54 Data Flow

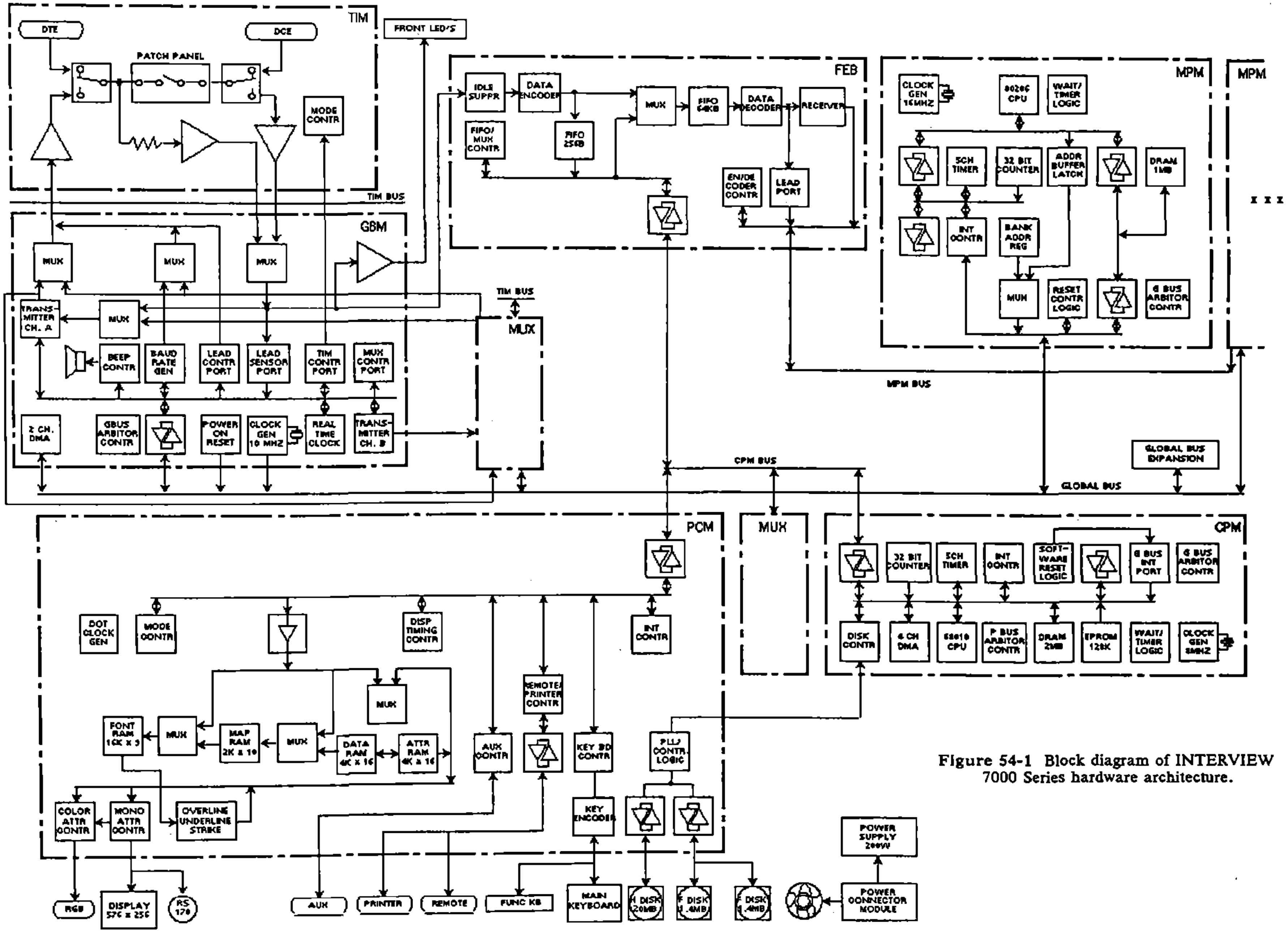


Figure 54-1 Block diagram of INTERVIEW 7000 Series hardware architecture.

54 Data Flow

Figure 54-1 is a block diagram showing the components on each of the six types of logic board in the INTERVIEW 7000 Series. The components on the TIM (Test Interface Module) also are shown. Figure 54-2 indicates the flow of data among the various functional components of the unit.

54.1 Two Types of CPU

The brain of the INTERVIEW is the Motorola 68010 processor on the CPM (Central Processing Module). See Figure 54-1. The 68010 processor controls operations in the unit not directly under control of the user program. 68010 operations include fetching power-up software and initialization routines from the EPROM, controlling disk I/O, and maintaining setup and statistics screens. The operating system in the 68010 is pSOS.

An Intel 80286 processor controls the operation of the MPM (Multiple Processor Module). The MPM does all higher level processing of receive data. The board also generates the transmit data to be sent out in emulate mode. The 80286 uses a basic, multitasking real-time executive operating system.

An INTERVIEW 7000 and 7200 *TURBO* has one MPM with its own 80286 CPU. The INTERVIEW 7500 and 7700 *TURBO* always have three MPMs, each with its own 80286 CPU.

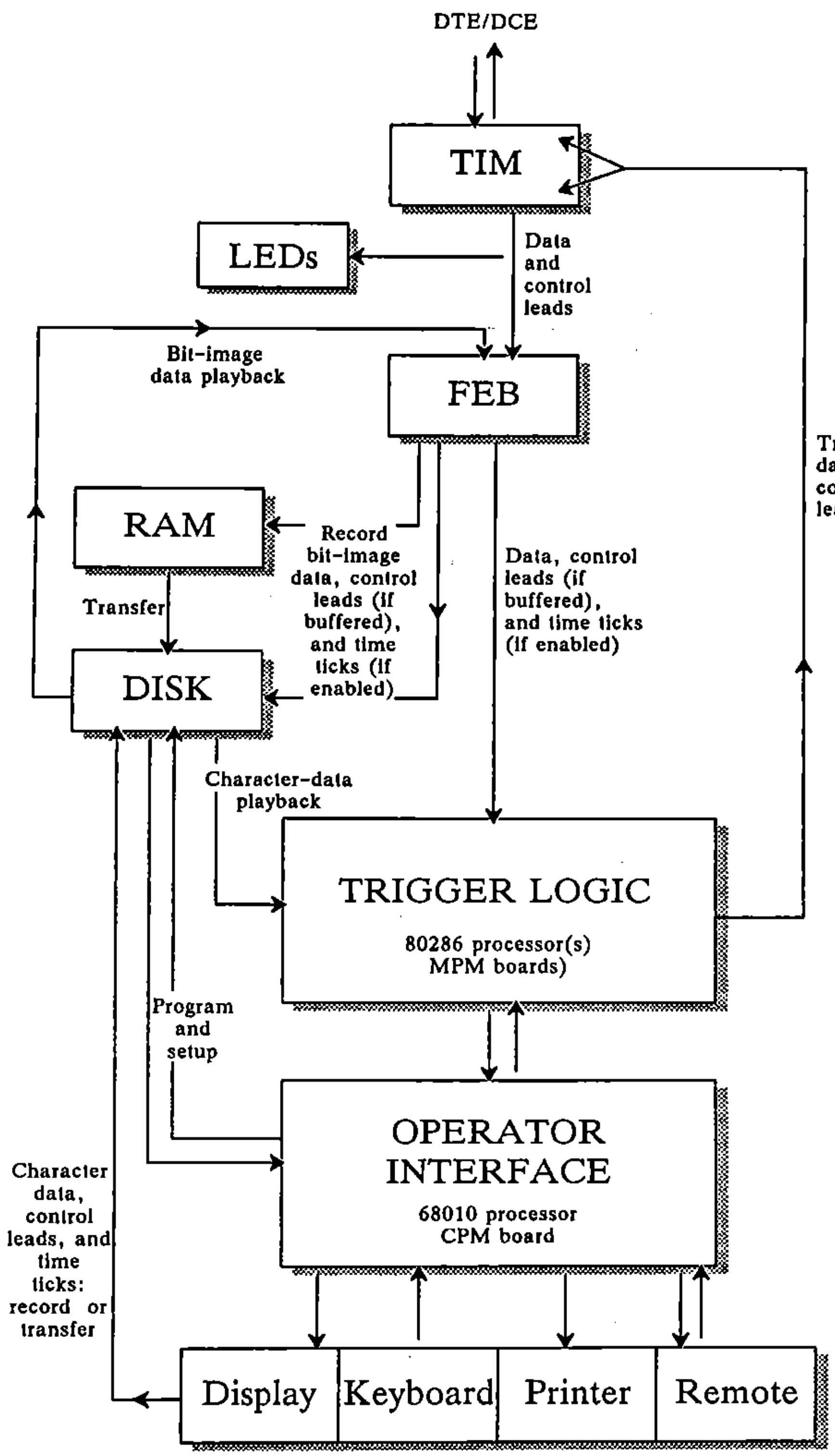

DTE/DCE
TIM
LEDs
Data and control leads
Bit-image data playback
FEB
Transmitted data and control leads
RAM
Record bit-image data, control leads (if buffered), and time ticks (if enabled)
Data, control leads (if buffered), and time ticks (if enabled)
Transfer
DISK
Character-data playback
TRIGGER LOGIC
80286 processor(s) MPM boards)
Program and setup
OPERATOR INTERFACE
68010 processor CPM board
Character data, control leads, and time ticks: record or transfer
Display
Keyboard
Printer
Remote

Figure 54-2 INTERVIEW 7000 Series functional diagram.

The 80286 operates on software located in the DRAM on the MPM. See Figure 54-1. This software is the user program—setups, trigger menus, protocol spreadsheet, and protocol state machines (layer packages)—translated and compiled by the CPM and loaded into the MPM. The program will tell the MPM how to process the data, what trigger conditions to look for in the data stream, etc.

The CPM polls the MPM continuously to see if data is available to be output to the printer or the plasma display. This data includes character data, trace data, prompts, and values to be posted to the statistics screens.

While the CPM accesses the MPM on a regular basis, there is no access in the reverse direction. That is, *the user program running on the MPM has no direct access to the CPM*. The user cannot write to one of the menu screens, for example.

54.2 Front-End Buffer

Note in Figure 54-2 that the front-end buffer (FEB) lies squarely between the line interface and (1) the recording medium and (2) the program logic on the MPM. This means that control leads may or may not be recorded and may or may not be seen by the trigger-menu and spreadsheet conditions—depending on the FEB setup (see Section 9).

Once control leads and time ticks (that is, the original timing values) are recorded alongside character data, they are locked in. Since the FEB is not on the playback path for character data, FEB selections do not apply.

Bit-image data, however, does pass through the FEB during playback. Except for the **Idle Suppress** field, FEB selections apply. This means that control leads and time ticks, if recorded with the data, *must* be enabled in order for the program logic to detect them.

Not only characters but also leads and time ticks, if enabled in the FEB setup, are captured automatically in the display buffer (that is, the screen buffer or character RAM).

Data, time ticks, and control leads are encoded in a special storage format by a data-encoder chip on the FEB board. See Figure 54-1. The encoded data is buffered to be sent to the PCM (Peripheral Control Module) for recording and to the MPM for processing.

The encoding process is driven by clock pulses on the line interface. This means that in the absence of external clock (or, if the INTERVIEW is emulating DCE, in the absence of internal clock), neither line data, time ticks nor EIA leads will be recorded or presented to the receivers and to the program logic.

55 Program Main

Softkey–selectable programming "tokens" entered by the user on the Protocol Spreadsheet are translated automatically into C during the initial compiler phases after $\boxed{\text{RUN}}$ is pressed. Trigger Menu setups also are translated into C. When the translation is complete, the compiler takes over and converts the C code into object code. The C variables and routines used by the translator are documented throughout this volume.

