software, takes this p code and interprets it carrying out the necessary evaluations, actions and overhead activities. Details of such software techniques are well known and standard techniques are used throughout.

Facilities are provided so that the applications program can have access to the loop variables and so determine and use the information as to which state is active in any loop. Statements such as GET LOOP as shown above are typical.

Discussing the various types of statements,

the p code for all statements consists of two general categories of code

- firstly, one or more bytes constituting an operator which defines the statement type and is used by the interpreter to select the service routine to be used.
- secondly, optionally data as required.

The types of statements (shown with examples) that are of interest are:

- loop initialising statements

- START LOOP WITH STATE 200

The p code for start loop, consists of the operator, the loop reference, and the state reference. The loop reference may conveniently be the address of the start of the loop data in the task table. The state reference may conveniently be the address of the state in the program p code.

The service routine consists of

- oncetest (see ONCETEST routine in this specification) and skip remainder if necessary
- transfer state address into the loop variable in the correct loop location in the task table
- set oncetest flags etc as required
- save decision point data
- exit to process next operator

- RESTART 3000

- similar to above but start loop 1 at state 3000
- zero flags, registers etc. as appropriate.
- save decision point data
- exit to start of scan

- state changing statements

GOTO 201

- transfer new state address into loop variable in task table for loop currently being processed
- clear once flag
- save decision point data
- exit to process next loop

GOSUB 5000

- onctest then as GOTO, except save address of next operator and of address of state to be returned to in save area in task table for use by RETURN statement.
- save onctest flag and mask
- save decision point data
- exit to next loop

- RETURN

- get saved addresses from save table so can process remainder of calling state
- restore onctest flag and mask
- save decision point data
- set condition flag
- exit to process rest of calling state

- Two of the statements above are once only statements, namely start loop and gosub. All the other 3 function once only also by the nature of what they do, i.e. they change the active state in the loop in which they appear. Because they do this, they are not executed on the next scan through the loop. They thus have no need to refer to a ON-CETEST routine and should not be classed as once only statements. The reason gosub has this need, is that although it alters its own loop, after the return from the subroutine the calling state will become active again possibly for many scans, and it would be inappropriate to call the subroutine every scan, but calling it on a 'once per state activation' basis, with a return from a subroutine not counted as a state activation, or on an edge activation basis is logical.
- once only statements. Most statements are once only statements. The following are some other examples which can be implemented either as edge activated or once per state activation types as required.

- START TIMER 5
- INCREMENT REGISTER 17

- It will be readily apparent why it is desirable to start a timer or increment a register only once per state activation or when a condition comes true. E.g. to start the timer every scan would just hold it reset but not running while in that state, instead of resetting to 0 and then allowing it to count the time that passes.
- The execution of these statements involves little more than writing a 0 into the appropriate timer which is then counted up once per time interval by an interrupt routine until it eventually sticks at timer overflow (255 for 3 bits), or adding, one to the number stored in the appropriate register.

- Once only operation.

- The ONCETEST routine in the source code disclosure shows how to implement routines that execute their function every time the function (their state is active and their condition part is true) becomes true. There is a simplified way of providing once only operation. This involves having a single Boolean flag per loop. This flag is set by any routine which alters the state that is active in that loop. It is reset at the end of the first scan through that loop just before moving on to process the next loop, and optionally for ease at the end of every scan through the loop. It is therefore true only during the first scan through a newly active state. It is used by the service routines for any once only statements, and these execute their function only if the flag is true. The net value of this technique depends on the trade-off between a simplified interpreter and more flexibility in the language.

- Under some circumstances it can be advantageous to be able to have a once only routine repeat the execution of its function every scan. For instance to hold a timer reset until the state becomes deactivated and then automatically release it. For this purpose, a REPEAT statement can be provided which when executed by the interpreter causes the next, or subsequent statements up to an 'end repeat' statement, to execute every scan which can be done by the interpreter selecting alternative entry points to the service routines for those repeating statements, which entry points ignore ONCETEST and the first scan flag.

- every scan statement. The prime example of this is the output statement.

- OP 45 ON

This statement says effectively 'output 45 is on while the state the statement appears in is active and subject to any condition applying to it'. The output automatically turns off when the state is deactivated. It does not say

'turn output 45 on', and need another statement to turn it off later. This is achieved by having all the bits in the output map set to 0 at the beginning of each scan, and providing a service routine for the statement that sets the appropriate bit in the map each scan the state is active and the condition part is true.

5 miscellaneous statements.

- DEFINE

10 A class of DEFINE statements is provided for variables or constants. These may be manually inserted into the program or altered, or automatically inserted when a new name is recognised. They are of the form DEFINE STATE 5 = This State and basically associate the name with the numerical value of the variable/constant, and amount to a symbol table stored as part of the program. They do nothing during program execution and are just skipped, and so do not slow the interpreter significantly but they are used during program listing by the print routines.

15 - Statements can be provided to test whether a particular state is active in some defined loop. This involves going to the loop variable for the loop in question, extracting the loop reference, which may be the address of the p code for the state that is active, and if so going to the state p code and extracting the state number which may be stored as the second and third bytes of p code. A comparison is then carried out. The statements must be compatible in terms of variable types, and may be of the form

20 - GET LOOP
IF RESULT = 315
GOTO 350
25 or
- IF STATE 315 IN LOOP 3
GOTO 350

30 By way of additional disclosure for the sake of clarity, the following charts show the routines already described but in greater detail, and also some of the other supporting routines necessary to make such a controller a practical tool.

Source code has been disclosed for implementing the key routines on a particular microprocessor.

These following charts follow the source code disclosed closely if not exactly and use entry and exit points named similarly to the labels in the source code to assist with understanding of the code and cross referencing.

35 Simplified Charts.

The following charts show the broad flow of control for our implementation of the novel scanning methods disclosed.

40

45

50

55

STARTPROG

Initialise task table to
 Loop 1, state 1 only,
 loop pointer to loop 1, and p-
 code pointer to state 1.
 Initialise other variables as
 necessary.

FIRSTOP-----|

Read statement

STATE statement? -- yes ----- |

no

PAUSESTATE -- yes ----- INCT

no

Go to monitor

```

      NXTOPR
      |
      Read next statement via p-code
5      pointer
      |
      STATE, PAUSESTATE
      |
      or
10     BREAKSTATE?
      |
      no
      |
      GOTO  -- yes ----- GOTOI
15     |
      no
      |
      START? -- yes ----- STARTI
20     |
      no
      |
      GOSUB? -- yes ----- GOSUBI
25     |
      no
      |
      RETURN? -- yes ----- RETURNI
30     |
      no
      |
      Must be normal control
35     statement, therefore carry out
      the appropriate function of
      evaluating a condition or
      carrying out some action
      |
40     I ----- Advance p-code pointer to next
      statement operator
      |
      NXTOPR
45
50
55

```

INCT

Advance the loop pointer and
read address of next state
block for processing from the
loop variable indicated.
(Advancement is from the first
active loop through all active
loops in numerical order of
loop number to the last, and
then back to the first, and so
on repetitively.

FIRSTOP

GOTOI

|

5 Condition flag true? -- no -----I

yes

|

10 save decision point address and
loop number in trace table.

|

15 write address of new state block
into loop variable for the loop
being processed

|

INCT

 STARTI

|

20 Condition flag true
going transition detected? -- no ---I

25 yes

|

save decision point address and
loop number in trace table.

|

30 write address of new state block
into loop variable for the loop
number specified in the start
statement. (If the specified loop
is already active just overwrite
35 the existing value of state block
address. If the loop is not
active, slot a new loop variable
into the array in numerical order
and write both the loop number and
40 the address of the new state block
into the variable).

|

-

GOSUB I

5 Condition flag true going
 transition detected? - no ---I

 yes

10 save decision point address and
 loop number in trace table.

 save value of p-code pointer,
 value of current state block
 15 address, and condition flag
 transition information for use by
 RETURNI processing later.

 write address of new state block
 20 into loop variable for the loop
 being processed, and into p-code
 pointer

 FIRSTOP

25 RETURNI

30 Condition flag true? -- no ---I

 yes

 save decision point address and
 loop number in trace table.

