Ted Carmely is involved in data acquisition and control, laboratory automation, and real-time systems. His twenty-two years of experience in systems software have included the design of compilers, operating systems, and communications programs, with an occasional foray into digital logic design at the Jet Propulsion Laboratory, Magnavox Research Research Labs, and Hughes Research Labs. He has taught courses in mathematics and computer science at various colleges in Southern California, including “Real-Time Computer Systems” at UCLA. Carmely received his BA in math at UCLA, an MA in math at California State University, and an MS in computer science at USC. When away from computers, he likes to play Mozart and Dixieland on his clarinet.
INTRODUCTION

While the principles of Finite State Machines (FSMs) are familiar to many computer professionals, the techniques used to design such devices are not commonly used in software development.

Even when it is obvious from the start that a required piece of code represents a FSM (e.g., a language parser, a command interpreter, etc.), we rarely go beyond sketching a state flow diagram which eventually gets trashed long before the completion of the project.

This paper will examine a simple and practical method for creating software which is based on the idea that many computer programs have all of the properties of Finite State Machines and can thus be implemented effectively by applying some of the procedures used to design those machines.

The method we will be discussing has the following features:

° It is applicable at the initial stages of conceptual design as well as at the module coding level.
° It produces extensible and maintainable code.
° It reduces code complexity.
° The design, code, and documentation are tightly coupled.
° It incorporates an effective debugging method.
° It allows software design and verification "by committee".

What justifies applying the FSM paradigm to solving some software design problems and which kind of problems is this approach best suited for? It is fairly obvious that any software module will contain a bounded number of instructions, it will have a finite number of memory element in the form of constant and variables, and the inputs will have a bounded domain, even though that domain may sometimes be too large for any practical use.

Thus all software modules satisfy, at least in principle, the requirements of a finite state machine.

However, if you consider a typical computational routine, for instance the cube root of a 64-bit floating point number, you can easily conclude that the Finite State approach would be highly inappropriate even though it still satisfies all of the requirements of a FSM.

The difficulty lies of course in the fact that while the set of 64-bit floating point numbers is indeed finite, it would be highly impractical to attempt to enumerate that entire set in order to apply the Finite State paradigm.

On the other hand, we know from past encounters with parsers, command interpreters, or device drivers that the FSM model can be very effective in some situations.

In fact, the class of problems where the Finite State model can be put to good use is much larger than the limited linguistic applications we had to muddle through in a course on Compiler design.

In general, most software problems which have a manageable number of valid inputs and are "control intensive" rather than algorithmic, i.e., they contain a substantial number of control-flow statements (loops, conditional and unconditional branches, etc.) compared to computational statements, can be good candidates for the Finite State approach.

AN EXAMPLE

In order to prove our point and at the same time demonstrate the details of the technique, we will discuss an example of a software application which appears to be rather far removed from FSMs: a patient data management system for a medical office.

You are cautioned that this is an oversimplified academic exercise and may bear no resemblance to any real-world system.

We will concentrate on creating the top-level user interface and leave the rest to the ingenuity of the reader.

Let's assume that the design specs for our program, called PATINFO, require a hierarchical structure for the various display screens and that function keys will be used to select the desired operation and to move from screen to screen.

Also, the bottom line on the monitor will display a menu of the appropriate entry keys, together with a short description of their function.

The menu hierarchy is shown in figure 1. Each selection is labelled with the appropriate function button; the initial screen allows for four possible selections: BILLING, INSURANCE, PATIENT DATA, and EXIT PROGRAM.

Note that the last option is selected via function button F10; we'll consistently reserve that button to back up from a screen at any level to its parent screen.
INTRODUCTION

While the principles of Finite State Machines (FSMs) are familiar to many computer professionals, the techniques used to design such devices are not commonly used in software development.

Even when it is obvious from the start that a required piece of code represents a FSM (e.g., a language parser, a command interpreter, etc.), we rarely go beyond sketching a state flow diagram which eventually gets trashed long before the completion of the project.

This paper will examine a simple and practical method for creating software which is based on the idea that many computer programs have all of the properties of Finite State Machines and can thus be implemented effectively by applying some of the procedures used to design those machines.

The method we will be discussing has the following features:

° It is applicable at the initial stages of conceptual design as well as at the module coding level.
° It produces extensible and maintainable code.
° It reduces code complexity.
° The design, code, and documentation are tightly coupled.
° It incorporates an effective debugging method.
° It allows software design and verification "by committee".