Briefly, the translator makes the following conversions: it turns TESTs into tasks; STATE names into labels; STATEs into *waitfor* clauses; CONDITIONS into *waitfor* expressions that include event variables; and ACTIONS into statements and routines, also inside of *waitfor* clauses.

Then the translator creates a program *main* function that calls every task in the program.

55.1 Translating a Simple Test into C

Suppose that the following simple program, intended to sound the INTERVIEW's alarm at 1 P.M., has been entered on the Protocol Spreadsheet.

```
STATE: sample1
  CONDITIONS: TIME 1300
  ACTIONS: ALARM
```

When the user presses $\boxed{\text{RUN}}$, *roughly* the following C coding (with some extraneous code removed for clarity) is generated and then compiled:

```
extern fast_event fevar_time_of_day;
extern volatile unsigned short crnt_time_of_day;
task
{
  main ()
  {
    state_sample1:
    waitfor
    {
      fevar_time_of_day && (crnt_time_of_day == 1300):
      {
        sound_alarm ();
      }
    }
  }
} dtest_0;
main ()
{
  dtest_0 ();
}
```

Note that the translator has assigned *state_sample1* to a default TEST named *dtest_0*. It converted the TEST into a task and placed *state_sample1* inside of the task. Then it created a program *main* function and used the program *main* to call every test/task in the program. The tasks appear in the task list in the same order in which they appear in the spreadsheet program. In this instance there was only one task to call.

If you try to enter the program above on the spreadsheet entirely in C, in the first place you will have to surround it with a pair of curly braces. Then it will not compile. The translator does not look inside of curly braces (except to expand constants). It simply lifts up the braced C regions and places them intact into its translation of the softkey portion of the program, before adding a program *main*—even when, as in this instance, a program *main* already is included in a C region. The two *main* functions conflict here, and the compiler issues the error message, "*Error 109: Function main redefined.*" .

If we were to remove the *main* function from our C version, the program would compile but it still would not *work*. Here's why. When the translator looks at a program made up entirely of C code, it doesn't see anything. So it creates a program *main* with a task-list that is empty. The task that is declared in the program above (*dtest_0*) is never called.

The rule, then, is that a Protocol Spreadsheet program containing tasks written in C must always have at least one softkey STATE (with its implied task) that calls all the tasks.

55.2 A Minimum of One Softkey *State*

Here is a Protocol Spreadsheet test that works and yet has the minimum number of softkey tokens—one. Note that we have given the task *dtest_0* a new name, since the translator will declare the task-name *dtest_0* as the default test for our new softkey state, *task_list*.

```
{
   extern fast_event fevar_time_of_day;
   extern volatile unsigned short crnt_time_of_day;
   task
   {
      main ()
      {
         state_sample1:
         waitfor
         {
            fevar_time_of_day && (crnt_time_of_day == 1300):
            {
               sound_alarm ();
            }
         }
      }
   } c_test;
}
   STATE: task_list
         {
            c_test ();
         }
```

And here is the program as it is actually compiled. Note that the translator has added a program main that calls *dtest_0* (which in turn calls *c_test*).

```
extern fast_event fevar_time_of_day;
extern volatile unsigned short crnt_time_of_day;
task
{
  main ()
  {
    state_sample1:
    waitfor
    {
      fevar_time_of_day && (crnt_time_of_day == 1300):
      {
        sound_alarm ();
      }
    }
  }
} c_test;
task
{
  main ()
  {
    state_task_list:
    {
      c_test ();
      waitfor                   /* This empty waitfor is automatically generated in any state
  {                                that does not contain a waitfor. */
      }
    }
  }
} dtest_0;
main ()
{
  dtest_0 ();
}
```

55.3 Writing the Test Entirely in C

The INTERVIEW is equipped with tools—namely, the *#pragma hook 0* preprocessor directive and linkable-object (LOBJ) files—that make it possible to write a version of the test completely in C.

NOTE: For more information on *#pragma hook* directives, see Section 59.4. Refer also to Section 14.3(P) on linkable-object files.

Write the following C code to an ASCII file (*hook_ctest.s*) using the Protocol Spreadsheet editor's WRITE/U command. Then delete the code from the spreadsheet. Go to the File Maintenance screen and and create a linkable-object file (*hook_ctest.o*) using the Compile command.

```
#pragma hook 0 "c_test();"
extern fast_event fevar_time_of_day;
extern volatile unsigned short crnt_time_of_day;
task
  {
    main()
     {
       state_sample1:
       waitfor
         {
          fevar_time_of_day && (crnt_time_of_day == 1300):
           {
             sound_alarm();
           }
         }
     }
  } c_test_task;
c_test()
  {
    c_test_task();
  }
```

Notice that the "hook" is a call to the routine *c_test*. This routine's only purpose is to start the task, *c_test_task*. A task name is always local to a linkable-object file and never *directly* copied from it. If you try to call the task directly in the *#pragma hook 0* directive, therefore, the spreadsheet program (shown below) will not compile. Since the task name is local to the file, the following error message will be displayed: *"Error 140: Unresolved reference c_test_task."* The rule for including tasks in a linkable-object file, then, is to let the *#pragma hook 0* directive call a routine which starts the task(s).

> NOTE: Since task names are local to a file, the definition of *c_test_task* also cannot be located in a referenced LOBJ file different from the one in which it is called.

The Protocol Spreadsheet program required to execute the test consists of a single line:

```
OBJECT: "hook_ctest.o"
```

When translated, the program looks like this:

```
#pragma object "hook_ctest.o"
main()
  {
    c_test();
  }
```

Notice that the routine *c_test* is located within the top-level program *main*. The hook text from a *#pragma hook 0* directive is always put at the end of *main*'s task list. At this point, since *c_test* has not been previously declared, it is assumed to be an *extern* function (not a task) that returns an *int*. The linkable-object file(s) referenced in the spreadsheet program will be searched for the routine's definition.

56 Regions in Spreadsheet

C language can be embedded in a Protocol Spreadsheet program at several access points. A C region can be opened at the top of the program, or in an OBJECT, IL_BUFFERS, CONSTANTS, LAYER, TEST, STATE, CONDITIONS, or ACTIONS block.

At these points, simply begin the C region with an opening curly brace. Make your entry and terminate it with a closing curly brace.

The remainder of this section describes C code blocks related to the spreadsheet components, from largest to smallest.

56.1 Layer and Test

The *main* function of a *task* is the highest level function that may be programmed by the user of the INTERVIEW 7000 Series. The keyword *task* in a C region corresponds to the TEST: softkey token on the Protocol Spreadsheet. Typing TEST: **keyboard_alarm** on the spreadsheet is the equivalent of the following C coding:

```
task
{
   #pragma layer 1
   main()
   {
      /* declarations, state-labels, and statements go here */
   }
}
layer_1_test_keyboard_alarm;
```

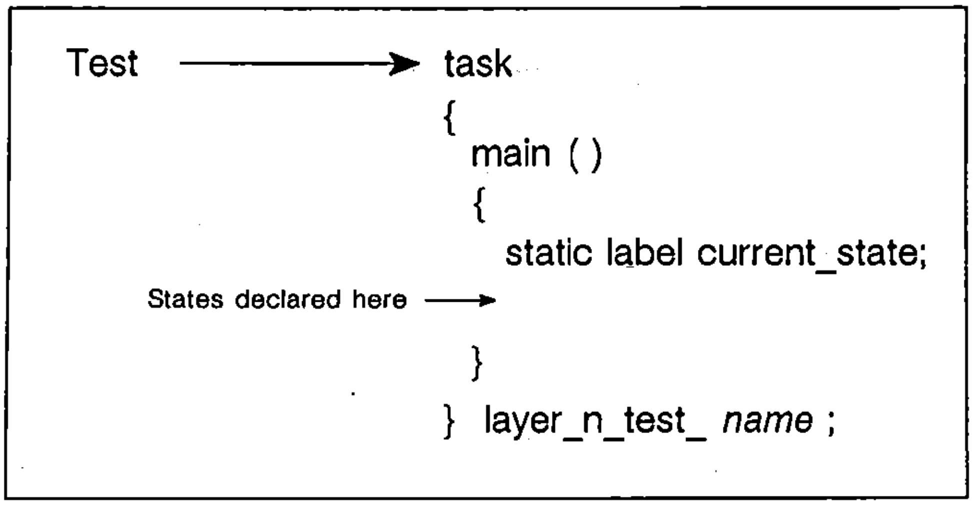
The INTERVIEW is multitasking, so more than one task/test may be defined. All tasks/tests run concurrently if they are included in the task list created by the translator when it generates the program *main* function. See Section 55, Program Main, for an explanation of how this automatic program *main* is created.

Layers have no existence in C independent of the tasks that they contain. When a user enters the LAYER: token on the spreadsheet followed by a layer number, the C translator prefixes that number to the name of each task that follows. Note in the example above that the test name **keyboard_alarm** was given a *layer_1_test* prefix.

The C translator also issued the preprocessor directive *#pragma layer 1*. The compiler uses this layer declaration to distribute tasks efficiently among 80286 processors. This pragma is an optimizing feature and is not strictly required in the body of the task.

The C translator does nothing else with the layer number other than convert it into a prefix to the task name and construct the *#pragma* directive.

The layer number does, of course, determine many of the branching *softkey* selections that will be available to the user who is not programming in C. The C programmer will find that none of the variables or routines mentioned in this manual is specific to a particular layer. A variable or routine that is supplied, for example, by the X.25 Layer 3 personality package (at the time that the package is loaded in via the Layer Setup screen) will still be available inside of a task that nominally belongs to Layer 1 or Layer 2.



```
task
{
  main ( )
  {
    static label current_state;

  }
} layer_n_test_ name ;
```

Figure 56-1 C equivalent of a spreadsheet test.

56.2 State, Enter State, and Next State

A STATE on the Protocol Spreadsheet is a label in C, used as a target of a *goto* statement. Typing STATE: **alarm_on** on the spreadsheet is the equivalent of this C coding, placed inside of the braces that follow the task *main*:

```
static label current_state;
state_alarm_on:
current_state = state_alarm_on_loop;
{
   /* statements go here */
   goto (current_state);
   state_alarm_on_loop:
   waitfor
   {
      /* condition clauses go here, each comprised of expression, colon(:), and statements */
   }
   goto (current_state);
}
```

Note that the C translator has taken STATE: **alarm_on** on the Protocol Spreadsheet and produced two state labels, *state_alarm_on* and *state_alarm_on_loop*. The first state label is followed by statements that will be executed immediately upon entering the state. The "loop"-state label always introduces a *waitfor* construction. Both states end in a statement to *goto (current_state)*.

The translator's version of a state includes overhead to cover all cases, including special cases. The loop state is not strictly required, and a streamlined version of the basic state coding that eliminates the extra state will work in most instances:

```
static label current_state;
state_alarm_on:
{
   /* declarations and statements go here */
   waitfor
   {
      /* condition clauses go here, each comprised of expression, colon(:), and statement(s) */
   }
   goto (current_state);
}
```

Note these points about states created entirely by the programmer:

- A *goto* statement cannot be used inside of a *waitfor* construction.
- You must use a *break* statement to exit the *waitfor* construction.
- You may dispense with the *current_state* variable and *goto* a state label, in which case the opening and closing parens may be omitted.

(A) Declaring States

The state name followed by the colon (:) is itself a label declaration and does not require an additional declaration.

(B) Enter State

The C translator puts a *waitfor* construction into every "loop" state. If you want a statement to be executed immediately without waiting for an event, you may place that statement in the nonloop state, outside of the *waitfor* statement. The following is an example of a state in which the *sound_alarm* routine is executed immediately.

```
static label current_state;
state_alarm_on:
current_state = state_alarm_on_loop;
{
   sound_alarm();
   goto (current_state);
   state_alarm_on_loop:
   waitfor
   {
   }
   goto (current_state);
}
```

The example above is the equivalent of this spreadsheet entry:

```
STATE: alarm_on
   CONDITIONS: ENTER_STATE
   ACTIONS: ALARM
```

A hybrid version also may be created:

```
STATE: alarm_on
{
   sound_alarm();
}
```

The *sound_alarm* function is executed immediately, since the translator places it above the *waitfor*. When you enter a CONDITIONS: block on the spreadsheet, you move inside a *waitfor*—unless you place your C region immediately following an ENTER_STATE.