35 restore value of p-code pointer,
 and value of calling state block
 address in loop variable, and
 transition information, as saved
 40 by GOSUBI processing

 I

45 Data Structure

The above flow charts depend on the following data structure and assumptions.

- 50 1. A program preferably stored in p-code, but possibly stored in some other form, consisting of at least a list of one or more state blocks.
2. Each state block consisting of a state statement delimiting the start of the block, or similar delimiting means, none, one or more statements defining control action to be carried out including control of active states and loops, and ending at the start of the next state block or at some other convenient delimiter.
- 55 3. At least some of the statements available being able to define the static conditions that shall prevail while the state is active, as opposed to a list of procedural tasks to be carried out. E.g. OP 10 ON means 'output 10 is on', and not 'turn output 10 on', (for later turning off by some other statement). Note output 10 will go off automatically when the state becomes inactive. Note that the repetitive scanning and other special operating system support combines with the language to provide this feature.

4. At least some of the statements being able to act once only when the state is activated, and to automatically skip execution on subsequent scans through that state while it stays active.
5. A list of loop variables that may be active or inactive, and if active that associate a loop number with an active state, and where the numerical order of said loop numbers control the order of scanning of said active states.
5. GOTO statements that can work from the necessary evaluation to change the active state in the loop in which the statement appears without there being any need to evaluate which state is active in that loop, and a program structure that allows this to occur.
7. START statements to make a loop variable active and load a pointer to the active state block into the loop variable.
8. GOSUB/RETURN statements that act AT THE STATE LEVEL as described.

All Drummond's basic claims of

1. Only one state active in a set at once.
2. One becoming active makes remainder in set inactive etc., apply.

Note except we can make loops active and inactive, and in the latter case make active states inactive without making any other active state in the same set active in turn.

Detail Flow Charts describing Interpreter routines.

Labels and variable names are shown in CAPITALS.

Program starts at STARTINTER

The flowcharts detail essential elements in the algorithms. They do not necessarily describe house-keeping activities included in the program listing, such as maintaining registers, because these activities are specific to particular implementations.

```

                                INCT
                                |
5      Locate address of entry in task
      table for next loop
                                |
      Put in task table pointer
                                |
10     GETSTP -----|
      Get loop number from table
                                |
                                |
15     Is it 0? -- no ----- MORE
                                |
                                yes
                                |
      Get monitor flag
                                |
                                |
      Is it up? -- yes ----- MONITOR1
                                |
                                no
25     STARTINTER-----|
      Do input output scan
                                |
                                |
30     Scan error? -- yes ----- OFFLINEX1
                                |
                                no
                                |
35     Character from terminal requests
      break in scanning? -- no ----- CSRENT
                                |
                                yes
                                |
40     Flag that terminal interrupt has
      occurred
                                |
      MONITOR1 (Go to the monitor program)
                                |
45     OFFLINEX1---- Flag that scan error has occurred -- MONITOR1

```

CSRENT

```
Initialise task table pointer
```

Get address of task from task table

MORE

Set up P-code pointer from address
of task from task table

Set decision point register to
initial value of task start

```

Initialise condition flag

```

```

Initialise ONCEMASK

```

Get first operator in state block
(can only be a state type)

STATE? --- yes ----- STATEII

הס

PAUSESTATE? --- yes----- INCT

no

(Must be BREAKSTATE)

Save state number in STATESTORE
with P-code pointer set to address
of state operator

Flag that a breakstate has been encountered

MONITOR

STATE II

I3
I2
I1

Point p-code pointer at next operator

(Amount of adjustment depends on routine just finished and where it left the pointer).

NXTOPR-----

Get operator

VBRI-----

Do calculated jump into
Interpreter jump table using value
of operator to define address in
jump table

Jump via jump table to service
routine for statement in program
represented by the particular
operator being processed

PARTICULAR SERVICE ROUTINE TO
SERVICE PARTICULAR OPERATOR

Skip function

Once a decision has been made that the condition required to carry out an action is true, then it is no longer necessary to continue evaluating the remaining terms in the condition statement because they can have no effect on the net result of the statement overall. This situation arises when an OR type statement is encountered and the condition flag is true when the statement is read. As an example, in the statement

IF IP 5 ON AND IP 6 ON OR IP 7 ON SET OP 66 ON

If both inputs 5 & 6 are on, the statement SET OP 66 ON will be carried out independent of the state of input 7, and therefore the evaluation of input 7 (and any other condition evaluating statements following) may be skipped. A special algorithm is provided for this and is as follows.

SKIP2

Increment p-code pointer to next operator

Get operator and load it into low byte of program counter so causing a jump into the jump table

In the jump table, do one of the following

- if the jump was to an entry for an IF or action operator, jump further to label VBR1

- if the jump was to an entry for an OR or an AND operator, jump back to SKIP2

This algorithm relies on having the operators coded so that their binary values are the same as the low byte of the address in the jump table at which their respective entries reside, or the value could be equal to an offset for calculating the jump address into the table.

Example Service routines.

Routine to execute statements SET OP (set output) or SET FLAG

Statements are stored in memory as p-code and these statements may conveniently be stored as 3 consecutive bytes -
byte

- 1 - operator indicating kind of statement in this case meaning SET OP or SET FLAG
- 2 - address in internal output or flag map in memory which holds the byte containing this output/flag
- 3 - mask to select actual output/flag

Note the same routine may be used for both flags and outputs and byte 1 is identical, only the address determines whether the variable is an output or flag.

SOFT

('SET OP' , 'SET FLAG')

Condition flag true?--- no ----- I2

yes

Get variable address from p-code

OR bit specified in p-code into variable

I1

Service routine for AND IP (IP means input),

OR IP
IF IP
AND FLAG
OR FLAG
IF FLAG

5

p-code is as above - operator, address, mask.

10

AIFOI
('AND IP' , 'AND FLAG')

15

Condition true?--- no ----- 12

yes

|

IIFOC

20

OIFOI
('OR IP' and 'OR FLAG')

25

Condition true?--- yes ----- SKIP2

no

|

IIFOI-----
('IF IP'
'IF FLAG'

30

Set condition true

|

IIFOC-----

35

Save value of p-code pointer as
decision point in decision point
register.

|

40

Get address of input byte in input
map from p-code.

|

45

Get mask from p-code and test
correct bit in byte selected with
address

|

Set condition flag to reflect
state of selected bit

|

50

II

HALT

HALT is a conditional operation which if activated causes transfer of processing to the next loop.

55

HALTI

|

5 Condition true? -- no -----12

yes

|

10 INCT

Note that during the processing of any one state, two state type operators are read and processed. One at the beginning of the state block and one at the end. How they are processed is different. The state type operators found at the beginning of a state block are processed as shown in the charts above. The operators found at the end of blocks are processed as below. They simply indicate the end of processing for this state, and transfer processing to the state that is active in the next loop. The operator at the end of a block is in fact the first operator in the next state block in line in the program listing.

State type operators include STATE, PAUSESTATE and BREAKSTATE

STATE
PAUSESTATE
BREAKSTATE

|

|

25 INCT

GOTO

The P-code for GOTO is - operator, new state address (i.e. address of state operator at beginning of new state block in P-code in memory).

GOTOI

|

35 Condition true?--- no ----- 12

yes

|

40 Call STOREHIST to save the latest saved value of the decision point in the trace table.

|

45 Change the task table state address for this loop to new value from p-code

|

50 Clear task table TTONCE for this loop record

|

INCT

55 Subroutine 'StoreHistory'

Uses the trace table which is a linear table in memory as a wrap around 'endless tape' storage buffer. A pointer to the last used/ next to be used location is saved as a separate pointer variable. Once the table has been filled, older data

is overwritten by new data.

STOREHIST

Tracing of this loop disabled? - yes - STOREHISTX

no
Get saved value of trace table
pointer

If pointer indicates end of
table has been reached then set
pointer to beginning and wrap
around

Write address of decision point
into trace table (address is
saved by all condition
evaluating routines and the
last saved value prior to the
action routine calling this
sub-routine is used).

Save loop number from task
table into trace table

Save new value of trace table
pointer

Return to calling routine.

Note: in our implementation, a trace enable flag is stored
encoded into the loop number highest bit. This can be done
because we implement no more than 7FH loops encoded into 3 bits.
A separate variable could be used. This variable is used for
the 'tracing of this loop disabled?' decision.