What justifies applying the FSM paradigm to solving some software design problems and which kind of problems is this approach best suited for? It is fairly obvious that any software module will contain a bounded number of instructions, it will have a finite number of memory elements in the form of constants and variables, and the inputs will have a bounded domain, even though that domain may sometimes be too large for any practical use.

Thus all software modules satisfy, at least in principle, the requirements of a finite state machine.

However, if you consider a typical computational routine, for instance the cube root of a 64-bit floating point number, you can easily conclude that the Finite State approach would be highly inappropriate even though it still satisfies all of the requirements of a FSM.

The difficulty lies of course in the fact that while the set of 64-bit floating point numbers is indeed finite, it would be highly impractical to attempt to enumerate that entire set in order to apply the Finite State paradigm.

On the other hand, we know from past encounters with parsers, command interpreters, or device drivers that the FSM model can be very effective in some situations.

In fact, the class of problems where the Finite State model can be put to good use is much larger than the limited linguistic applications we had to muddle through in a course on Compiler design.

In general, most software problems which have a manageable number of valid inputs and are "control intensive" rather than algorithmic, i.e. they contain a substantial number of control-flow statements (loops, conditional and unconditional branches, etc.) compared to computational statements, can be good candidates for the Finite State approach.

AN EXAMPLE

In order to prove our point and at the same time demonstrate the details of the technique, we will discuss an example of a software application which appears to be rather far removed from FSMs: a patient data management system for a medical office.

You are cautioned that this is an oversimplified academic exercise and may bear no resemblance to any real-world system.

We will concentrate on creating the top-level user interface and leave the rest to the ingenuity of the reader.

Let's assume that the design specs for our program, called PATINFO, require a hierarchical structure for the various display screens and that function keys will be used to select the desired operation and to move from screen to screen.

Also, the bottom line on the monitor will display a menu of the appropriate entry keys, together with a short description of their function.

The menu hierarchy is shown in figure 1. Each selection is labelled with the appropriate function button; the initial screen allows for four possible selections: BILLING, INSURANCE, PATIENT DATA, and EXIT PROGRAM.

Note that the last option is selected via function button F10; we'll consistently reserve that button to back up from a screen at any level to its parent screen.
THE STATE TRANSITION TABLE

Our next item of business is to describe the behavior of our machine by defining all of the stimulus-response relationships.

This is most commonly done by using the State Diagram (see references 1 through 4 below).

While diagrams are intuitively appealing, they tend to get very messy as they get grow and are difficult to modify and maintain; we will opt for the State Transition Table (STT) which can be easily created and maintained on a computer with a text editor or spreadsheet program.

The rows in the STT will correspond to the various inputs to the FSM, while the columns will represent the different states of the machine.

Let's take a second to talk about the significance of machine states.

The concept of state is rather abstract; a state can be roughly described as one of the internal configurations of a FSM.

Since each FSM contains a finite number of memory elements, each capable of storing a discrete number of values, the number of possible states equals the allowable combinations of values for all the memory elements; and a specific machine state could be identified by an ordered list of numbers corresponding to a specific setting of the memory elements.

Consider for instance, a 3-digit display on a digital voltmeter. This type of display consists of 4 windows; disregarding possible decimal points or signs, the leftmost window can display 0 or 1, while the other three can all display the digits 0 through 9.

Thus the sequence 1543, for example, constitutes one of the 2000 possible states for the device.

When we are dealing with a software implementation of a FSM, the concept of state can become even more nebulous; a state corresponds to a specific section of code; and the machine will be in a particular state when the corresponding code segment is being executed.

This is a rather inadequate definition, since we could call the same subroutine from different places in the code representing different states. Thus the same code (i.e. the subroutine) could be executing in different states.

As we shall see shortly, our method requires that we explicitly define a state variable and reassign its value whenever the state changes. Thus the state of our FSM at any moment in time would be the current value of that variable.

Let's get back to creating the State Transition Table: we will first list all valid inputs on separate lines. Hence there will be entries for function buttons F1, F2, F3, F4, and F10; we will also add one more line to account for all invalid inputs by lumping them into a single line, not surprisingly labelled OTHER.

At this point we need to identify all required columns which, as we said, correspond to all of the valid states.

We know of that our FSM has at least an initial state, in which the program displays the top level menu of function keys. We thus proceed to fill every entry in the first state column labelled "0".