An ENTER_STATE condition may cause the translator to generate an *if* statement in the nonloop state (above the *waitfor* state). Here is a spreadsheet example:

```
STATE: alarm_on
  CONDITIONS: ENTER_STATE
    COUNTER anyname EQ 3
  ACTIONS: ALARM
```

This is the C version:

```
static label current_state;
state_alarm_on:
current_state = state_alarm_on_loop;
{
  if (counter_anyname.current  == 3) sound_alarm();
  goto (current_state);
  state_alarm_on_loop:
  waitfor
  {
  }
  goto (current_state);
}
```

And here is a hybrid version:

```
STATE: alarm_on
{
  if (counter_anyname.current  == 3) sound_alarm();
}
```

(C) Next State

The C translator supplies the statement "*goto (current_state)*" at the bottom of every state that it codes. If *current_state* has been redefined and if the program reaches the bottom of the state, the *goto* statement will redirect the program toward a new state label. That is how the program is redirected into *state_alarm_on_loop* in this translator's version of STATE: alarm_on:

```
static label current_state;
state_alarm_on:
current_state = state_alarm_on_loop;
{
  goto (current_state);
  state_alarm_on_loop:
  waitfor
  {
  }
  goto (current_state);
}
```

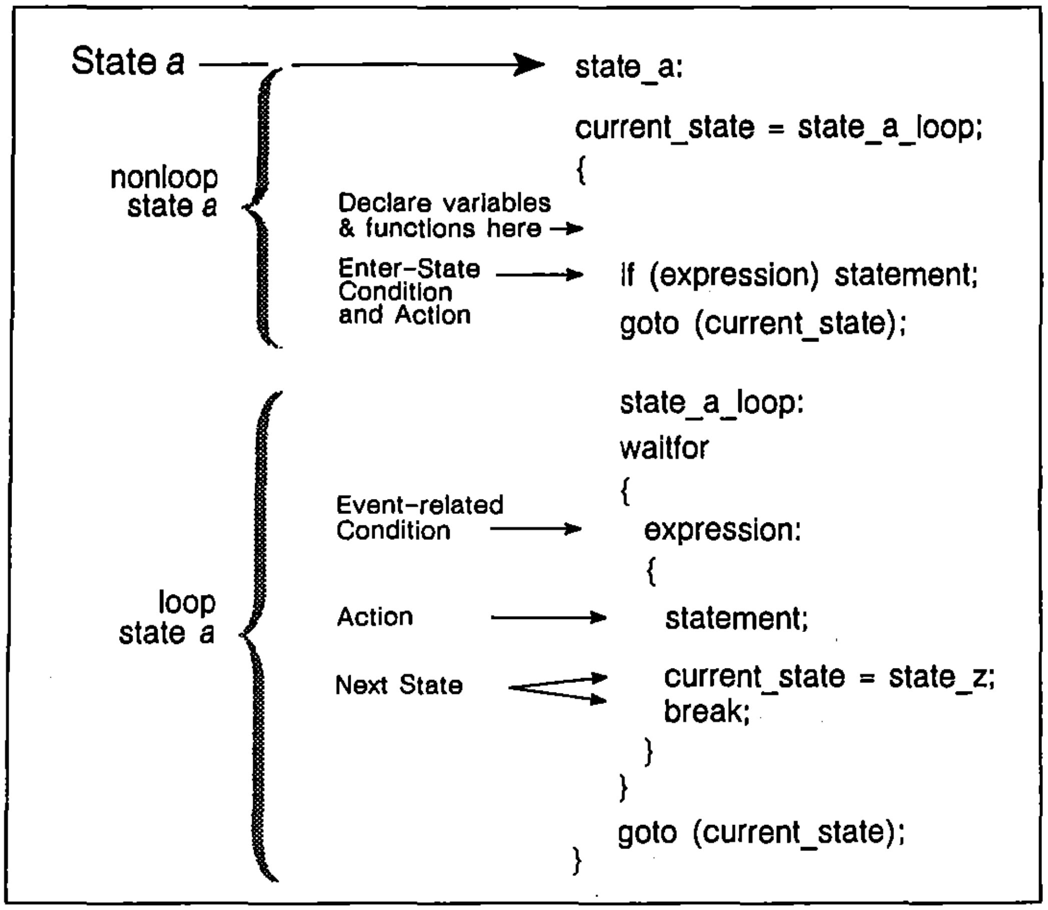


Figure 56-2 Basic C structure of a spreadsheet state.

If the user wants to redefine *current_state*, he may do so in the nonloop state, in which case the loop (*waitfor*) state will be bypassed:

```
static label current_state;
state_alarm_on:
current_state = state_alarm_on_loop;
{
   current_state = state_alarm_off;
   goto (current_state);
   state_alarm_on_loop:
   waitfor
   {
   }
   goto (current_state);
}
state_alarm_off:
/* etc. */
```

The example above is the equivalent of this spreadsheet entry:

```
STATE: alarm_on
   CONDITIONS: ENTER_STATE
   NEXT_STATE: alarm_off
STATE: alarm_off
```

The following hybrid code also will produce the same result. No *break* is necessary, since the translator will place the C region above the *waitfor*.

```
STATE: alarm_on
{
   current_state = state_alarm_off;
}
STATE: alarm_off
```

Or the user may redefine *current_state* in the *waitfor* statement itself, inside the loop state. The only way out of a *waitfor* statement is a *break*, so the translator must furnish a *break* whenever it converts a NEXT_STATE action into C (unless, as in the example above, the condition that triggered the NEXT_STATE action was ENTER_STATE, and consequently the program never entered the *waitfor* loop). The following example uses NEXT_STATE:

```
STATE: alarm_on
   CONDITIONS: KEYBOARD " "
   ACTIONS: ALARM
      PROMPT "press space bar--alarm now disabled"
   NEXT_STATE: alarm_off
STATE: alarm_off
   CONDITIONS: KEYBOARD " "
   ACTIONS: PROMPT "press space bar--alarm is activated"
   NEXT_STATE: alarm_on
```

Here is the C version:

```
static label current_state;
state_alarm_on:
current_state = state_alarm_on_loop;
{
   goto (current_state);
   state_alarm_on_loop:
   waitfor
   {
      keyboard_new_any_key && (keyboard_any_key == ' '):
      {
         sound_alarm();
         display_prompt ("press space bar--alarm now disabled");
         current_state = state_alarm_off;
         break;
      }
   }
   goto (current_state);
}
```

```
state_alarm_off:
current_state = state_alarm_off_loop;
{
   goto (current_state);
   state_alarm_off_loop:
   waitfor
   {
      keyboard_new_any_key && (keyboard_any_key == ' '):
      {
         display_prompt ("press space bar--alarm is activated");
         current_state = state_alarm_on;
         break;
      }
   }
   goto (current_state);
}
```

Various hybrid versions are possible. Here is one:

```
STATE: alarm_on
   CONDITIONS:
   {
      keyboard_new_any_key && (keyboard_any_key == ' ')
   }
   ACTIONS:
   {
      sound_alarm();
      display_prompt ("press space bar--alarm now disabled");
      current_state = state_alarm_off;
      break;
   }
STATE: alarm_off
   CONDITIONS:
   {
      keyboard_new_any_key && (keyboard_any_key == ' ')
   }
   ACTIONS:
   {
      display_prompt ("press space bar--alarm is activated");
      current_state = state_alarm_on;
      break;
   }
```

56.3 Conditions and Actions

When a condition is translated into C code by the INTERVIEW, the resulting expression is enclosed in braces at the top of a *waitfor* statement. The only exception to this rule is the ENTER_STATE condition—see Section 56.2(B), above.

The conditional expression is followed by a colon and then by the statement that constitutes the action to be taken when the condition is true. If more than one action is coded, braces must be used to form a statement block. See Figure 56-3.

Typing CONDITIONS: KEYBOARD " " on the spreadsheet is the equivalent of this C coding, placed inside of the braces that follow the reserved word *waitfor*:

```
keyboard_new_any_key && (keyboard_any_key == ' '):
{
   /* action-statements or routines go here */
}
```

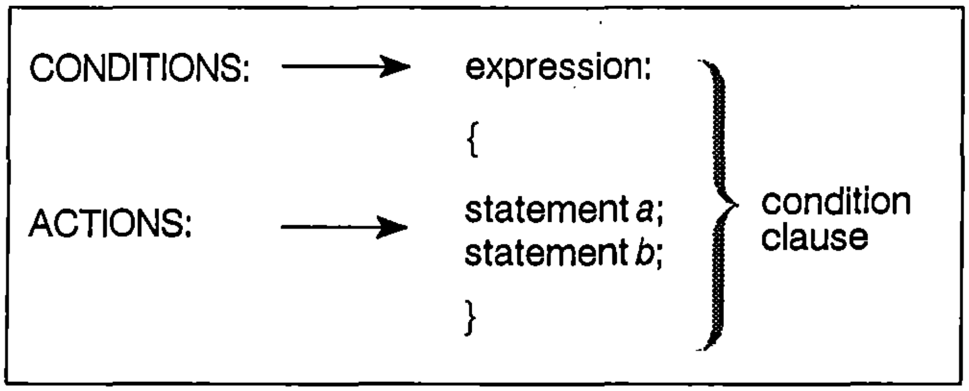


Figure 56-3 The translator converts the Condition-and-Action "trigger" into a condition clause inside of a *waitfor* statement.

(A) Multiple Condition Clauses

Following the semicolon that terminates the statement (or following the statement block), you may enter another condition clause. These clauses correspond to triggers on the Trigger menus or conditions-and-actions blocks inside a state on the Protocol Spreadsheet. Multiple condition clauses may be placed inside of one *waitfor* construction. (There is only one *waitfor* statement per state.)

Here is an example of a state with two "triggers":

```
STATE: keyboard_prompt
  CONDITIONS: KEYBOARD "1"
  ACTIONS: ALARM
    PROMPT "You have pressed the 1 key."
  CONDITIONS: KEYBOARD "2"
  ACTIONS: ALARM
    PROMPT "You have pressed the 2 key."
```

A version in C would have two condition clauses:

```
state_keyboard_prompt:
waitfor
{
  keyboard_new_any_key && (keyboard_any_key == '1'):
  {
    sound_alarm();
    display_prompt ("You have pressed the 1 key.");
  }
  keyboard_new_any_key && (keyboard_any_key == '2'):
  {
    sound_alarm();
    display_prompt ("You have pressed the 2 key.");
  }
}
```

If you are mixing spreadsheet tokens with C, place condition clauses inside of STATE: blocks. Any C region at the top of a State block is placed above the automatic *waitfor* statement. You must therefore supply your own *waitfor* word, since a condition clause is syntactically valid only inside of a *waitfor*. An example follows.

```
STATE: keyboard_prompt
{
   waitfor
   {
      keyboard_new_any_key && (keyboard_any_key == '1'):
      {
         sound_alarm();
         display_prompt ("You have pressed the 1 key.");
      }
      keyboard_new_any_key && (keyboard_any_key == '2'):
      {
         sound_alarm();
         display_prompt ("You have pressed the 2 key.");
      }
   }
}
```

A word of warning is in order. When your program executes this code, it will find itself stuck in a *waitfor* statement beneath the label *state_keyboard_prompt*. If you want to exit this *waitfor*, you must execute a *break* in a statement block in one of the condition clauses. Once you have broken outside of the *waitfor*, you may *goto* another state.