REMARK

The P-code for REMARK is - operator, address of next operator in program, remark character string, (next operator).

REMI

Get address of next operator
from p-code and put in pcode
pointer.

NXTOPR

START

START is understood more easily by understanding the following:

The task table is an array of records, some of which hold data and represent active loops that have been started, and some are empty and represent loops that have not been started. The array is dynamic in the sense that all non-empty records are packed to the start of the table and are ordered in numerical order of the value of the loop variables in the records. Inserting a new loop record into the array involves inserting it in numerical order into the packed array and if necessary moving higher order loops down the array to allow this to be done.

Each record is a loop record and has fields that hold the following simple variables -

TTLOOP - The number of the loop the record represents, and that also decides the order in which active states are scanned. Loop numbers are mutually exclusive. In our representation valid loop numbers are all less than 127. We store the loop number as 7 bits of an 8 bit variable, and use the 8th bit as a 'loop tracing disable flag', see STOREHIST. Both the loop number and the flag are stored in TTLOOP.

TTSTATE - the address of the active state for the loop, i.e. the address in memory of the program p-code for the state operator which defines the start of the program state block for the active state.

TTONCE - This is the onceflag variable used by ONCETEST

TTACCUM - The intermediate result and data transfer register for this loop. It is used by statements that manipulate numerical data.

TTEXTEN - An extension for TTACCUM

TTSTATESTR

TTONCESTR

TTRTNAD - Three variables for preserving information from TTSTATE, TTONCE and the p-code pointer respectively for use by GOSUB and RETURN statements.

The p-code for START is operator, loop number, new state address.

```

      STARTI
      |
      |
      | Call subroutine ONCETEST
      |
5     |
      |
      | Do routine this time? --- no ----- I3
      |
      | yes
      |
10    | Call subroutine STOREHIST
      |
      | Point at task table record for
      | loop currently being processed
      |
15    | Get current loop number from
      | task table
      |
      |
20    | Current loop number < loop
      | number in START statement? --- no ----- I1
      |
      | yes
      |
25    | Call subroutine SEARCHTT
      |
      | I1

```

SEARCHTT

```

5      Initialise 'number of loops
      counter' Initialise task record
      pointer
L2FD-----|
          Decrement counter
          |
10      Loop counter value shows
      searched all loops?    -yes----- NOVAD
                                (I.e. table is full)
                                |
15      no
          |
      Loop in record pointed at = 0
      i.e. empty              --yes-- LEQZ
                                |
20      no
          |
      Loop in record pointed at =
      number of loop in START
      statement 2-code        -- yes----- LEQZ
25      |
          no
          |
      Loop in record pointed at > --
      number of loop in START
      statement 2-code        -- yes----- SHLP
30      |
          no
          |
35      Increment pointer to next
      record in task table
          |
          L2FD
40
45
50
55

```

SHLP

Move whole block of records,
from that indicated by record
pointer to last record but one
inclusive, down one record in
task table towards table end to
make space for insertion of new
loop. Last loop is overwritten
but must be empty because
program range checking on loop
numbers precludes starting of
too many loops.

LEQZ-----|

Get loop number from p-code and
put in task table record
indicated by pointer

Get state address from p-code
and put in task table record
indicated by pointer

Initialise remainder of record,
i.e. ACCUM, EXTEN etc.

Return to caller.

Once test.

ONCETEST

5 Get ONCEMASK

 AND mask with TTONCE, the
 onceflag for this loop record.
 10 The result is saved value of
 condition flag as it was when
 this instruction was processed
 last scan

15 Condition flag was true? -- yes ----- ONCE1

 no

20 Condition flag is now true -- no ----- ONCE2

 yes

25 Set Onceflag bit indicated by
 mask to true, leaving rest of
 Onceflag as it was on entry to
 sub-routine.

30 Set value returned to caller to

 TRUE

 ONCE3

35 ONCE1

 Condition flag is true? --- yes ----- ONCE2

40 no

 Set Onceflag bit indicated by
 mask to false, leaving rest of
 onceflag as it was on entry to
 45 sub-routine.

ONCE2-----|

50 Set value returned to caller to

 FALSE

ONCE3-----|

55 Shift oncemask one bit

 Return to caller.

Monitor routine to display history.

```

5          DISPHIST
          |
Print '<CR,LF> Li State      1      5      10      15      20'
Print '<CR,LF>'
          |
10      Copy trace table to buffer by
        calling SETSNAP. (The trace
        table is changing all the time
        a program is running and
        causing changes of state. The
15      monitor may also be running and
        printing past history. So that
        stable data is available to
        this routine the actual table
        is copied into a buffer and the
        printing uses the buffer as
        data. The buffer also has a
        copy of the trace table
        pointer.)
          |
25      Get value of copied trace table
        pointer adjusted to be relevant
        to the copy buffer.
          |
        Set line number counter to 0
30      |
DISPHIST1-|
          |
35      Terminal request to abort
        routine?                                - yes -DISPHISTR
          |
        no
          |
40      Printed maximum lines buffer
        can hold?                                - yes- DISPHISTX
          |
        no
          |
45      Print line number and tab
        across
          |
        If pointer indicates end of
        buffer has been reached then
50      set pointer to beginning and
        wrap around
          |
          |
          |
55      |

```

5 Loop number invalid & thus ----- yes- DISPHISTX
 this buffer entry empty?

 no

10 Get loop number from buffer and
 temporarily save

15 Call subroutine STPADR to get
 state number equivalent to
 pointer value saved in buffer.
 (The pointer points to the
 decision point which is some
 way through a state block. The
 state number required is the
 number of that block)

20 Print state number

25 Use saved value of loop number
 to tab across page to column
 equivalent to loop

30 Print saved value of loop
 number

 Increment line count

 DISPHIST1

35 DISPHISTX

 Print 'END<CR,LF>' message

40 DISPHISTR-

 Return to caller

45 Subroutine STPADR

STPADR

5 Initialise pointer for STEPON
 routine to first p-code
 location in program

STPADR3-----|

10 Call subroutine STEPON

15 If operator A is STATE,
 PAUSESTATE, or BREAKSTATE then
 save address of STATE operator.
 (See description of STEPON for
 explanation of
 operator/instruction A)

20 Passed target address in p-
 code?

-- no --- STPADR3

25 yes

 Extract address of state number
 for passing to caller.

30 Return to caller

35 Monitor routine to display an expanded 'line' of the history table.

DISPLINE

5 Set Outstanding_line_count to
the number of the line to be
displayed obtained as a
parameter from the monitor
command initiating this routine

10 Number in valid range? -- no ----- DISPLINER

yes

15 Point at trace table buffer
first logical entry, i.e. the
last data entered and that is
pointed to by the table
pointer stored with the trace
20 table and copied to the buffer
as the buffer pointer

DISPLINE1-----|

25 Increment pointer by one record

30 If pointer indicates end of
buffer has been reached then
set pointer to beginning and
wrap around

35 Outstanding_line_count = 0? -- yes ----- DISPLINE3

no

Decrement Outstanding_line_count

40 DISPLINE1

DISPLINE3

Record pointed to is empty? -- yes ----- DISPLINER

no

Call STPADR to get address of
state displayed in this line
and use as p-code pointer

DISPLINE4-----

Terminal request to abort
routine?

- yes - DISPLINEX

NO

Convert p-code for one
statement to text and print it.

if at decision point then
print '** Decision point **'
<CR,LF>

Step on to next p-code operator

Is it a STATE, PAUSESTATE or
BREAKSTATE

- yes - DISPLINEX

no

DISPLINE4

DISPLINER

Print 'Line number not valid'
<CR,LF>

DISPLINEX-----

Return to caller

Display Loops i.e. display those states that are active for each of the loops that are running.

DISFLOOP

Print loop numbers as column headings

Set pointer to task table

Set loop counter to 0

GETNT-----

Increment counter

Counter = max number of loops + 1? - yes ----- PRSEX

BO

NEXE-----

Get loop number from task table

Is loop number = 0 indicating
passed end of list of running
loops with active states?

-- yes ~~~~~ PRSEX

no

This loop number = loop number
in counter?