Each entry at the intersection of a column and a row consists of two fields: the action field, which occupies most of the box, and the next state field, in the lower right corner of the box.

For example, the action field in box F4_0 (i.e. row F4, column 0) contains "BEEP" and "ILLEGAL ENTRY"; this indicates that we want the program to output an audible beep and display the message "ILLEGAL ENTRY" on the monitor. The next state field in the box contains NC for "No Change" to indicate that the next state is the same as the current one.

This reflects the fact that in the initial menu function key F4 is not valid and we would like the program to clearly indicate that to the user while still remaining in the initial state.

It is customary in more traditional State Transition Tables to enter the numeric value of the next state (in this case "0") even when the state does not change; but in dealing with a software implementation of a FSM there is a definite advantage in explicitly indicating no change as we have done.

In the process of filling all entries in the first we have identified several new states for our FSM.

Each new state will correspond to a column which again will be filled in manner similar to what we have done above.

We continue building the table column by column, until no more new states can be found, and all the existing columns have been filled.

The completed table is shown in figure 2; there are of course other possible realizations for the table which would differ in the order or the labelling of the columns.

Note that while the states are labelled in numerical order, there is no ordering implied; we could have used letters or abbreviations for the labels.

Note also that the next state field in box F10_0 is filled with an asterisk; this is to indicate that once you leave the program the next state is irrelevant; it's analogous to pulling the plug.
THE STATE TRANSITION TABLE

Our next item of business is to describe the behavior of our machine by defining all of the stimulus-response relationships.

This is most commonly done by using the State Diagram (see references 1 through 4 below).

While diagrams are intuitively appealing, they tend to get very messy as they get grow and are difficult to modify and maintain; we will opt for the State Transition Table (STT) which can be easily created and maintained on a computer with a text editor or spreadsheet program.

The rows in the STT will correspond to the various inputs to the FSM, while the columns will represent the different states of the machine.

Let's take a second to talk about the significance of machine states.

The concept of state is rather abstract; a state can be roughly described as one of the internal configurations of a FSM.

Since each FSM contains a finite number of memory elements, each capable of storing a discrete number of values, the number of possible states equals the allowable combinations of values for all the memory elements; and a specific machine state could be identified by an ordered list of numbers corresponding to a specific setting of the memory elements.

Consider for instance, a 3-digit display on a digital voltmeter. This type of display consists of 4 windows; disregarding possible decimal points or signs, the leftmost window can display 0 or 1, while the other three can all display the digits 0 through 9.

Thus the sequence 1543, for example, constitutes one of the 2000 possible states for the device.

When we are dealing with a software implementation of a FSM, the concept of state can become even more nebulous; a state corresponds to a specific section of code; and the machine will be in a particular state when the corresponding code segment is being executed.

This is a rather inadequate definition, since we could call the same subroutine from different places in the code representing different states. Thus the same code (i.e. the subroutine) could be executing in different states.

As we shall see shortly, our method requires that we explicitly define a state variable and reassign its value whenever the state changes. Thus the state of our FSM at any moment in time would be the current value of that variable.

Let's get back to creating the State Transition Table: we will first list all valid inputs on separate lines. Hence there will be entries for function buttons F1, F2, F3, F4, and F10; we will also add one more line to account for all invalid inputs by lumping them into a single line, not surprisingly labelled OTHER.

At this point we need to identify all required columns which, as we said, correspond to all of the valid states.

We know of that our FSM has at least an initial state, in which the program displays the top level menu of function keys. We thus proceed to fill every entry in the first state column labelled "0".

Each entry at the intersection of a column and a row consists of two fields: the action field, which occupies most of the box, and the next state field, in the lower right corner of the box.

For example, the action field in box F4_0 (i.e. row F4, column 0) contains "BEEP" and "ILLEGAL ENTRY"; this indicates that we want the program to output an audible beep and display the message "ILLEGAL ENTRY" on the monitor. The next state field in the box contains NC for "No Change" to indicate that the next state is the same as the current one.

This reflects the fact that in the initial menu function key F4 is not valid and we would like the program to clearly indicate that to the user while still remaining in the initial state.

It is customary in more traditional State Transition Tables to enter the numeric value of the next state (in this case "0") even when the state does not change; but in dealing with a software implementation of a FSM there is a definite advantage in explicitly indicating no change as we have done.

In the process of filling all entries in the first we have identified several new states for our FSM.