If you add softkey CONDITIONS, ACTIONS, or NEXT_STATE blocks to the state above, they will be placed inside a different *waitfor* statement, the one that is created automatically inside a state called *state_keyboard_prompt_loop*. See Section 56.2 (particularly Figure 56-2). What may look like a single state on the spreadsheet really will be two different states which never are active at the same time.

(B) Multiple Expressions

Expressions may be logically *and*ed (&&) or *or*ed (||) together inside a condition clause. Here is the spreadsheet version of a CONDITIONS block with two expressions:

```
CONDITIONS: KEYBOARD "2"
   FLAG keyboard_disabled 0
ACTIONS: PROMPT "You have pressed the 2 key."
```

Inside the condition clause in C, the translator supplies a double ampersand (&&) to connect the keyboard expressions with the flag expression:

```
keyboard_new_any_key && (keyboard_any_key == '2')
&& (flag_keyboard_disabled.current == 0):
{
   display_prompt ("You have pressed the 2 key.");
}
```

Inside a CONDITIONS block, the translator is able to *and* a softkey condition correctly with a C expression. Note that the user types the C expression without a terminating colon. The translator will supply one later:

```
CONDITIONS: KEYBOARD "2"
{
   flag_keyboard_disabled.current == 0
}
ACTIONS: PROMPT "You have pressed the 2 key."
```

The *and*ing is also successful when the C expression is placed above the softkey condition inside the CONDITIONS block:

```
CONDITIONS:
{
   flag_keyboard_disabled.current == 0
}
   KEYBOARD "2"
ACTIONS: PROMPT "You have pressed the 2 key."
```

If you want to insert a comment into a Conditions block, remember that the translator does not look inside of C regions (except to expand constants). It will take the comment and *and* it with the rest of the expressions in the Conditions block. Since a comment is not a C expression, the program will not compile: see Section 56.3(D). Note in the following example that a 1 has been inserted inside the C region along with the comment in order to make the code compile and in order to make the expression "true."

```
CONDITIONS:
{
   /* This comment will be anded with the keyboard expression. */ 1
}
   KEYBOARD "2"
ACTIONS: PROMPT "You have pressed the 2 key."
```

(C) Event Variables

The translator converts most Conditions blocks on the Protocol Spreadsheet into two or more expressions linked by the logical *and* operator (&&). The keyboard condition in the examples above was typical: KEYBOARD "2" on the spreadsheet became a pair of expressions logically *and*ed in C.

The first expression, *keyboard_new_any_key*, is an event variable. Event variables are very important in the INTERVIEW implementation of C, and the programmer should observe the following rules of thumb:

1. *An event variable usually is paired with a nonevent variable.* At the moment an event variable comes true in a *waitfor* construction, all nonevent (or "status") variables attached to that event variable are evaluated for truth or falsity. Whenever any keyboard key is struck, the event variable *keyboard_new_any_key* comes true. At that moment, the nonevent expression *keyboard_any_key* == '2' is evaluated to determine whether it is true or false.

2. *A* waitfor *statement must include at least one event expression.* A *waitfor* statement without an event variable will not compile. There must be some event that *might* transpire to cause the nonevent expressions to be evaluated.

3. *An event variable may appear alone in an expression.* It is possible (though unusual) to have an event expression that is not *and*ed with a nonevent expression. When the translator converts CONDITIONS: DTE GOOD_BCC into C, for example, the resulting expression is this simple event variable:

   fevar_gd_bcc_td:

4. *A nonevent variable also may appear alone.* It also is possible (though the translator does not do this inside of *waitfor* statements) to have a nonevent expression that is not *and*ed with an event expression—as long as there is an event expression somewhere in the *waitfor* construction. The following program will compile and work:

```
{
  extern fast_event keyboard_new_any_key;
  extern volatile unsigned short keyboard_any_key;
}
     STATE: keyboard_prompt
     CONDITIONS:
     {
       keyboard_new_any_key && (keyboard_any_key == '1')
     }
     ACTIONS: PROMPT "You have pressed the 1 key."
     CONDITIONS:
     {
       keyboard_any_key == '2'
     }
     ACTIONS: PROMPT "You have pressed the 2 key."
```

   In this example, *keyboard_any_key* == '2' is not *and*ed with an event variable. As a result, it is attached automatically to the event variable *keyboard_new_any_key* in the Conditions block above. If there had happened to be other event variables in the state, it would have been attached to them as well; so that when any event in the state came true, *keyboard_any_key* == '2' would be evaluated.

   NOTE: Other event variables in the state would cause *keyboard_any_key* to be *evaluated*, but would not necessarily cause it to be *updated*. Event variables are guaranteed to update only their associated nonevent variables. In the example above, *keyboard_any_key* is an associated nonevent variable for the event variable *keyboard_new_any_key*.

5. *Two event variables may not be combined.* Two event variables may never be combined in a condition clause, since two events never are simultaneous. Since all spreadsheet conditions have event variables associated with them—counter conditions have the *counter_name_change* event variable, for example—it might seem impossible to combine a counter with another

condition in a single CONDITIONS block. In fact, in the case of a few special combinable conditions—buffer-full, counter, flag, and EIA are examples—the translator will sometimes omit the event variable. When two or more combinable conditions are combined, the translator uses a first come, first served rule that is explained in Section 57.3, Programming Considerations.

(D) Evaluating Nonevent Expressions

Nonevent expressions are true if they have a nonzero value. In the following program, the "trigger" will sound the alarm when any keyboard key is struck because all of the nonevent expressions are nonzero:

```
{
  extern fast_event keyboard_new_any_key;
}
    STATE: boolean
    CONDITIONS:
    {
      keyboard_new_any_key && 1 && 99 && 10003
    }
    ACTIONS: ALARM
```

This version never will sound the alarm, because one of the *and*ed components is zero:

```
{
  extern fast_event keyboard_new_any_key;
}
    STATE: boolean
    CONDITIONS:
    {
      keyboard_new_any_key && 1 && 0 && 10003
    }
    ACTIONS: ALARM
```

Relational expressions like *keyboard_any_key* == '2' and logical expressions connected by && (like those above) and || are defined automatically to have the value 1 if true and 0 if false.

(E) Multiple Statements

Statements may be blocked together inside a condition clause. Here is the spreadsheet version of an ACTIONS block with two statements:

```
CONDITIONS: KEYBOARD "2"
ACTIONS: PROMPT "You have pressed the 2 key."
  ALARM
```

The C version is a condition clause with two routines, *display_prompt* and *sound_alarm*, inside a block or compound statement:

```
keyboard_new_any_key && (keyboard_any_key == '2'):
{
  display_prompt ("You have pressed the 2 key.");
  sound_alarm();
}
```

A hybrid version, part spreadsheet language and part C language, will work:

```
CONDITIONS: KEYBOARD "2"
ACTIONS: PROMPT "You have pressed the 2 key."
{
   sound_alarm();
}
```

The hybrid example as it stands will not allow you to declare routines and variables, because the translator will place these declarations in a statement block beneath the *display_prompt* routine. For declarations, move the C region to the top of the Actions block; or use double braces to open a new statement block lower down, since declarations are legal following the left brace that introduces any compound statement.

56.4 Example of Complete C Program

Some of the examples in the previous pages of this section were incomplete, in that they included variables that were not declared, or they lacked a softkey **STATE** that could generate a proper program main. The following is an extended example that compiles and runs. It includes many of the pieces that formed the shorter examples in this section. It is written for the Protocol Spreadsheet *as completely as possible* in C. (See Section 55.3 on how to write a program completely in C.)

```
{
   extern fast_event keyboard_new_any_key;
   extern volatile unsigned short keyboard_any_key;
   task
   {
      main()
      {
         static label current_state;
         state_alarm_on:
         current_state = state_alarm_on_loop;
         {
            goto (current_state);
            state_alarm_on_loop:

         waitfor
            {
               keyboard_new_any_key && (keyboard_any_key == ' '):
               {
                  sound_alarm();
                  display_prompt ("press space bar--alarm now disabled");
                  current_state = state_alarm_off;
                  break;
               }
            }
            goto (current_state);
         }
         state_alarm_off:
         current_state = state_alarm_off_loop;
         {
            goto (current_state);
            state_alarm_off_loop:
            waitfor
            {
               keyboard_new_any_key && (keyboard_any_key == ' '):
                {
                  display_prompt ("press space bar--alarm is activated");
                  current_state = state_alarm_on;
                  break;
                }
              }
              goto (current_state);
            }
          }
        }
        layer_1_test_keyboard_alarm;
      }
          STATE: task_list
          {
            layer_1_test_keyboard_alarm();
          }
```

56.5 Summary of C Regions

The translator removes the outer braces from a C region and places it into one of the six basic levels of source code shown in Figure 56-4.

(A) Declarations

Declare your variables and routines in a C region, delimited by curly braces { and }, at the top of your program or at the top of a Constants, Layer, Test, State, or Actions block. Declare a variable preceded by its type descriptors and followed by a semicolon, as in these examples:

```
{
  extern fast_event keyboard_new_key;
  extern fast_event keyboard_new_any_key;
  extern fast_event fevar_time_of_day;
  short minutes;
}
```

We have not bothered to declare routines in most of the examples in the manual, since it is not necessary. In the absence of a declaration, the compiler assumes that the routine is external and that it returns an integer. In nearly all cases, this assumption works. In the few cases where a routine returns a *long* (*get_68k_phys_addr* is an example), it must be declared.

1. *Automatic declaration.* In cases where the translator declares a variable automatically, the user does not have to declare the variable himself. For example, a KEYBOARD condition, when entered via softkey, will declare the variable *keyboard_new_key* automatically for the entire program. When a variable has been declared twice in a program block, the program may not run. Instead, the compiler will put up a message such as the following: *Error 110: keyboard_new_key redeclared.* In software version 5.00 and in earlier software, the compiler flagged double declarations and aborted the compilation.

   Sometimes it is difficult to keep track of the exact version of a variable that the translator is declaring. Some external variables have been improved for the use of C programmers, and we have documented the newer version in our tables and in many of our examples. The translator may still use an older version of the variable.

In an earlier software release, for example, the variable *extern event keyboard_new_key* was speeded up and renamed *extern fast_event keyboard_new_key*. The translator still uses the older name to declare the variable.

The variable *keyboard_new_any_key* is a still more recent improved version of *keyboard_new_key*—improved in that it detects the striking of non-ASCII keys as well as the ASCII set. The translator never declares *keyboard_new_any_key* automatically.

Similarly, the translator uses an older version of *extern fast_event fevar_eia_changed*. The older version is *extern event evar_eia_changed*. In the earlier software, compiler error messages such as "*keyboard_new_key redeclared*" and "*Variable fevar_eia_changed undeclared*" will inform you what the translator is doing in each instance.

2. *Legal declaration*. Declarations are legal following the left brace that introduces any compound statement. Figure 56-4 shows that when the user opens a braced C region following a TEST:, STATE:, or ACTIONS: keyword, the translator removes the outer braces from the C region and plants the C code just inside the left brace at Level 2, 4, and 6 of the source code. Declarations therefore are valid at the top of these regions.

   Declarations should be grouped at the top of any region, since they are not allowed in a statement block below an executable statement. This program will not compile, because the *sound_alarm* routine precedes a declaration:

```
{
  extern fast_event fevar_eia_changed;
}
     STATE: lead_changes
        CONDITIONS:
        {
          fevar_eia_changed
        }
        ACTIONS:
        {
          sound_alarm();
          int lead_changes;
          lead_changes ++;
        }
```

   Declarations never are legal at Level 5 (Figure 56-4)—that is, preceding the colon in a condition clause inside a *waitfor* statement. Declarations always are legal at Level 1, since there are no executable statements at that level.

   The set of variables listed as *extern* cannot be declared below Level 1. *Extern* has a specialized meaning at the task level or lower: it is used to "forward-declare" a variable without actually reserving storage space. The variable must be declared again (but not as *extern*) in the body of the task.