-yes- E.DOK

по

Tab across page to next loop position

Increment loop counter

NEXE

FDOK

Get state address from task
table (address is address of
state operator at start of that
state block in program p-code.
The number of the state follows
the operator in the p-code.)

Get state number from p-code
and print it.

Increment task table pointer to
next task record

GETNT

PRSEX ----- Print <CR,LF> ---- Return to caller

Claims

1. A programmable controller (10) which executes a finite state applications program (14) and facilitates debugging of said program characterised in that said finite state applications program comprises two or more program loops (46), each of said program loops emulates a state diagram and has two or more blocks of statements (48, 50, 52), each block when executed by the controller (10) being a program state, and each program state being active only when one or more conditions or operations cause said program state to occur, and said programmable controller includes:
 - (a) means for enabling (68, 10A) two or more of said program loops to become active for execution and means for enabling (68, 10A) only one block of said blocks within any active program loop to become active for execution at any one time;
 - (b) means for recording (82, 10A) said occurrences of each said program state for later display; and
 - (c) means for recording (82, 10A) for later display said one or more conditions or operations causing the occurrence of each said program state.
2. A programmable controller as claimed in claim 1 wherein only one state can be active in each active program loop at one time, and the active state is indicated by the storage of a value in a variable (94) indicative of the active state block, that variable being assigned to indicate state activity.
3. A programmable controller as claimed in either of claims 1 or 2 wherein said means for enabling comprises at least one said variable (100, 102, 104, 106, 110), each said at least one variable being associated with one said two or more active program loops for storing an identifier (94) of the start of the active block and said variable having an identifier (92) to indicate the active program loop which includes the active block.
4. A programmable controller as claimed in either of claims 2 or 3 further comprising means for storing (68, 10A) each said variable associated with said two or more active program loops for facilitating multitasking of a plurality of said active program loops and for controlling the sequence of execution of said active program loops.
5. A programmable controller as claimed in any of the preceding claims wherein said means for recording (82, 10A) said occurrences of each said program state for later display comprises at least one second variable (70, 71, 72), each said at least one second variable being used to record an identifier indicative of a change in program state.
6. A programmable controller as claimed in claim 5 wherein said conditions or operations causing a program state to occur are indicated by said second variable (70, 71, 72) which stores an identifier of the statement where said



FDOK

|
Get state address from task
table (address is address of
state operator at start of that
state block in program p-code.
The number of the state follows
the operator in the p-code.)

|
Get state number from p-code
and print it.

|
Increment task table pointer to
next task record

GETNT

PRSEX ----- Print <CR,LF> ---- Return to caller

Claims

1. A programmable controller (10) which executes a finite state applications program (14) and facilitates debugging of said program characterised in that said finite state applications program comprises two or more program loops (46), each of said program loops emulates a state diagram and has two or more blocks of statements (48, 50, 52), each block when executed by the controller (10) being a program state, and each program state being active only when one or more conditions or operations cause said program state to occur, and said programmable controller includes:
 - (a) means for enabling (68, 10A) two or more of said program loops to become active for execution and means for enabling (68, 10A) only one block of said blocks within any active program loop to become active for execution at any one time;
 - (b) means for recording (82, 10A) said occurrences of each said program state for later display; and
 - (c) means for recording (82, 10A) for later display said one or more conditions or operations causing the occurrence of each said program state.
2. A programmable controller as claimed in claim 1 wherein only one state can be active in each active program loop at one time, and the active state is indicated by the storage of a value in a variable (94) indicative of the active state block, that variable being assigned to indicate state activity.
3. A programmable controller as claimed in either of claims 1 or 2 wherein said means for enabling comprises at least one said variable (100, 102, 104, 106, 110), each said at least one variable being associated with one said two or more active program loops for storing an identifier (94) of the start of the active block and said variable having an identifier (92) to indicate the active program loop which includes the active block.
4. A programmable controller as claimed in either of claims 2 or 3 further comprising means for storing (68, 10A) each said variable associated with said two or more active program loops for facilitating multitasking of a plurality of said active program loops and for controlling the sequence of execution of said active program loops.
5. A programmable controller as claimed in any of the preceding claims wherein said means for recording (82, 10A) said occurrences of each said program state for later display comprises at least one second variable (70, 71, 72), each said at least one second variable being used to record an identifier indicative of a change in program state.
6. A programmable controller as claimed in claim 5 wherein said conditions or operations causing a program state to occur are indicated by said second variable (70, 71, 72) which stores an identifier of the statement where said



condition or operation causing a change in program state was executed.

- 5
7. A programmable controller as claimed in either of claims 5 or 6 wherein said second variable indicates the active program loop which includes said statement.
- 10
8. A programmable controller as claimed in any one of claims 5 to 7 further comprising means for storing said second variable, said means for storing comprising a trace table (82).
- 15
9. A programmable controller as claimed in any one of the preceding claims wherein said finite state applications program further comprises compound statements which consist of a conditional part and an action part, the conditional part consisting of one or more simple condition evaluating statements, each evaluating whether some system condition is true or false and wherein the conditional part determines whether the actions contained in the action part shall be carried out.
- 20
10. A programmable controller as claimed in claim 9 wherein syntax rules are provided in said finite state applications program for statements so that when the recording of said one or more conditions or operations causing the occurrence of each said program state includes the recording of the location in the finite state applications program at which a decision was taken that a conditional part of the state activity altering compound statement is true then such information defines all the necessary information to uniquely determine which term in which conditional part of which statement in which program state caused which transition in state activity when read in the context of the finite state applications program.
- 25
11. A programmable controller as claimed in either of claims 9 or 10 wherein the recording of said one or more conditions or operations causing the occurrence of each said program state includes the recording of the location of the simple condition evaluating statement in the finite state applications program at which a decision was taken that the conditional part of the compound statement was true.
- 30
12. A programmable controller as claimed in any one of the preceding claims wherein the recording of said one or more conditions or operations causing the occurrence of each said program state includes the time at which the transition in state activity occurred.
- 35
13. A programmable controller as claimed in any one of the preceding claims wherein debugging functions are provided automatically such that they may be utilised by said programmable controller while executing said finite state applications program as a system service without the need for user intervention and said debugging functions being initiated within the execution of said programmable controller.
- 40
14. A programmable controller as claimed in claim 13 wherein said programmable controller has means which turn a program state into a BREAKSTATE which, when it becomes active, causes a predetermined debugging function to occur causing said finite state applications program to halt so that debugging and investigation processes may be initiated.
- 45
15. A programmable controller as claimed in either of claims 13 or 14 wherein said programmable controller has means to insert an optionally conditional HALT statement into a program state which, when the HALT statement is executed, causes a predetermined debug function to occur causing a halt to the processing of the program loop containing that program state but does not affect the processing of any other program loops.
- 50
16. A programmable controller as claimed in any one of claims 13 to 15 wherein said programmable controller has means which turn a program state into a PAUSESTATE which, when it becomes active, causes a predetermined debug function to occur causing a halt in processing of the program loop in which that program state occurs but does not affect the processing of any other program loops.
- 55
17. A programmable controller as claimed in any one of the preceding claims having program structure means for eliminating the need to evaluate whether any particular program state is active for the purpose of taking state dependent decisions or carrying out state dependent activities which depend on the activity of a program state in the program loop in which that state dependent statement appears.
18. A programmable controller as claimed in claim 17 wherein said program structure means comprises rules whereby all the statements which involve state dependent evaluations or actions dependent upon a particular program state



are gathered within the block for that particular program state and executing means to execute only those statements that appear in the blocks of those program states that are active in the system whereby the need to evaluate whether a particular program state is active is eliminated by the fact that only active program blocks within active program loops are associated with said variable (100, 102, 104, 106, 110) and, therefore, program blocks within program loops which are not associated with a variable (100, 102, 104, 106, 110) are inactive.

19. A programmable controller as claimed in any of the preceding claims wherein said finite state applications program is stored in pseudo code which is executed by a pseudo code interpreter program.

20. A programmable controller as claimed in any one of the preceding claims having delimiting means for delimiting the blocks of statements, said delimiting means consisting of state statements associating a state identifier with each block of statements.