Each new state will correspond to a column which again will be filled in manner similar to what we have done above.

We continue building the table column by column, until no more new states can be found, and all the existing columns have been filled.

The completed table is shown in figure 2; there are of course other possible realizations for the table which would differ in the order or the labelling of the columns.

Note that while the states are labelled in numerical order, there is no ordering implied; we could have used letters or abbreviations for the labels.

Note also that the next state field in box F10_0 is filled with an asterisk; this is to indicate that once you leave the program the next state is irrelevant; It's analogous to pulling the plug.
on a hardware FSM.

An optimal table would contain the minimum number of required states. In a problem as simple as ours it is easy to come up with a minimal solution. But optimizing the State Transition table for a complex FSM can be rather complex; the subject is discussed in detail in references 1 and 2.

GENERATING CODE

We will use the table as a template to generate the required code.

Our initial instincts tell us that a two dimensional 6-by-6 table should correspond to a code arrangement consisting of an outer 6-way decision tree or CASE structure for the 6 possible states; and at each branch there would be a second level 6-way decision tree whose branches would correspond to the six possible inputs.

Or if you prefer, the outer structure could depend on the inputs, while the inner CASE structures would represent the states.

We will however take a different course of action by using a single level CASE structure, each of whose branches represents a selection of one state and one input.

While this stratagem may not always reduce the total number of cases or the length of the code, "flattening" a two dimensional problem into a one dimensional structure results in software which is less prone to coding faults and is easier to maintain.

Let's examine the pseudo-code for program PATINFO in figure 3: we have declared two single-byte variables, INPUT and STATE; there is also a 2-byte variable called INPUT_STATE.

The program begins by waiting for a keyboard input. That input is then concatenated with the current state and stored in the variable INPUT_STATE.

The CASE structure shows that we covered the State Transition Table in figure 2 by traversing it column by column from left to right; but any ordering of the boxes in the table would work equally well.

This makes the code easy to maintain, since adding cases created by new inputs or states could be done by inserting the appropriate code anywhere in the CASE structure.

Note that a "NC" in the next state field tells us that we need not reset the STATE variable and thus can dispense with unnecessary assignment statements.

Also, all illegal situations (F04_0, F04_10, etc.) have been lumped into the "default" section of the CASE structure, since they are all handled in the same way.

The efficiency of the code can be further improved by making sure that variables INPUT and STATE occupied consecutive memory locations and that variable INPUT_STATE referred to the same addresses; this can be accomplished in FORTRAN with an EQUIVALENCE statement or in C with a union.

We could then remove statement "INPUT_STATE = concat(INPUT,STATE)" thus reducing both the code and the execution time.

CONCLUSIONS

Hopefully our simple exercise has convinced the reader of the value of using the FSM model in solving some software problems. As we have seen, the fundamental step in the technique was to create a State Transition Table.

The table can be effective at the lowest design phase just before code is to be created, because it constitutes both a template for the code and also becomes a maintenance document after completion of the project.

At the initial stages of design for a large software project, developing a State Transition Table on a blackboard could allow several individuals to cooperate effectively in the design.

Because of the simple one-level CASE structure, the resulting software is easy to debug and maintain.

Code testing can be facilitated by following variable INPUT_STATE; by tracing the changes in its values via a few temporary (or conditionally compiled) "print" statements we can gain significant insight into the proper operation of our software.

REFERENCES

on a hardware FSM.

An optimal table would contain the minimum number of required states. In a problem as simple as ours it is easy to come up with a minimal solution. But optimizing the State Transition table for a complex FSM can be rather complex; the subject is discussed in detail in references 1 and 2.

GENERATING CODE

We will use the table as a template to generate the required code.

Our initial instincts tell us that a two dimensional 6-by-6 table should correspond to a code arrangement consisting of an outer 6-way decision tree or CASE structure for the 6 possible states; and at each branch there would be a second level 6-way decision tree whose branches would correspond to the six possible inputs.

Or if you prefer, the outer structure could depend on the inputs, while the inner CASE structures would represent the states.

We will however take a different course of action by using a single level CASE structure, each of whose branches represents a selection of one state and one input.

While this stratagem may not always reduce the total number of cases or the length of the code, "flattening" a two dimensional problem into a one dimensional structure results in software which is less prone to coding faults and is easier to maintain.