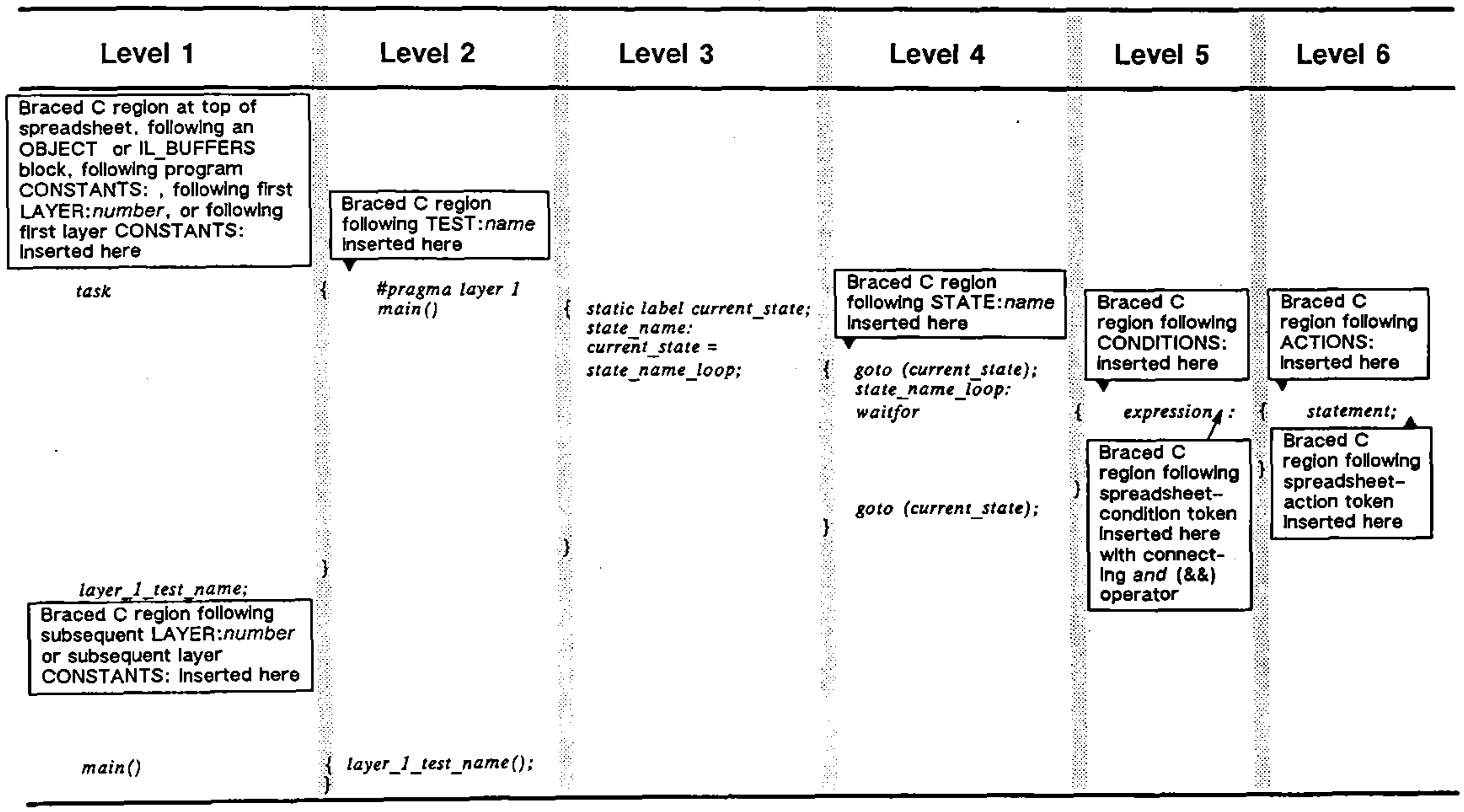

Level 1
Level 2
Level 3
Level 4
Level 5
Level 6
Braced C region at top of spreadsheet, following an OBJECT or IL_BUFFERS block, following program CONSTANTS: , following first LAYER:*number*, or following first layer CONSTANTS: inserted here
task
layer_1_test_name;
Braced C region following subsequent LAYER:*number* or subsequent layer CONSTANTS: inserted here
main()
Braced C region following TEST:*name* inserted here
{ #pragma layer 1
main()
}
{ layer_1_test_name();
}
{ static label current_state;
state_name:
current_state =
state_name_loop;
}
Braced C region following STATE:*name* inserted here
{ goto (current_state);
state_name_loop:
waitfor
goto (current_state);
}
Braced C region following CONDITIONS: inserted here
{ expression :
}
Braced C region following spreadsheet-condition token inserted here with connecting *and* (&&) operator
Braced C region following ACTIONS: inserted here
{ statement;
}
Braced C region following spreadsheet-action token inserted here

Figure 56-4 The translator removes the outer braces from a C region and places it into one of six basic levels of source code. The "telescoping" of the braces indicates the scope of declarations. A variable or routine declared for Level 1 is declared for the remainder of Level 1 and across all levels to the right.

3. *Scope.* The "telescoping" of the braces in Figure 56-4 indicates the scope of declarations. A variable or routine declared for Level 1 is declared for the remainder of Level 1 and across all levels to the right. This means that a variable or routine declared at the top of Level 1 will be global throughout the program. You can force a declaration to the top of Level 1 by placing it in braces (1) at the top of the Protocol Spreadsheet; (2) before or after an OBJECT or IL_BUFFERS block; (3) inside a CONSTANTS block above the Layer level; (4) inside the first LAYER block on the spreadsheet; or (5) inside the CONSTANTS block in the first LAYER block.

   Here is an example of a global declaration:

```
{
  extern fast_event fevar_ela_changed;
}
LAYER: 1
  TEST: leads
    STATE: init
      CONDITIONS:
      {
        fevar_ela_changed
      }
      ACTIONS: PROMPT "Status of a lead has changed."
```

   A variable or routine declared at Level 1 (Figure 56-4) is declared for subsequent layers and tests, whether the subsequent layer is higher or lower. The concept of higher and lower layers is relevant to softkey entry on the Protocol Spreadsheet, but is not carried over into the source code. To the compiler, a TEST in Layer 2 and a TEST in Layer 3 are simply concurrent tasks. The task that is first in the program is compiled first. That is the only meaning of "higher" and "lower" to the compiler.

   A variable or routine may have its scope limited to a particular Test, State, or Actions block. A variable or routine also may be redeclared at different levels. Given more than one valid declaration, the lower or *nearer* one applies.

4. *Initialization.* A variable must be of the *static* storage class to pass its value into a *waitfor* statement. Declarations at Level 1 of the source code (Figure 56-4) are always *static*, whether or not they are declared so. A variable that is initialized at Level 4 (Figure 56-4) must be declared as *static* by the programmer if the initialized value is to be used inside a *waitfor*.

(B) Statements

Executable statements may occur at four levels (Figure 56-4) in the source code: at Level 2 of the program *main* function, where the function is defined; at Levels 3 and 4, where the task *main* function is defined; and at Level 6, inside a *waitfor* statement. The programmer has no access to Level 3. To access Level

4, the programmer may open a C region just beneath the STATE: *name* identifier. He may access Level 6 by opening a braced C region below the ACTIONS: keyword.

Levels 1 and 2 are reserved for declarations. The program *main* function executes statements at Level 2 (see the bottom of Figure 56-4), but this function is accessible only to the translator.

57 Events

In Run mode, the user program in the INTERVIEW moves from program STATE to program STATE. In each state a set of conditions is tested, with one or more actions the result of a particular condition coming true.

In the INTERVIEW's implementation of C, a "state" is a special control structure called a *waitfor* clause that is placed in the program directly following a *label* named for the state. Program movement is controlled by *goto* statements that reference these labels.

Each *waitfor* clause defines a set of interrupts ("events") that it is waiting for. When a *waitfor* clause is active and an interrupt/event occurs that is defined in that clause, the entire clause is processed. All of the conditions in the clause are tested and appropriate actions (statements, operations, routines) are executed.

The *waitfor* clause is a mechanism designed specifically for the data-communications testing environment, in which the program must interact at high speed with a variety of unpredictable inputs.

57.1 Example of Event: *fevar_time_of_day*

In the *waitfor* clause in an earlier example (Section 55 of this volume), the condition was this:

```
fevar_time_of_day && (crnt_time_of_day == 1300)
```

Once every minute, the CPM sends an interrupt to the MPM. This interrupt takes the form of a *fevar_time_of_day* event.

If the program includes a *fevar_time_of_day* condition, the interrupt each minute will cause the variable *crnt_time_of_day* to be updated.

If the current state includes a *fevar_time_of_day* condition, the interrupt each minute will satisfy that condition. At the same time all other conditions in the clause, including non-event (that is, non-interrupt-driven) conditions such as *crnt_time_of_day == 1300*, will be tested.

The relationship between an event variable such as *fevar_time_of_day* and its associated nonevent variable (in this case, *crnt_time_of_day*) can be summarized as follows: the event variable anywhere in the program causes the nonevent variable to be updated each time the event occurs. The event variable in the currently active *waitfor* loop causes the nonevent condition to be tested each time the event occurs.

Figure 57-1 illustrates this relationship, as well as the relationship between an event and a nonassociated variable. The figure shows, for example, how an EIA event might cause the time-of-day variable to be checked but not updated; and how a time-of-day event might cause the EIA-status variable to be checked but not updated. "Event" in the figure means event variable, while "variable" means nonevent variable.

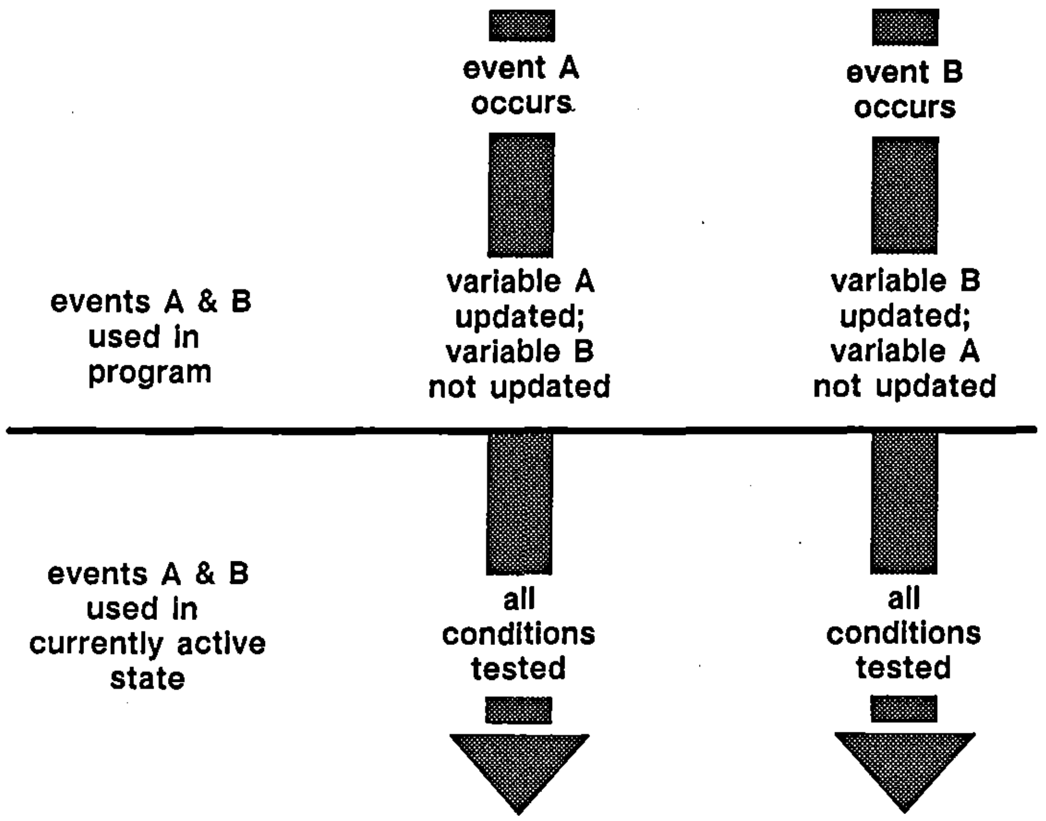

event A occurs
event B occurs
events A & B used in program
variable A updated; variable B not updated
variable B updated; variable A not updated
events A & B used in currently active state
all conditions tested
all conditions tested

Figure 57-1 This figure is meant to show the effect of event A on its associated variable (variable A) as well as its effect on a nonassociated variable (variable B).