21. A programmable controller (10) which executes a finite state applications program (14) and facilitates debugging of said program characterised in that said program is stored in pseudo code and executed by a pseudo code interpreter program, said finite state applications program comprising two or more program loops (46), each of said program loops emulates a state diagram and has two or more blocks of statements (48, 50, 52), each block when executed by the controller (10) being a program state, and each program state being active only when one or more conditions or operations cause said program state to occur, and said programmable controller includes means for enabling (68, 10A) two or more of said program loops to become active for execution and means for enabling (68, 10A) only one block of said blocks within each active program loop to become active for execution at any one time.

22. A programmable controller as claimed in claim 21 having means for recording (68, 94, 104) said occurrences of each said program state for later display.

23. A programmable controller as claimed in either of claims 21 or 22 having means for recording (82, 10A) for later display said one or more conditions or operations causing the occurrence of each said program state.

24. A programmable controller as claimed in any of claims 21 to 23 which progresses said finite state applications program by taking from a first initialised variable the value in that variable indicative of the start of the first program state block to be processed, processing the first state block, modifying the value stored in that or another variable if modification is directed by the processing of said first state block, maintaining any required record of current state block activity, and wherein processing the finite state applications program consists of repetitively processing each of the active state blocks in like manner in desired order.

25. A programmable controller as claimed in any one of claims 21 to 24 having delimiting means for delimiting the blocks of statements, said delimiting means consisting of state statements associating a state identifier with each block of statements.

26. A programmable controller as claimed in any one of claims 21 to 25 wherein only one state can be active in each active program loop at one time, and the active state is indicated by the storage of a value in a variable (94) indicative of the active state block, that variable being assigned to indicate state activity.

27. A programmable controller as claimed in any one of claims 21 to 26 wherein said means for enabling comprises at least one said variable (100, 102, 104, 106, 110), each said at least one variable being associated with one said two or more active program loops for storing an identifier (94) of the start of the active block and said variable having an identifier (92) to indicate the active program loop which includes the active block.

28. A programmable controller as claimed in either of claims 26 or 27 further comprising means for storing (68, 10A) each said variable associated with said two or more active program loops for facilitating multitasking of a plurality of said active program loops and for controlling the sequence of execution of said active program loops.

29. A programmable controller as claimed in any one of claims 21 to 28 wherein said means for recording (82, 10A) said occurrences of each said program state for later display comprises at least one second variable (70, 71, 72), each said at least one second variable being used to record an identifier indicative of a change in program state.

30. A programmable controller as claimed in claim 29 wherein said conditions or operations causing a program state to occur are indicated by said second variable (70, 71, 72) which stores an identifier of the statement where said



condition or operation causing a change in program state was executed.

31. A programmable controller as claimed in either of claims 29 or 30 wherein said second variable indicates the active program loop which includes said statement.

32. A programmable controller as claimed in any one of claims 29 to 31 further comprising means for storing said second variable, said means for storing comprising a trace table (82).

Patentansprüche

1. Eine programmierbare Steuereinheit (10), die ein Anwenderprogramm mit endlichen Zuständen ausführt und eine Fehlersuche des Programms ermöglicht, dadurch gekennzeichnet, daß das Anwenderprogramm mit endlichen Zuständen zwei oder mehr Programmschleifen (46) aufweist, wobei jede Programmschleife ein Zustandsdiagramm emuliert und zwei oder mehrere Blöcke von Anweisungen (48, 50, 52) hat und jeder Block bei einer Ausführung durch die Steuereinheit (10) ein Programmzustand ist und wobei jeder Programmzustand nur dann aktiv ist, wenn eine oder mehrere Bedingungen oder Zustände bewirken, daß der Programmzustand auftritt, und wobei die programmierbare Steuereinheit aufweist:

(a) Mittel zum Freigeben (68, 10A) von zwei oder mehr der Programmschleifen für ein Aktivwerden zur Ausführung und Mittel zum Freigeben (68, 10A) lediglich eines Blocks der Blöcke innerhalb jeder aktiven Programmschleife, um für die jederzeitige Ausführung aktiv zu werden;

(b) Mittel zum Aufzeichnen (82, 10A) des Auftretens jedes Programmzustands für die spätere Darstellung; und

(c) Mittel zum Aufzeichnen (82, 10A) für die spätere Darstellung der einen oder der mehreren Bedingungen und Operationen, die das Auftreten jedes der Programmzustände verursachen.

2. Eine programmierbare Steuereinheit nach Anspruch 1, wobei in jeder aktiven Programmschleife zu einem Zeitpunkt nur ein Zustand aktiv sein kann und der aktive Zustand durch das Speichern eines Wertes in einer Variablen (94) angegeben ist, die für den aktiven Zustandsblock indikativ ist, wobei die Variable dazu bestimmt ist, die Aktivität des Zustands anzugeben.

3. Eine programmierbare Steuereinheit entweder nach Anspruch 1 oder nach Anspruch 2, wobei das Mittel zum Freigeben wenigstens eine solche Variable (100, 102, 104, 106, 110) aufweist, jede der wenigstens einen Variable einer der beiden oder mehreren aktiven Programmschleifen zum Speichern eines Identifizierers (94) des Starts des aktiven Blocks zugehörig ist und die Variable einen Identifizierer (92) hat, um die aktive Programmschleife, die den aktiven Block aufweist, anzugeben.

4. Eine programmierbare Steuereinheit entweder nach Anspruch 2 oder nach Anspruch 3, weiter mit einem Mittel zum Speichern (68, 10A) der Variablen, die den zwei oder mehr aktiven Programmschleifen zugehörig sind, zum Ermöglichen eines Mehrprozeßbetriebs einer Mehrzahl der aktiven Programmschleifen und zum Steuern der Abfolge der Ausführung der aktiven Programmschleifen.

5. Eine programmierbare Steuereinheit nach einem der vorangehenden Ansprüche, wobei das Mittel zum Aufzeichnen (82, 10A) des Auftretens jedes Programmzustands für die spätere Darstellung wenigstens eine zweite Variable (70, 71, 72) aufweist, und die wenigstens eine zweite Variable verwendet wird, um einen Identifizierer aufzuzeichnen, der für eine Änderung des Programmzustands indikativ ist.

6. Eine programmierbare Steuereinheit nach Anspruch 5, wobei die Bedingungen oder Operationen, die verursachen, daß ein Programmzustand auftritt, durch die zweite Variable (70, 71, 72) angegeben werden, die einen Identifizierer der Anzeige speichert, wo der Zustand oder die Operation, die eine Änderung des Programmzustands verursacht, ausgeführt wurde.

7. Eine programmierbare Steuereinheit entweder nach Anspruch 5 oder nach Anspruch 6, wobei die zweite Variable die aktive Programmschleife angibt, die die Anweisung beinhaltet.

8. Eine programmierbare Steuereinheit nach einem der Ansprüche 5 bis 7, weiter mit Mitteln zum Speichern der zwei-



ten Variable, wobei das Mittel zum Speichern eine Ablaufabelle (82) aufweist.