Let’s examine the pseudo-code for program PATINFO in figure 3: we have declared two single-byte variables, INPUT and STATE; there is also a 2-byte variable called INPUT_STATE.

The program begins by waiting for a keyboard input. That input is then concatenated with the current state and stored in the variable INPUT_STATE.

The CASE structure shows that we covered the State Transition Table in figure 2 by traversing it column by column from left to right; but any ordering of the boxes in the table would work equally well.

This makes the code easy to maintain, since adding cases created by new inputs or states could be done by inserting the appropriate code anywhere in the CASE structure.

Note that a "NC" in the next state field tells us that we need not reset the STATE variable and thus can dispense with unnecessary assignment statements.

Also, all illegal situations (F04_0, F04_10, etc.) have been lumped into the "default" section of the CASE structure, since they are all handled in the same way.

The efficiency of the code can be further improved by making sure that variables INPUT and STATE occupied consecutive memory locations and that variable INPUT_STATE referred to the same addresses; this can be accomplished in FORTRAN with an EQUIVALENCE statement or in C with a union.

We could then remove statement "INPUT_STATE = concat(INPUT,STATE)" thus reducing both the code and the execution time.

CONCLUSIONS

Hopefully our simple exercise has convinced the reader of the value of using the FSM model in solving some software problems. As we have seen, the fundamental step in the technique was to create a State Transition Table.

The table can be effective at the lowest design phase just before code is to be created, because it constitutes both a template for the code and also becomes a maintenance document after completion of the project.

At the initial stages of design for a large software project, developing a State Transition Table on a blackboard could allow several individuals to cooperate effectively in the design.

Because of the simple one-level CASE structure, the resulting software is easy to debug and maintain.

Code testing can be facilitated by following variable INPUT_STATE; by tracing the changes in its values via a few temporary (or conditionally compiled) "print" statements we can gain significant insight into the proper operation of our software.

REFERENCES


FIGURES 1-3

PATINFO

F01: BILLING
- F01: GENERATE INVOICES
- F01: ONE PATIENT
- F02: MONTHLY ACTIVITY
- F04: EXIT

F02: LIST AGED ACCOUNTS
- F01: 60 DAYS
- F02: 90 DAYS
- F03: DELINQUENT
- F10: EXIT

F02: INSURANCE
- F01: NEW INSURANCE FORM
- F02: UPDATE INSURANCE INFO
- F10: EXIT

F03: PATIENT DATA
- F01: NEW PATIENT
- F02: UPDATE PATIENT INFO
- F03: ENTER VISIT
- F04: GENERATE RECALL REMINDER
- F10: EXIT

F10: EXIT PROGRAM/RETURN TO OPERATING SYSTEM

<table>
<thead>
<tr>
<th>INPUTS</th>
<th>0</th>
<th>10</th>
<th>11</th>
<th>12</th>
<th>20</th>
<th>30</th>
</tr>
</thead>
<tbody>
<tr>
<td>F01</td>
<td>MENU_BIL</td>
<td>MENU_INV</td>
<td>ONE_PAT</td>
<td>LATE60</td>
<td>NEW_INS</td>
<td>NEW_PAT</td>
</tr>
<tr>
<td></td>
<td>10</td>
<td>11</td>
<td>NC</td>
<td>NC</td>
<td>NC</td>
<td>NC</td>
</tr>
<tr>
<td>F02</td>
<td>MENU_INS</td>
<td>MENU_AGD</td>
<td>MONTHLY</td>
<td>LATE90</td>
<td>UPD_INS</td>
<td>UPD_PAT</td>
</tr>
<tr>
<td></td>
<td>20</td>
<td>12</td>
<td>NC</td>
<td>NC</td>
<td>NC</td>
<td>NC</td>
</tr>
<tr>
<td>F03</td>
<td>MENU_PAT</td>
<td>BEEP_ILNTRY</td>
<td>BEEP_ILNTRY</td>
<td>DELINQ</td>
<td>BEEP_ILNTRY</td>
<td>VISIT</td>
</tr>
<tr>
<td></td>
<td>20</td>
<td>12</td>
<td>NC</td>
<td>NC</td>
<td>NC</td>
<td>NC</td>
</tr>
<tr>
<td>F04</td>
<td>BEEP_ILNTRY</td>
<td>BEEP_ILNTRY</td>
<td>BEEP_ILNTRY</td>
<td>BEEP_ILNTRY</td>
<td>BEEP_ILNTRY</td>
<td>RECALL</td>
</tr>
<tr>
<td></td>
<td>20</td>
<td>12</td>
<td>NC</td>
<td>NC</td>
<td>NC</td>
<td>NC</td>
</tr>
<tr>
<td>F10</td>
<td>BACK_TO_OS</td>
<td>MENU_STA</td>
<td>MENU_BIL</td>
<td>MENU_BIL</td>
<td>MENU_STA</td>
<td>MENU_STA</td>
</tr>
<tr>
<td></td>
<td>0</td>
<td>10</td>
<td>10</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>OTHER</td>
<td>BEEP_ILNTRY</td>
<td>BEEP_ILNTRY</td>
<td>BEEP_ILNTRY</td>
<td>BEEP_ILNTRY</td>
<td>BEEP_ILNTRY</td>
<td>BEEP_ILNTRY</td>
</tr>
<tr>
<td></td>
<td>NC</td>
<td>NC</td>
<td>NC</td>
<td>NC</td>
<td>NC</td>
<td>NC</td>
</tr>
</tbody>
</table>