57.2 Various Origins of *waitfor* Events

Interrupts sent to the MPM from the CPM include *fevar_time_of_day* and *keyboard_new_key*. Interrupts sent to the MPM by the SCC (Serial Communications Controller) chip in the FEB include *fevar_rcvd_char_td*, *fevar_gd_bcc_rd*, and *fevar_eia_changed*. Some interrupts are sent to the user program by the protocol state machines in the layer packages. Examples are *dce_frame* and *dte_packet*.

Interrupts also can be generated by the program itself. The program sends an interrupt in the form of a "signal." *counter_name_change* and *flag_name_change* are events that are signaled by the program itself, since the program is in charge of all counter and flag increments, decrements, and sets.

57.3 Programming Considerations

By itself in a *waitfor* clause, *crnt_time_of_day == 1300* never can be true, since only interrupts/events cause the nonevent conditions in the clause to be processed. On the other hand, *counter_name_change && flag_name_change* never can return true, since two events cannot occur simultaneously.

Because two events never are simultaneous, the programmer (and the built-in translator) has a decision to make whenever two nonevent conditions, such as *counter_name.current == 3* and *flag_name.current == 5*, are *and*ed together. If the programmer writes *counter_name_change && (counter_name.current == 3) && (flag_name.current == 5)*, the condition may be true when *counter_name.current* transitions to 3 but it never will be true when *flag_name.current* transitions to 5, since there is no interrupt to cause the condition to be checked at that moment. If an interrupt (*flag_name_change*) is tied to *flag_name.current*, then *counter_name.current* transitioning to 3 will not be detected.

When the user combines a flag condition with a counter condition on a single Trigger Setup menu, the translator solves the dilemma of which event to "wait for" by generating a two-pronged *waitfor* condition that is approximately the following:

```
(counter_name_change && (counter_name.current == 3) &&
(flag_name.current == 5)) || (flag_name_change &&
(counter_name.current == 3) && (flag_name.current == 5));
```

On the Protocol Spreadsheet, the translator simply attaches the appropriate event variable to the first softkey condition listed. If the user enters

```
CONDITIONS: COUNTER name EQ 3
   FLAG name 101
```

the translator converts this to *(counter_name_change && (counter_name.current == 3) && (flag_name.current == 5)*. The user is then free to repeat the combined condition, reversing the order of the elements (and therefore invoking the *flag_name_change* interrupt) the second time around.

NOTE: The examples in Section 57.3 above are somewhat simplified. The actual translator versions are made more complicated by the inclusion of *counter_name.old* and *flag_name.old* variables that are explained in Section 65.

58 Receiving and Transmitting Data

As the INTERVIEW monitors the data source (line or disk), it signals the arrival of each character by an event variable (*fevar_rcvd_char_rd* or *fevar_rcvd_char_td*) and it stores each character momentarily in a variable (*rcvd_char_rd* or *rcvd_char_td*) accessible by the user. Data can be taken from the line in this form and copied into memory or into an interlayer message buffer. BOP-framed data is copied automatically into an interlayer ("IL") buffer.

The user transmits data from the INTERVIEW by creating a transmit-data structure and then referencing the structure in an *ll_transmit* routine. Or the user may copy the data into an interlayer buffer (or simply reference the data in the buffer) and then call out the buffer in an *ll_il_transmit* routine.

The IL buffers have several advantages as a storage medium for data. First, they are reusable. They are allocated dynamically and erased automatically unless the user takes steps to maintain them. Without these reusable buffers, data in Run mode would quickly eat up all of the memory in the unit.

Second, IL buffers support linked lists. There are routines that will start a list, insert data at the top of a list, and append data to the bottom of a list. Linked lists are well suited to layered-protocol transmissions, where the transmit string is built incrementally as the transmission moves down the layers.

58.1 Locating Data in an IL Buffer

When a BOP frame is placed automatically in an IL buffer, a data primitive is created automatically and the event variable *m_lo_ph_prmtv* is signaled. The segment number of the IL buffer is recorded in the variable *m_lo_ph_il_buff*. The offset from the start of the buffer to the start of the data is recorded in the variable *m_lo_ph_sdu_offset*. This offset is always 32 bytes. What is considered data at higher layers may have a larger offset, since each layer's data begins farther into the frame. See Figure 58-1 for an illustration of a gradually shrinking "service data unit" (SDU) and a gradually expanding SDU offset.

By default, there are sixteen IL buffers. (See Sections 27.5 and 66 for information on changing the number/size of IL buffers.) The address of each memory location in these buffers is 32 bits. The high-order 16 bits is the 80286 segment number. This is the number that the software passes around when it wants to identify an IL buffer, simply because 16 bits are easier and faster to pass around than 32 bits, the low-order 16 of which are always zero when we are discussing the starting location of each buffer.

When we want to look at data in the buffer, we need to reference not a 16-bit segment number but a 32-bit address. So we cast the segment number (always a *short*, 16 bits) into a *long* and move the number over to its high-order position, sixteen bits to the left. We add 32 to the number to bypass the header information for the buffer. Then we cast the new *long* as a character pointer. Here, for example, is *m_lo_ph_il_buff* converted into a pointer to the first byte in a frame:

```
char * m_frame_ptr;
m_frame_ptr = (void*)(((long)m_lo_ph_il_buff << 16) + 32);
```

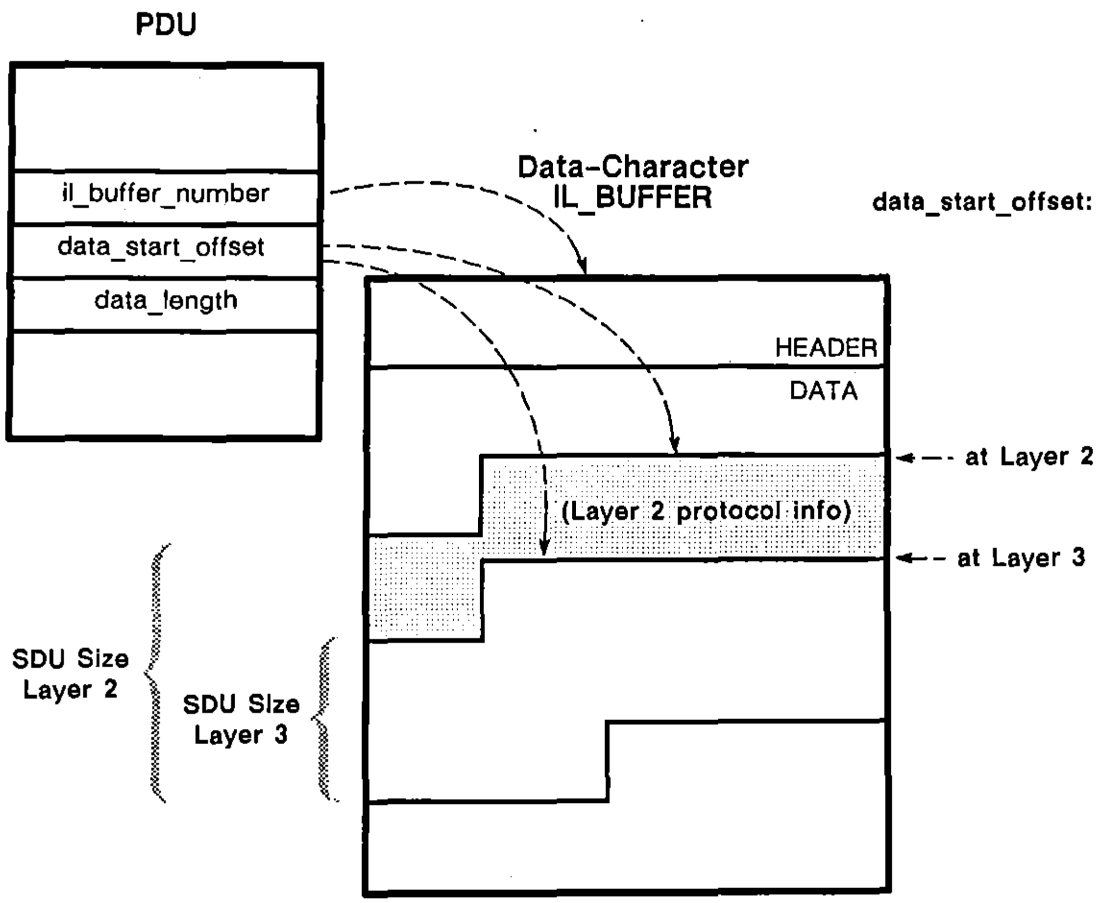


Figure 58-1 When an IL buffer is passed upward, the data offset changes and the data length changes, but the buffer itself does not change.

58.2 Monitor Path vs. Receive Path

The variables *m_lo_ph_prmtv*, *m_lo_ph_il_buff*, and *m_lo_ph_sdu_offset* are part of a set of monitor services that handle IL buffers in both monitor and emulate modes. These variables are updated for data on either data lead. The layer packages use these variables to generate the protocol traces. The translator uses them to implement spreadsheet condition-tokens such as PH_TD_DATA IND and DTE INFO.

Another set of variables are maintained in emulate mode and are updated for data on the receive side only. These variables have names that reveal their obvious relationship to the monitor set: *lo_ph_prmtv*, *lo_ph_il_buff*, *lo_ph_sdu*, etc. These receive-side variables are used by the translator to implement spreadsheet condition-tokens such as PH_DATA IND and RCV INFO.

Whenever a BOP frame is placed automatically in an IL buffer during an emulate run, events *m_lo_ph_prmtv* and *lo_ph_prmtv* both are signaled. The segment number of the same IL buffer is recorded in two variables, *m_lo_ph_il_buff* and *lo_ph_il_buff*.

58.3 Passing a Buffer Upwards

Layer 1 stores data in IL buffers and passes these buffers to Layer 2 automatically, as we have seen. If a Layer 2 personality package is loaded in from the Layer Setup screen, the second data byte in the buffer (the 34th byte overall) is checked to determine the frame type. If the contents of the buffer is an Info frame, a data primitive is created automatically and the event variable *m_lo_dl_prmtv* is signaled. The segment number of the IL buffer is recorded in the variable *m_lo_dl_il_buff*. This is the same segment number that was stored previously in *m_lo_ph_il_buff*.

The offset from the start of the buffer to the start of the data—Layer 2 or data link (DL) data—is recorded in the variable *m_lo_dl_sdu_offset*. This offset is always 34 in MOD 8. This number represents the 32-byte buffer header plus a 2-byte frame header that is of no interest to Layer 3, which will use *m_lo_dl_il_buff* and *m_lo_dl_sdu_offset* to construct its packet trace.

The size of the data component in the buffer is stored in the variable *m_lo_dl_sdu_size*. This number will be 2 bytes smaller than the variable *m_lo_ph_sdu_size*.

If no layer packages are loaded, none of the buffer-handling services are provided automatically at Layer 2 or higher. The programmer can provide the services "manually" as indicated above.