- 5 9. Eine programmierbare Steuereinheit nach einem der vorangehenden Ansprüche, wobei das Anwenderprogramm mit endlichen Zuständen weiter zusammengesetzte Anweisungen aufweist, die aus einem Bedingungsteil und einem Aktionsteil bestehen, wobei der Bedingungsteil aus einer oder mehreren einfachen Bedingungen, die Anweisungen bewerten, besteht, die beurteilen, ob eine Systembedingung wahr oder falsch ist, und wobei der Bedingungsteil bestimmt, ob die Aktionen, die in dem Aktionsteil beinhaltet sind, ausgeführt werden sollen.
- 10 10. Eine programmierbare Steuereinheit nach Anspruch 9, wobei Syntaxregeln in dem Anwenderprogramm mit endlichen Zuständen vorgesehen sind für Anweisungen der Art, daß dann, wenn das Aufzeichnen der einen oder mehreren Bedingungen oder Operationen, die das Auftreten jedes Programmzustands verursachen, das Aufzeichnen des Orts in dem Anwenderprogramm mit endlichen Zuständen, an dem die Entscheidung getroffen wurde, das ein Bedingungsteil der den aktiven Zustand ändernden zusammengesetzten Anweisung wahr ist, beinhalten, diese Information alle erforderlichen Informationen bestimmt, um ausschließlich zu bestimmen, welcher Ausdruck in welchem Bedingungsteil welcher Anweisung in welchem Programmzustand welchen Übergang in eine Zustandsaktivität verursacht, wenn diese in dem Zusammenhang mit dem Anwenderprogramm mit endlichen Zuständen gelesen wird.
- 15 11. Eine programmierbare Steuereinheit entweder nach Anspruch 9 oder nach Anspruch 10, wobei das Aufzeichnen der einen oder mehreren Bedingungen oder Operationen, die das Auftreten jedes Programmzustands verursachen, das Aufzeichnen des Orts der die einfache Bedingung bewertenden Anweisung in dem Anwenderprogramm mit endlichen Zuständen, bei der die Entscheidung gefallen ist, daß der Bedingungsteil der zusammengesetzten Anweisung wahr war, beinhaltet.
- 20 12. Eine programmierbare Steuereinheit nach einem der vorangehenden Ansprüche, wobei das Aufzeichnen der einen oder mehreren Bedingungen oder Operationen, die das Auftreten jedes Programmzustands verursacht, den Zeitpunkt beinhaltet, zu dem der Übergang in die Zustandsaktivität aufgetreten ist.
- 25 13. Eine programmierbare Steuereinheit nach einem der vorangehenden Ansprüche, wobei Fehlersuchfunktionen derart automatisch vorgesehen sind, daß diese von der programmierbaren Steuereinheit während der Ausführung des Anwenderprogramms mit endlichen Zuständen als eine Systemleistung ohne das Erfordernis eines Eingreifens des Verwenders verwendet werden können und daß die Fehlersuchfunktionen innerhalb der Ausführung der programmierbaren Steuereinheit initiiert werden.
- 30 14. Eine programmierbare Steuereinheit nach Anspruch 13, wobei die programmierbare Steuereinheit ein Mittel aufweist, das einen Programmzustand in einen BREAKSTATE bringt, der, wenn er aktiv wird, eine vorgegebene Fehlersuchfunktion ausführt, die auftritt, um zu bewirken, daß das Anwenderprogramm mit endlichen Zuständen anhält, so daß die Fehlersuche und Überprüfungsvorgänge initiiert werden können.
- 35 15. Eine programmierbare Steuereinheit entweder nach Anspruch 13 oder nach Anspruch 14, wobei die programmierbare Steuereinheit ein Mittel hat, um eine optionale Konditional-HALT-Anweisung in einen Programmzustand einzubringen, der, wenn die HALT-Anweisung ausgeführt wird, eine vorgegebene Fehlersuchfunktion verursacht, um ein Anhalten der Ausführung der Programmschleife, die den Programmzustand beinhaltet, zu verursachen, jedoch die Verarbeitung jeder anderen Programmschleife nicht beeinflußt.
- 40 16. Eine programmierbare Steuereinheit nach einem der Ansprüche 13 bis 15, wobei die programmierbare Steuereinheit ein Mittel hat, das einen Programmzustand in einen PAUSESTATE bringt, der dann, wenn er aktiv wird, eine vorgegebene Fehlersuchfunktion verursacht, um ein Stoppen der Ausführung der Programmschleife, in der der Programmzustand auftritt, zu bewirken, jedoch die Ausführung der anderen Programmschleifen nicht beeinflußt.
- 45 17. Eine programmierbare Steuereinheit nach einem der vorangehenden Ansprüche, mit einer Programmstruktur zum Vermeiden des Erfordernisses der Bewertung, ob ein bestimmter Programmzustand aktiv ist, zum Zweck des Ausführens von vom Zustand abhängigen Entscheidungen oder zum Ausführen von vom Zustand abhängigen Aktivitäten, die von der Aktivität des Programmzustands in der Programmschleife, in der die von dem Zustand abhängige Anweisung auftritt.
- 50 18. Eine programmierbare Steuereinheit nach Anspruch 17, wobei das Programmstrukturmittel Regeln aufweisen, durch die alle Anweisungen, die von dem Zustand abhängige Bewertungen oder von einem bestimmten Programm-
- 55



zustand abhängige Aktionen betreffen, in dem Block für den jeweiligen Programmzustand zusammengefaßt sind und Ausführungsmittel zum Ausführen lediglich derjenigen Anweisungen, die in den Blöcken der Programmzustände auftreten, die in dem System aktiv sind, aufweist, wodurch das Erfordernis zur Bewertung, ob ein bestimmter Programmzustand aktiv ist, durch die Tatsache vermieden wird, daß nur aktiven Blöcke innerhalb aktiver Programmschleifen Variable (100, 102, 104, 106, 110) zugehörig sind und daher Programmblöcke in Programmschleifen, denen keine Variablen (100, 102, 104, 106 110) zugehörig sind, inaktiv sind.

19. Eine programmierbare Steuereinheit nach einem der vorangehenden Ansprüche, wobei das Anwenderprogramm für endliche Zustände in einem Pseudocode gespeichert ist, der von einem Pseudocode-Übersetzerprogramm ausgeführt wird.

20. Eine programmierbare Steuereinheit nach einem der vorangehenden Ansprüche, mit einem Begrenzungsmittel zum Begrenzen der Blöcke von Anweisungen, wobei das Begrenzungsmittel aus Zustandsanweisungen besteht, das einem Zustandsidentifizierer mit jedem Block von Anweisungen zugehörig ist.

21. Eine programmierbare Steuereinheit (10), die ein Anwenderprogramm (14) mit endlichen Zuständen ausführt und eine Fehlersuche des Programms erlaubt, dadurch gekennzeichnet, daß das Programm in einem Pseudocode gespeichert ist und von einem Pseudocode-Übersetzerprogramm ausgeführt wird, wobei das Anwenderprogramm mit endlichen Zuständen zwei oder mehrere Programmschleifen (46) aufweist, die jeweils ein Zustandsdiagramm emulieren oder zwei oder mehrere Blöcke von Anweisungen (48, 50, 52) aufweisen, die dann, wenn sie von der Steuereinheit (10) ausgeführt werden, ein Programmzustand sind, und jeder Programmzustand nur dann aktiv ist, wenn eine oder mehrere Bedingungen oder Operationen bewirken, daß der Programmzustand auftritt und wobei die programmierbare Steuereinheit Mittel zum Freigeben (68, 10A) von zwei oder mehreren der Programmschleifen aufweist, damit diese für eine Ausführung aktiv werden und Mitteln zum Freigeben (58, 10A) lediglich eines Blocks der Blöcke innerhalb jeder aktiven Programmschleife, um zu jedem Zeitpunkt für eine Ausführung aktiv zu werden.

22. Eine programmierbare Steuereinheit nach Anspruch 21, mit Mitteln zum Aufzeichnen (68, 94, 104) des Auftretens jedes der Programmzustände für eine spätere Darstellung.

23. Eine programmierbare Steuereinheit entweder nach Anspruch 21 oder nach Anspruch 22, mit Mitteln zum Aufzeichnen (82, 10A) des einen oder der mehreren Bedingungen oder Operationen, die das Auftreten jedes der Programmzustände verursachen, für eine spätere Darstellung.

24. Eine programmierbare Steuereinheit nach einem der Ansprüche 21 bis 23, die das Anwenderprogramm mit endlichen Zuständen ausführt, in dem von einer zunächst initialisierten Variable der Wert, in der Variablen, die für den Start des ersten Programmzustandsblocks, der zu verarbeiten ist, indikativ ist, genommen wird, der erste Zustandsblocks verarbeitet wird, der Wert, der in dieser oder einer anderen Variable gespeichert ist, modifiziert wird, wenn eine Modifikation durch die Verarbeitung des ersten Zustandsblocks veranlaßt wird, jede erforderliche Aufzeichnung der jeweiligen Zustandsblocksaktivität beibehalten wird, und wobei das Verarbeiten des Anwenderprogramms mit endlichen Zuständen aus einem sich wiederholenden Verarbeiten jedes der aktiven Zustandsblöcke in derselben Weise in der gewünschten Reihenfolge besteht.

25. Eine programmierbare Steuereinheit nach einem der Ansprüche 21 bis 24, mit Begrenzungsmitteln zum Begrenzen der Blöcke von Anweisungen, wobei das Begrenzungsmittel aus Zustandsanweisungen besteht, die einem Zustandsidentifizierer für jeden Block von Anweisungen zugehörig sind.