Figure 1

Figure 2
FIGURES 1-3

PATINFO

F01: BILLING
- F01: GENERATE INVOICES
- F01: ONE PATIENT
- F02: MONTHLY ACTIVITY
- F04: EXIT

F02: LIST AGED ACCOUNTS
- F01: 60 DAYS
- F02: 90 DAYS
- F03: DELINQUENT
- F10: EXIT

F02: INSURANCE
- F01: NEW INSURANCE FORM
- F02: UPDATE INSURANCE INFO
- F10: EXIT

F03: PATIENT DATA
- F01: NEW PATIENT
- F02: UPDATE PATIENT INFO
- F03: ENTER VISIT
- F04: GENERATE RECALL REMINDER
- F10: EXIT

F04: EXIT PROGRAM/RETURN TO OPERATING SYSTEM

Figure 1

<table>
<thead>
<tr>
<th>INPUTS</th>
<th>0</th>
<th>10</th>
<th>11</th>
<th>12</th>
<th>20</th>
<th>30</th>
</tr>
</thead>
<tbody>
<tr>
<td>F01</td>
<td>MENU_BIL</td>
<td>MENU_INV</td>
<td>ONE_PAT</td>
<td>LATE60</td>
<td>NEW_INS</td>
<td>NEW_PAT</td>
</tr>
<tr>
<td>F02</td>
<td>MENU_INS</td>
<td>MENU_AGD</td>
<td>MONTHLY</td>
<td>LATE90</td>
<td>UPD_INS</td>
<td>UPD_PAT</td>
</tr>
<tr>
<td>F03</td>
<td>MENU_PAT</td>
<td>BEEP</td>
<td>BEEP</td>
<td>DELINQ</td>
<td>BEEP</td>
<td>VISIT</td>
</tr>
<tr>
<td>F04</td>
<td>BEEP</td>
<td>BEEP</td>
<td>BEEP</td>
<td>BEEP</td>
<td>BEEP</td>
<td>RECALL</td>
</tr>
<tr>
<td>F10</td>
<td>BACK TO OS</td>
<td>*</td>
<td>MENU_BIL</td>
<td>MENU_BIL</td>
<td>MENU_STA</td>
<td>MENU_STA</td>
</tr>
<tr>
<td>OTHER</td>
<td>BEEP</td>
<td>BEEP</td>
<td>BEEP</td>
<td>BEEP</td>
<td>BEEP</td>
<td>BEEP</td>
</tr>
</tbody>
</table>

Figure 2
start PATINFO
byte INPUT = " "
byte STATE = 0
2-byte integer INPUT_STATE

   call menu(START)
   do forever
       wait here till input received
       INPUT_STATE = concat(INPUT, STATE)
       switch(INPUT_STATE)

       case(F01_0)
           STATE = 10
           call menu(BIL)
       case(F02_0)
           STATE = 20
           call menu(INS)
       case(F03_0)
           STATE = 30
           call menu(PAT)
       case(F10_0)
           call exit()
       case(F01_10)
           STATE = 11
           call menu(INV)
       case(F02_10)
           STATE = 12
           call menu(AGD)
       case(F10_10)
           STATE = 0
           call menu(START)
       case(F01_11)
           call one_pat()
       case(F02_11)
           call monthly()
       case(F10_11)
           STATE = 10
           call menu(BIL)
       case(F01_12)
           call late(60)
       case(F02_12)
           call late(90)
       case(F03_12)
           call delinquency
       case(F10_12)
           STATE = 10