If layer packages are loaded, monitor-path variables (those variables whose names begin with *m_*) are updated automatically in order to drive the protocol traces. Receive-path variables such as *lo_dl_prmtv*, *lo_dl_il_buff*, and *lo_dl_sdu* are generated as needed by GIVE_DATA actions entered by the user on the Protocol Spreadsheet. Otherwise it is up to the C programmer to maintain these variables. For example, the user passing an IL buffer up to Layer 3 might write this code:

```
lo_dl_il_buff = lo_ph_il_buff;
lo_dl_sdu = (lo_ph_sdu + 2);
pdu_ptr->data_length = (pdu_ptr->data_length - 2);
signal (lo_dl_prmtv);
```

The same updates of variables and the same signal would be generated if the user called a *send_dl_prmtv_above* routine, as follows:

```
_set_maint_buff_bit (lo_ph_il_buff, &l2_relay_baton);
send_dl_prmtv_above (lo_ph_il_buff, l2_relay_baton, lo_ph_sdu + 2, pdu_ptr->data_length - 2,
0x45);
```

The *send_dl_prmtv_above* routine requires an SDU size value. There is no receive-path variable (equivalent to *m_lo_ph_sdu_size* on the monitor path) that maintains this value. Determine the SDU size from the *data_length* variable located in the *pdu*-structure. In the examples above, *pdu_ptr* is a structure pointer. The SDU size, therefore, is referenced as *pdu_ptr->data_length*. Refer to Section 66.1 for more information on the *pdu* structure.

> NOTE: Do not use *m_lo_ph_sdu_size* for receive-path routines such as *send_dl_prmtv_above*. It is not updated reliably at the same moment that other receive-path variables are updated.

0x45 is the code for a DL_DATA IND primitive.

58.4 Layer 1 Transmit

Line transmissions are accomplished through L1 transmit routines. Shown below is a program that ends in an *l1_il_transmit* routine. This routine puts the data contents (the service data unit or "SDU," not the buffer header) of an IL buffer out onto the line.

Note that there is a set of routines leading up to the transmit routine. This set of routines is necessary to get a buffer, to start a linked list inside the buffer, and finally to insert several chunks of data into the list before it is transmitted.

```
{
 unsigned short bufnum;
 unsigned short baton;
 unsigned short list_hd_offset;
 static unsigned char data[] = "((FOX))";
 static unsigned char pkt_hdr[3] = {0x10,0x07,0};
 static unsigned char frm_hdr[2] = {0x03, 0};
 int length;
 unsigned short transmit_tag = 1;
}
```

```
STATE: fox
 CONDITIONS: KEYBOARD " "
 ACTIONS:
 {
  _get_il_msg_buff(&bufnum,&baton);
  _start_il_buff_list(bufnum,&list_hd_offset);
  length = sizeof(data) -1;
  _insert_il_buff_list_cnt(bufnum,list_hd_offset,&data[0],length);
  _insert_il_buff_list_cnt(bufnum,list_hd_offset,&pkt_hdr[0],3);
  _insert_il_buff_list_cnt(bufnum,list_hd_offset,&frm_hdr[0],2);
  l1_il_transmit(bufnum,baton,list_hd_offset,transmit_tag);
 }
```

The transmit string will look like this on the INTERVIEW's data display:

```
ᴱₓᴺᵤᴾʟᴿʟᴺᵤTHE QUICK BROWN FOX JUMPS OVER THE LAZY DOG 0123456789 [G]
```

(A) Segment Number

The *l1_il_transmit* routine required four arguments as input. First, it required the segment number of the IL buffer that was intended to be transmitted. This number was supplied by the *_get_il_msg_buff* routine, and we called the number *bufnum*. By default, there are a total of sixteen numbered IL buffers available to the program. You may change the number (and size) of IL buffers via selections on the Protocol Spreadsheet (Section 27.5) or two C preprocessor directives—*#pragma il_buffers* and *#pragma il_buffer_size* (Section 66).

(B) Relay Baton

The second argument was the number of the "relay baton" or "maintain bit." This relay baton was supplied by the *_get_il_msg_buff* routine, and we called the variable that held the number *baton*. A relay baton is passed down automatically with every send or transmit routine and serves to hold the buffer until it has been processed by the next layer (or transmitted by Layer 1). Then the baton is freed.

There are sixteen numbered relay batons available *for each IL buffer*. At the moment that all sixteen batons (or maintain bits) are free, the buffer is returned automatically to the pool of free IL buffers and its contents are no longer available to the program.

In many applications—X.25 Layer 2 and Layer 3 personality packages, for example—an extra maintain bit is reserved (via the *_set_maint_buff_bit* routine) each time a buffer is sent down. This extra maintain bit is held onto in case a frame or packet must be resent, and is not freed (in a *_free_il_msg_buff* routine) until the outstanding frame or packet has been acknowledged.

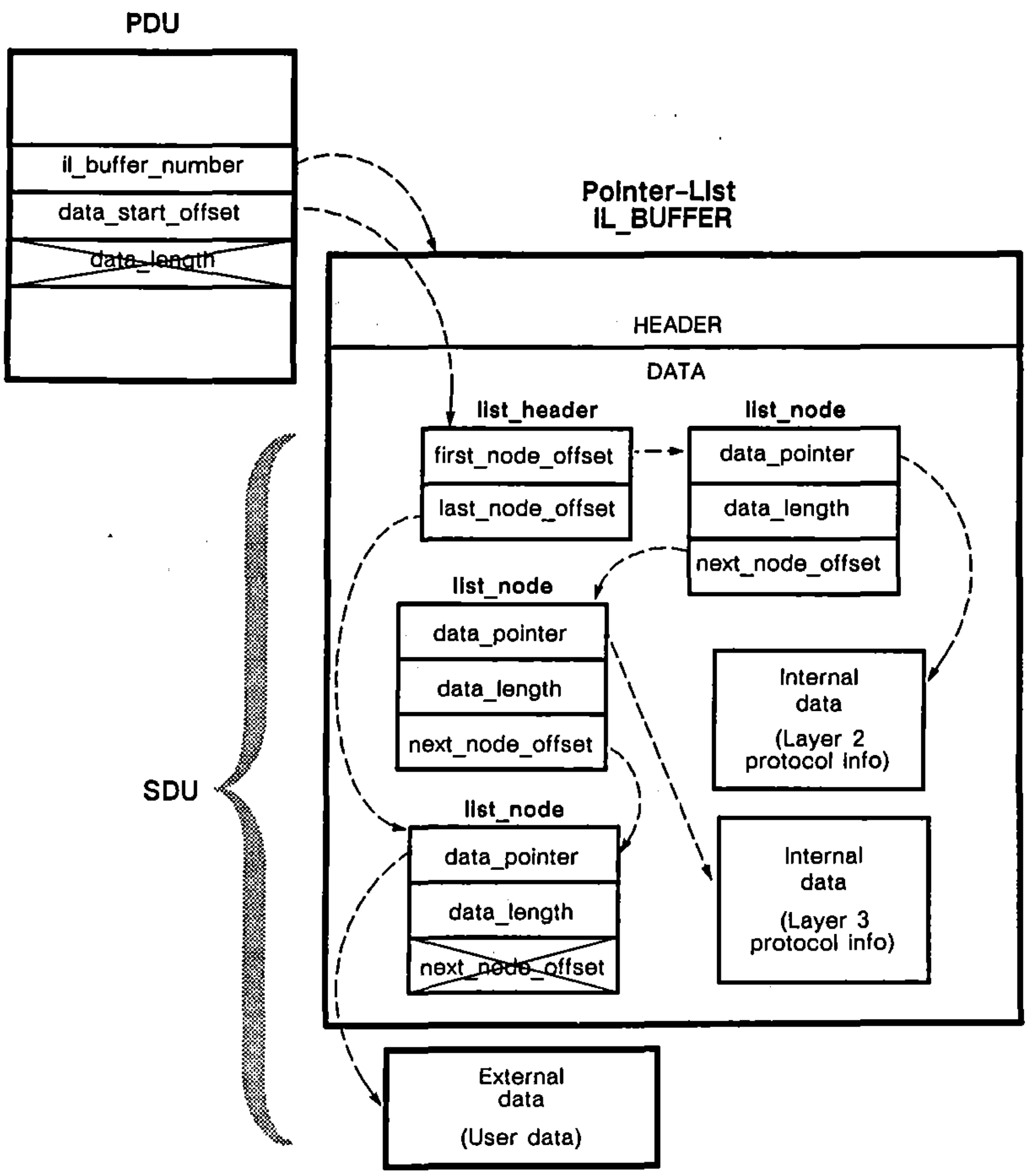


Figure 58-2 When an IL buffer is passed downward, the data-start offset gives the location of the list header. This list header and the various pieces of the transmission (the list nodes) are threaded together.

(C) List-Header Offset

In addition to buffer number and baton number, the *ll_il_transmit* routine also requires as input the offset from the start of the buffer to the linked-list header. This offset is supplied at the moment the linked list is started by the *_start_il_buff_list* routine. In the program above we called this offset *list_hd_offset*.

Figure 58-2 illustrates how the list header ties the linked list together by identifying the offsets to the first and last nodes. A list node is created by each *_insert_il_buff_list_cnt* or *_append_il_buff_list_cnt* routine. The program in Section 58.4 has three *_insert_il_buff_list_cnt* routines. The IL buffer that is transmitted therefore has three list nodes.

(D) Transmit Tag

The fourth argument in the *ll_il_transmit* routine is a "transmit tag" that determines the type of BCC to be appended to the transmission. This variable is stored in the 32-byte header of each IL buffer. Refer to the structure *il_buffer* in the table of OSI structures, Table 66-1.

A transmit tag of 1 means a good BCC and 2 means a bad BCC. 3 causes an aborted transmission.

58.5 Passing a Buffer Between Tasks

At this point we need to modify our *ll_il_transmit* program to allow different layers—which are simply separate concurrent tasks in the programming architecture—to contribute list nodes to the IL buffer intended for transmission. The resulting transmit string will be the same as before, but three different tasks will have contributed data components to the transmitted buffer. In our new program, a Layer 4 task will provide the fox message, Layer 3 will provide the *_insert_il_buff_list_cnt* routine that references the 3-byte packet header, and Layer 2 will provide the insert routine that references the 2-byte frame header.

How do the separate layer tasks communicate with each other so that the right buffer is accepted at the moment it is handed down? They relay information in the same way that tasks always communicate, by signals that are detected throughout the program as event variables. When Layer 4 sends an IL buffer down in a *send_n_prmtv_below* routine, an event variable at Layer 3 (*up_n_prmtv*, not shown in the program below but implied nevertheless in the N_DATA REQ condition) comes true and at the same time updates the variables *up_n_il_buff* and *up_n_sdu*. Layer 3 can use these variables to identify the new IL buffer and to determine the offset to the list header in that buffer. With this information, Layer 3 can insert its own list node into the buffer before passing it down to layer 2.

Here is the program, followed by a few explanatory comments:

```
{
 unsigned short bufnum;
 unsigned short l4_baton;
 unsigned short l3_baton;
 unsigned short l2_baton;
 unsigned short list_hd_offset;
 static unsigned char data[] = "((FOX)) ";
 static unsigned char pkt_hdr[3] = {0x10,0x07,0};
 static unsigned char frm_hdr[2] = {0x03,0};
 int length;
 extern volatile unsigned short up_n_il_buff;
 extern volatile unsigned short up_dl_il_buff;
 extern volatile unsigned short up_n_sdu;
 extern volatile unsigned short up_dl_sdu;
}
LAYER: 4
 STATE: fox
  CONDITIONS: KEYBOARD " "
  ACTIONS:
  {
   _get_il_msg_buff(&bufnum,&l4_baton);
   _start_il_buff_list(bufnum,&list_hd_offset);
   length = sizeof(data) -1;
   _insert_il_buff_list_cnt(bufnum,list_hd_offset,&data[0],length);
   send_n_prmtv_below(bufnum,l4_baton,list_hd_offset,0,0x64,0);
  }
LAYER: 3
 STATE: packet_header
  CONDITIONS: N_DATA REQ
  ACTIONS:
  {
   _insert_il_buff_list_cnt(up_n_il_buff,up_n_sdu,&pkt_hdr[0],3);
   _set_maint_buff_bit(up_n_il_buff,&l3_baton);
   send_dl_prmtv_below(up_n_il_buff,l3_baton,up_n_sdu,0,0x44,0);
  }
LAYER: 2
 STATE: frame_header
  CONDITIONS: DL_DATA REQ
  ACTIONS:
  {
   _insert_il_buff_list_cnt(up_dl_il_buff,up_dl_sdu,&frm_hdr[0],2);
   _set_maint_buff_bit(up_dl_il_buff,&l2_baton);
   send_ph_prmtv_below(up_dl_il_buff,l2_baton,up_dl_sdu,0,0x24,0);
  }
```

In the send-primitive routines, the hex values 64, 44, and 24 identify the primitives as *data* requests. See, for example, the values of *up_n_prmtv_code* in Table 66-4.