26. Eine programmierbare Steuereinheit nach einem der Ansprüche 21 bis 25, wobei lediglich ein Zustand in jeder aktiven Programmschleife zu einem Zeitpunkt aktiv sein kann und der aktive Zustand durch das Speichern eines Wertes in einer Variablen (94) angegeben wird, die für den aktiven Zustandsblock indikativ ist, wobei die Variable zum Angeben der Zustandsaktivität zugeordnet ist.

27. Eine programmierbare Steuereinheit nach einem der Ansprüche 21 bis 26, wobei das Mittel zum Freigeben wenigstens eine der Variablen (100, 102, 104, 106 110) aufweist, wobei die wenigstens eine Variable mit einer der beiden oder mehreren aktiven Programmschleifen zum Speichern eines Identifizierers (94) des Starts des aktiven Blocks zugehörig ist und die Variable einen Identifizierer (92) hat, um die aktive Programmschleife anzugeben, die den aktiven Block aufweist.

28. Eine programmierbare Steuereinheit nach Anspruch 26 oder nach Anspruch 27, weiter mit Mitteln zum Speichern



(68, 10A) jeder der Variablen, die mit den beiden oder mehreren aktiven Programmschleifen zum Ermöglichen eines Mehrprozeßbetriebs einer Mehrzahl der aktiven Programmschleifen und zum Steuern der Abfolge der Ausführung der aktiven Programmschleifen zugehörig ist.

- 5 29. Eine programmierbare Steuereinheit nach einem der Ansprüche 21 bis 28, wobei das Mittel zum Aufzeichnen (82, 10A) des Auftretens jedes Programmzustands für eine spätere Darstellung wenigstens eine zweite Variable (70, 71, 72) aufweist, wobei die wenigstens eine zweite Variable verwendet wird, um einen Identifizierer aufzuzeichnen, der für eine Änderung des Programmzustands indikativ ist.
- 10 30. Eine programmierbare Steuereinheit nach Anspruch 29, wobei die Bedingungen oder Operationen, die das Auftreten eines Programmzustands verursachen, durch die zweite Variable (70, 71, 72) angegeben werden, die einen Identifizierer der Anweisung speichert, bei der die Bedingung oder die Operation, die eine Änderung des Programmzustands verursacht, ausgeführt wurde.
- 15 31. Eine programmierbare Steuereinheit entweder nach Anspruch 29 oder nach Anspruch 30, wobei die zweite Variable die aktive Programmschleife angibt, die die Anweisung beinhaltet.
32. Eine programmierbare Steuereinheit nach einem der Ansprüche 29 bis 31, weiter mit Mitteln zum Speichern der zweiten Variable, wobei das Mittel zum Speichern eine Ablaufabelle (82) aufweist.

Revendications

- 25 1. Une commande programmable (10), exécutant un programme d'applications d'état fini (14) et facilitant le débogage dudit programme, caractérisé en ce que ledit programme d'applications d'état fini comprend deux boucles de programme (46) ou plus, chacune desdites boucles de programme émulant un diagramme d'état et ayant deux blocs d'instructions (48, 50, 52) ou plus, chaque bloc, lorsqu'il a été exécuté par la commande (10), étant un état du programme, et chaque état de programme étant actif seulement lorsque une ou plusieurs conditions ou opérations provoquent l'apparition dudit état de programme et ladite commande programmable comprenant :
 - 30 (a) des moyens de validation (68, 10A), concernant deux ou plus desdites boucles de programme, afin de les rendre actives et permettre leur exécution et des moyens de validation (68, 10A) n'opérant que sur un bloc parmi lesdits blocs dans ladite boucle de programme active, afin de l'activer et d'en permettre l'exécution à un moment donné quelconque;
 - 35 (b) un moyen d'enregistrement (82, 10A) desdites manifestations de chaque état de programme en vue d'un affichage ultérieur; et
 - 40 (c) un moyen d'enregistrement (82, 10A) destiné à un affichage ultérieur desdites une ou plusieurs conditions ou opérations, provoquant l'apparition de chaque état de programme.
- 45 2. Une commande programmable selon la revendication 1, dans laquelle un seul état peut être actif dans chaque boucle de programme active à la fois et l'état actif étant indiqué par la mise en mémoire d'une valeur dans une variable (94) indicative du bloc à l'état actif, cette variable étant affectée pour indiquer l'activité d'état.
- 50 3. Une commande programmable selon l'une quelconque des revendications 1 ou 2, dans laquelle lesdits moyens de validation comprennent au moins une desdites variables (100, 102, 104, 106, 110), chaque dite au moins une variable étant associée à une boucle de programme active parmi les deux boucles ou plus, afin de stocker un identificateur (94) du début du bloc actif et ladite variable ayant un identificateur (92) destiné à indiquer quelle est la boucle de programme active comprenant le bloc actif.
- 55 4. Une commande programmable selon la revendication 2 ou 3, comprenant en outre un moyen de stockage (68, 10A) de chaque variable associée auxdites boucles de programme actives, qui sont au nombre de deux ou plus, afin de faciliter un fonctionnement multitâche d'une pluralité de dites boucles de programme actives et pour commander la séquence d'exécution desdites boucles de programme actives.
5. Une commande programmable selon l'une quelconque des revendications précédentes, dans laquelle ledit moyen d'enregistrement (82, 10A) desdites manifestations de chaque état de programme afin d'avoir un affichage ultérieur



comprend au moins une deuxième variable (70, 71, 72), chaque dite au moins une deuxième variable étant utilisée pour enregistrer un identificateur indicatif d'une modification dans l'état de programme.

- 5 6. Une commande programmable selon la revendication 5, dans laquelle lesdites conditions ou opérations provoquant la manifestation d'un état de programme sont indiquées par ladite deuxième variable (70, 71, 72) qui stocke un identificateur de l'instruction à laquelle ladite condition ou opération provoquant une modification dans l'état de programme a été exécutée.
- 10 7. Une commande programmable selon l'une des revendications 5 ou 6, dans laquelle ladite deuxième variable indique quelle est la boucle de programme active qui comprend ladite instruction.
8. Une commande programmable selon l'une quelconque des revendications 5 à 7, comprenant en outre un moyen de stockage de ladite deuxième variable, ledit moyen de stockage comprenant une table de pistage-analyse (82).
- 15 9. Une commande programmable selon l'une quelconque des revendications précédentes, dans laquelle ledit programme d'applications d'état fini comprend en outre des instructions composites constituées d'une partie conditionnelle et d'une partie opérationnelle, la partie conditionnelle étant constituée d'une ou plusieurs instructions d'évaluation de condition unique, évaluant chacune si une condition système donnée est vraie ou fausse et dans laquelle la partie conditionnelle détermine si les actions contenues dans la partie opérationnelle vont être effectuées.
- 20 10. Une commande programmable selon la revendication 9, dans laquelle des règles de syntaxe sont prévues dans ledit programme d'application d'état fini pour des instructions de sorte que lorsque l'enregistrement desdites une ou plusieurs conditions ou opérations provoquant l'apparition dudit état de programme comprend l'enregistrement de l'emplacement, dans le programme d'applications d'état fini auquel une décision a été prise dictant qu'une partie conditionnelle de l'activité d'état modifiant l'instruction composite est vraie, alors cette information définit la totalité de l'information nécessaire pour déterminer de manière unique quel terme dans quelle partie conditionnelle de quelle instruction dans quel état de programme a provoqué quelle transition dans l'activité d'état lorsqu'il a été lu dans le contexte du programme d'applications d'état fini.
- 25 11. Une commande programmable selon l'une des revendications 9 ou 10, dans laquelle l'enregistrement desdites une ou plusieurs conditions ou opérations provoquant la manifestation de chaque dit état de programme comprend l'enregistrement de l'emplacement de l'instruction d'évaluation de condition unique dans le programme d'applications d'état fini auquel une décision a été prise dictant que la partie conditionnelle de l'instruction composite était vraie.
- 30 12. Une commande programmable selon l'une quelconque des revendications précédentes, dans laquelle l'enregistrement desdites une ou plusieurs conditions ou opérations provoquant la manifestation dudit état de programme comprend le moment auquel la transition de passage à l'activité d'état s'est produite.
- 35 13. Une commande programmable selon l'une quelconque des revendications précédentes, dans laquelle les fonctions de débogage sont conduites automatiquement, de manière que, lorsqu'elles peuvent être utilisées par ladite commande programmable tout en exécutant ledit programme d'applications d'état fini, à titre de service du système, sans nécessiter d'intervention utilisateur et lesdites fonctions de débogage étant initialisées dans l'exécution de ladite commande programmable.
- 40 14. Une commande programmable selon la revendication 13, dans laquelle ladite commande programmable comporte des moyens qui provoquent le passage d'un état de programme en un ETAT INTERRUPTION qui, lorsqu'il devient actif, provoque l'exécution d'une fonction de débogage prédéterminée, faisant que ledit programme d'applications d'état fini s'arrête, de sorte qu'il peut y avoir début des processus de débogage et d'investigation.
- 45 15. Une commande programmable selon l'une des revendications 13 ou 14, dans laquelle ladite commande programmable comporte des moyens pour insérer une condition optionnelle ARRÊT dans un état de programme qui, lorsque l'état ARRÊT est exécuté, provoque l'exécution d'une fonction de débogage prédéterminée entraînant l'arrêt du traitement de la boucle de programme contenant cet état de programme mais n'affectant pas le traitement de l'une quelconque des autres boucles de programme.
- 50 16. Une commande programmable selon l'une quelconque des revendications 13 à 15, dans laquelle ladite commande programmable comporte un moyen qui provoque le passage de l'état de programme à l'ETAT PAUSE qui, lorsqu'il
- 55