call menu(BIL)
   case(F01_20)
       call ins(NEW)
   case(F02_20)
       call ins(UPDATE)
   case(F10_20)
       STATE = 0
       call menu(START)
   case(F01_30)
       call patient(NEW)
   case(F02_30)
       call patient(UPDATE)
   case(F03_30)
       call visit()
   case(F04_30)
       call remind()
   case(F10_30)
       STATE = 0
       call menu(START)
   default
       call exit()
       call beep()
       display("ILLEGAL ENTRY")
   end switch
   end do
end PATINFO

Figure 3
start PATINFO
byte INPUT = " "
byte STATE = 0
2-byte integer INPUT_STATE

call menu(START)
do forever
    wait here until input received
    INPUT_STATE = concat(INPUT, STATE)
switch(INPUT_STATE)
    case(F01_0)
        STATE = 10
        call menu(BIL)
    case(F02_0)
        STATE = 20
        call menu(INS)
    case(F03_0)
        STATE = 30
        call menu(PAT)
    case(F04_0)
        call exit()
    case(F01_10)
        STATE = 11
        call menu(INV)
    case(F02_10)
        STATE = 12
        call menu(AGD)
    case(F03_10)
        STATE = 0
        call menu(START)
    case(F01_11)
        call one_pat()
    case(F02_11)
        call monthly()
    case(F03_11)
        STATE = 10
        call menu(BIL)
    case(F01_12)
        call late(60)
    case(F02_12)
        call late(90)
    case(F03_12)
        call delinquency()
    case(F00_12)
        STATE = 10
    default
        call exit()
        call beep()
        display("ILLEGAL ENTRY")
end switch
end do
end PATINFO

call menu(BIL)
case(F01_20)
    call ins(NEW)
case(F02_20)
    call ins(UPDATE)
case(F03_20)
    call menu(START)
case(F01_30)
    call patient(NEW)
case(F02_30)
    call patient(UPDATE)
case(F03_30)
    call visit()
case(F04_30)
    call remind()
case(F00_30)
    STATE = 0
    call menu(START)
default
    call exit()
call beep()
end do

case(F01_11)
    call one_pat()
case(F02_11)
    call monthly()
case(F03_11)
    STATE = 10
    call menu(BIL)
case(F01_12)
    call late(60)
case(F02_12)
    call late(90)
case(F03_12)
    call delinquency()
case(F00_12)
    STATE = 10

call menu(START)
case(F01_20)
    call ins(NEW)
case(F02_20)
    call ins(UPDATE)
case(F03_20)
    call menu(START)
case(F01_30)
    call patient(NEW)
case(F02_30)
    call patient(UPDATE)
case(F03_30)
    call visit()
case(F04_30)
    call remind()
case(F00_30)
    STATE = 0
    call menu(START)
default
    call exit()
call beep()
end do

case(F01_11)
    call one_pat()
case(F02_11)
    call monthly()
case(F03_11)
    STATE = 10
    call menu(BIL)
case(F01_12)
    call late(60)
case(F02_12)
    call late(90)
case(F03_12)
    call delinquency()
case(F00_12)
    STATE = 10

call menu(START)
case(F01_20)
    call ins(NEW)
case(F02_20)
    call ins(UPDATE)
case(F03_20)
    call menu(START)
case(F01_30)
    call patient(NEW)
case(F02_30)
    call patient(UPDATE)
case(F03_30)
    call visit()
case(F04_30)
    call remind()
case(F00_30)
    STATE = 0
    call menu(START)
default
    call exit()
call beep()
end do

case(F01_11)
    call one_pat()
case(F02_11)
    call monthly()
case(F03_11)
    STATE = 10
    call menu(BIL)
case(F01_12)
    call late(60)
case(F02_12)
    call late(90)
case(F03_12)
    call delinquency()
case(F00_12)
    STATE = 10

call menu(START)
case(F01_20)
    call ins(NEW)
case(F02_20)
    call ins(UPDATE)
case(F03_20)
    call menu(START)
case(F01_30)
    call patient(NEW)
case(F02_30)
    call patient(UPDATE)
case(F03_30)
    call visit()
case(F04_30)
    call remind()
case(F00_30)
    STATE = 0
    call menu(START)
default
    call exit()
call beep()
end do

Figure 3