devient actif, provoque l'exécution d'une fonction de débogage prédéterminée provoquant l'arrêt du traitement de la boucle de programme dans laquelle cet état de programme se produit, mais n'affecte pas le traitement de l'une quelconque des autres boucles de programme.

- 5 17. Une commande programmable selon l'une quelconque des revendications précédentes, ayant un moyen de structure de programme destiné à éliminer la nécessité d'évaluer si un état de programme particulier est actif dans le but de prendre des décisions dépendantes de l'état ou d'exécuter des activités dépendantes de l'état, dépendant de l'activité d'un état de programme dans la boucle de programme dans laquelle cette instruction dépendante de l'état se produit.
- 10 18. Une commande programmable selon la revendication 17, dans laquelle ledit moyen de structure de programme comprend des règles, de manière que toutes les instructions qui font appel à des évaluations dépendantes de l'état ou à des actions dépendantes d'un état de programme particulier soient collectées dans le bloc pour cet état de programme particulier, et des moyens d'exécution étant prévus pour n'exécuter que les instructions se manifestant dans les blocs des états de programme qui sont actifs dans le système, de manière que la nécessité d'évaluer si un état de programme particulier est actif soit éliminée par le fait que seuls des blocs de programme actifs dans les boucles de programme actives sont associés à ladite variable (100, 102, 104, 106, 110) et que, par conséquent, les blocs de programme se trouvant dans des boucles de programme qui ne sont pas associées à la variable (100, 102, 104, 106, 110) soient inactifs.
- 15 19. Une commande programmable selon l'une quelconque des revendications précédentes, dans laquelle ledit programme d'applications d'état fini est stocké sous un pseudo-code qui est exécuté par un programme interpréteur de pseudo-codes.
- 20 20. Une commande programmable selon l'une quelconque des revendications précédentes, ayant des moyens de délimitation destinés à délimiter les blocs d'instructions, lesdits moyens de délimitation étant constitués d'instructions d'état associant un identificateur d'état à chacun des blocs d'instructions.
- 25 21. Une commande programmable (10) exécutant un programme d'applications d'état fini (14) et facilitant le débogage dudit programme, caractérisé en ce que ledit programme est stocké sous un pseudo-code et exécuté par un programme interpréteur de pseudo-codes, ledit programme d'applications d'état fini comprenant deux ou plusieurs boucles de programme (46), chacune desdites boucles de programme émulant un diagramme d'état et ayant deux ou plus blocs d'instructions (48, 50, 52), chaque bloc, lorsqu'il est exécuté par la commande (10) étant un état de programme et chaque état de programme étant actif seulement lorsqu'une ou plusieurs conditions ou opérations provoquent l'apparition dudit état de programme, et ladite commande programmable comprenant des moyens de validation (68, 10A), pour valider deux ou plusieurs boucles de programme afin de les rendre actives pour une exécution et des moyens de validation (68, 10A) pour ne valider qu'un bloc parmi lesdits blocs dans chaque boucle de programme active afin de le rendre actif et permettre une exécution à tout moment.
- 30 22. Une commande programmable selon la revendication 21, ayant un moyen d'enregistrement (68, 94, 104), destiné à enregistrer les manifestations de chaque état de programme en vue d'un affichage ultérieur.
- 35 23. Une commande programmable selon la revendication 21 ou 22, ayant des moyens d'enregistrement (82, 10A), destinés à un affichage ultérieur desdites une ou plusieurs conditions ou opérations provoquant l'apparition dudit état de programme.
- 40 24. Une commande programmable selon l'une quelconque des revendications 21 à 23, provoquant la progression dudit programme d'applications d'état fini en prenant depuis une première variable initialisée la valeur dans cette variable qui soit indicative de l'état du premier état de bloc de programme à traiter, en effectuant le traitement dudit premier bloc d'état, la modification de la valeur stockée dans cette variable ou dans une autre si la modification est concernée par le traitement dudit premier bloc d'état, le maintien de tout enregistrement requis de l'activité de bloc d'état en cours et dans lequel le traitement du programme d'applications d'état fini consiste en un traitement répétitif de chacun desdits blocs d'état actifs d'une manière analogue, dans l'ordre souhaité.
- 45 25. Une commande programmable selon l'une quelconque des revendications 21 à 24, comportant des moyens de délimitation destinés à délimiter les blocs d'instructions, lesdits moyens de délimitation étant constitués d'instructions d'état associant un identificateur d'état à chaque bloc d'instructions.
- 50
- 55



26. Une commande programmable selon l'une quelconque des revendications 21 à 25, dans laquelle seul un état peut être actif dans chaque boucle de programme actif à la fois, et l'état actif étant indiqué par le stockage d'une valeur dans une variable (94), indicative du bloc d'état actif, cette variable étant affectée pour indiquer l'activité d'état.
- 5 27. Une commande programmable selon l'une quelconque des revendications 21 à 26, dans laquelle ledit moyen de validation comprend au moins une variable (100, 102, 104, 106, 110), chaque au moins une variable étant associée à l'une des deux boucles de programme actives ou plus, afin de stocker un identificateur (94) du début du bloc actif et ladite variable ayant un identificateur (92) destiné à indiquer la boucle de programme active comprenant le bloc actif.
- 10 28. Une commande programmable selon la revendication 26 ou 27, comprenant en outre un moyen de stockage (68, 10A), destiné à stocker chaque variable associée auxdites deux boucles de programme actives ou plus, afin de faciliter un travail en multitâche de la pluralité desdites boucles de programme actives et afin de contrôler la séquence d'exécution desdites boucles de programme actives.
- 15 29. Une commande programmable selon l'une quelconque des revendications 21 à 28, dans laquelle ledit moyen d'enregistrement (82, 10A) enregistrant lesdites apparitions de chaque état de programme en vue d'un affichage ultérieur comprend au moins une deuxième variable (70, 71, 72), chaque dite au moins une deuxième variable étant utilisée pour enregistrer un identificateur qui est indicatif d'une modification dans l'état de programme.
- 20 30. Une commande programmable selon la revendication 29, dans laquelle lesdites conditions ou opérations provoquant la manifestation d'un état de programme sont indiquées par ladite deuxième variable (70, 71, 72) qui stocke un identificateur de l'instruction à laquelle ladite condition ou opération provoquant une modification de l'état de programme a été exécutée.
- 25 31. Une commande programmable selon l'une des revendications 29 ou 30, dans laquelle ladite deuxième variable indique la boucle de programme active comprenant ladite instruction.
- 30 32. Une commande programmable selon l'une quelconque des revendications 29 à 31, comprenant en outre un moyen de stockage de ladite deuxième variable, ledit moyen de stockage comprenant une table de pistage-analyse (82).
- 35
- 40
- 45
- 50
- 55