Note that there is no longer an *l1_il_transmit* routine in the program. When Layer 2 executes a *send_ph_prmtv_below* routine, Layer 1 handles the transmit function automatically.

The *send_ph_prmtv_below* routine does not have a transmit-tag argument that allows us to specify the BCC. Since the *ll_il_transmit* routine, which has a transmit-tag input, is being handled automatically, it is not immediately clear how you would send the transmit string with a bad BCC. Here is one way. Instead of the *send_ph_prmtv_below* routine at Layer 2, use the *ll_il_transmit* routine as follows:

```
ll_il_transmit(up_dl_il_buff,l2_baton,up_dl_sdu, 2);
```

The 2 in the argument represents the transmit tag for a bad BCC.

If it seems strange to be using an *ll_il_transmit* routine at Layer 2, remember that none of the variables or routines is really layer-specific. In C, layers are simply concurrent tasks.

A "realistic" implementation of this program might be made somewhat more complicated by two additional elements. One or more *_open_space_in_il_buff* routines might be used so that, as far as possible, text data could be copied into the buffer where it would then be erased when the buffer was freed. (One of the advantages of IL buffers is that the space inside them can be recycled.)

Another complication is that for the same transmission, more than one linked list might be started in a single buffer. The example under the *_insert_il_buff_list_cnt* routine in Section 66.3(A) shows Layer 2 accepting a buffer from Layer 3 and starting a new linked list. This allows Layer 3 to reconstruct its original linked list in case a packet-resend is needed.

58.6 Sample Transmit Program: Sync or Async Echo

This application monitors incoming data for text strings bounded by ␂ and ␃ or ␗. It copies these strings into an IL buffer and then echoes them back out onto the line, preceded by two ASCII sync characters. The program will work in most data formats as long as ASCII ␂ and ␃ are included.

The program may be modified for EBCDIC ␖, ␂, ␃, and ␗. Use received-character variables *fevar_rcvd_char_rd* and *rcvd_char_rd* for data received on RD.

```
{
   extern fast_event fevar_rcvd_char_td;
   extern volatile unsigned short rcvd_char_td;
   unsigned short number, length;
   unsigned short il_buffer_number, relay_baton, data_start_offset;
   unsigned char echo_string[100] = {'␖', '␖'};
}
STATE: look_for_stx
   CONDITIONS:
   {
      fevar_rcvd_char_td && rcvd_char_td == '␂'
   }
   ACTIONS:
   {
      number = 2;
      echo_string[number] = rcvd_char_td;
      number++;
   }
   NEXT_STATE: construct_echo_string
```

```
STATE: construct_echo_string
  CONDITIONS:
  {
    fevar_rcvd_char_td
  }
  ACTIONS:
  {
    echo_string[number] = rcvd_char_td;
    number++;
    if ((rcvd_char_td == 'ᴱx') || (rcvd_char_td == 'ᴱʙ'))
    {
      length = number;
    }
  }
  CONDITIONS: RECEIVE GOOD_BCC
  NEXT_STATE: transmit_echo_string
STATE: transmit_echo_string
  CONDITIONS: ENTER_STATE
  ACTIONS:
  {
    _get_il_msg_buff(&il_buffer_number, &relay_baton);
    _start_il_buff_list(il_buffer_number, &data_start_offset);
    _insert_il_buff_list_cnt(il_buffer_number, data_start_offset, echo_string, length);
    l1_il_transmit(il_buffer_number, relay_baton, data_start_offset, 1);
  }
  NEXT_STATE: look_for_stx
```

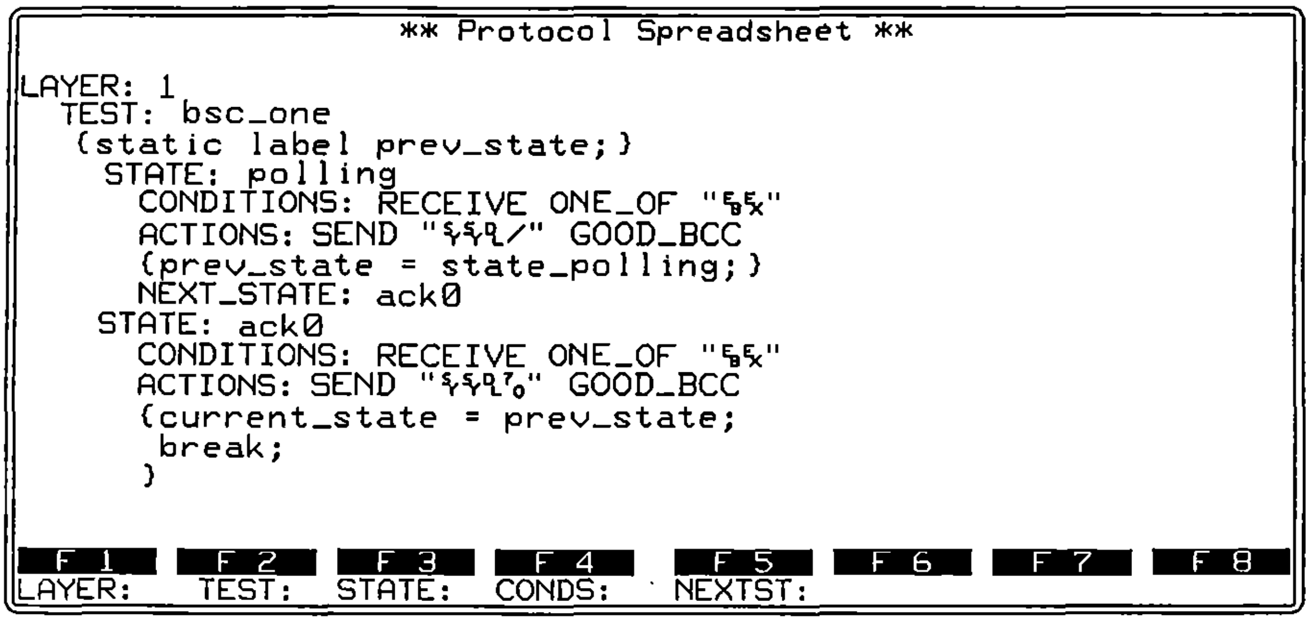
58.7 Sample Transmit Program: BOP Echo

When **Format:** BOP is selected on the Line Setup screen, every frame that is received at the line interface is placed in an IL buffer and passed up to Layer 2. This sample program makes a pointer to the I-field in the most recent IL buffer received at Layer 2, and then it echoes the data back out in the C equivalent of a SEND INFO action. If you try this program, be sure to load the X.25 or SDLC package at Layer 2.

```
{
  char * data_ptr;
  extern volatile unsigned short rcvd_frame_buff_seg;
  extern volatile unsigned short rcvd_frame_sdu_offset;
  extern volatile unsigned short rcvd_frame_sdu_size;
  struct send_frame_structure
  {
    unsigned char addr_type;
    unsigned char frame_type;
    unsigned char nr_type;
    unsigned char ns_type;
    unsigned char p_f_type;
    unsigned char bcc_type;
    unsigned char addr_value;
    unsigned char cntrl_byte;
    unsigned char nr_value;
    unsigned char ns_value;
  };
  struct send_frame_structure frame;
  unsigned short number, baton, offset;
}
```

```
LAYER: 2
  STATE: echo
    CONDITIONS: RCV INFO
    ACTIONS:
    {
      data_ptr = (void *)(((long)rcvd_frame_buff_seg << 16) + rcvd_frame_sdu_offset);
      _get_il_msg_buff(&number, &baton);
      _start_il_buff_list(number, &offset);
      _insert_il_buff_list_cnt(number, offset, data_ptr + 2, rcvd_frame_sdu_size - 2);
      frame.bcc_type = 1;
      send_frame(number, baton, offset, &frame);
    }
```

59 C Basics



```
                    ** Protocol Spreadsheet **

LAYER: 1
  TEST: bsc_one
   {static label prev_state;}
     STATE: polling
       CONDITIONS: RECEIVE ONE_OF "␗␃"
       ACTIONS: SEND "␖␖␐/" GOOD_BCC
       {prev_state = state_polling;}
       NEXT_STATE: ack0
     STATE: ack0
       CONDITIONS: RECEIVE ONE_OF "␗␃"
       ACTIONS: SEND "␖␖␐70" GOOD_BCC
       {current_state = prev_state;
        break;
       }

  F 1      F 2      F 3      F 4       F 5      F 6      F 7      F 8
LAYER:   TEST:    STATE:   CONDS:    NEXTST:
```

Figure 59-1 Using C to return to the previous state.

59 C Basics

C programming language as implemented in the INTERVIEW 7000 Series is based on the current ANSI recommendations. It contains several extensions to the language which enhance its utility in protocol testing, notably multi-tasking.

C is intended as an aid to INTERVIEW users who have advanced programming knowledge. A sophisticated programming tool, C can be applied to testing requirements which are not met by Protocol Spreadsheet selections. C is useful, for instance, in the analysis and "intelligent" manipulation of variable data strings anticipated within a complex protocol. Additional applications of C are the creation of customized protocol and program trace displays.

Figure 59-1 provides a means of returning to whatever state was the former state, without you the programmer knowing which state was previously active. This "go to previous state" function is not a standard spreadsheet feature. The example employs Bisync protocol to demonstrate the usefulness of this capability. The test begins in a state called **polling**. Here, an ACK1 is sent whenever the end of any received data is encountered, and the test passes to the state called **ack0**. This time when the end of received data is encountered, an ACK0 is sent, and the test returns to whatever state it was in formerly.

The first C region is the declaration of the variable *prev_state*, which allows the variable to be used anywhere within the test. In the second C region, the variable *prev_state* is initialized to the name of the active state. The third C region shows the transition of the test to the previously active state. Depending on the contents of the *prev_state* variable, the former state could be one of any number of states. This capability means that, as the programmer expands the simple test, the state ack0 can be used again and again as a utility state from which the test returns to the former state, removing the need for repetitive spreadsheet entry.

59.1 Notable Variations in C

The AR version of C varies in certain respects from the ANSI standard. Notable exceptions to the standard are outlined below. A full set of implementation-defined variations appear in Appendix K.

(A) Reserved Words

The following two reserved words, in addition to those covered in the ANSI standard, are included in C:

task

waitfor

(B) Predeclared Identifiers

The following type identifiers are always predeclared. They are not defined in any *#include* files, nor are their definitions required in any program. Thus they are part of the INTERVIEW C lexicon, even though they are not reserved words and therefore do not appear in the language summary in Appendix K.

event

fast_event

label

(C) Floating Point Notation

Since Floating Point Notation is not required in the protocol testing environment and since corresponding calculations could degrade processing speed, floating point notation is omitted from the AR implementation of C. Fixed point calculations, however, are performed.

(D) Values Returned from C Functions

Functions declared within AR's implementation of C may only return values for data types which are 1, 2, or 4 bytes long. Consequently, a function cannot legally return most structure or union types.

59.2 Editing a C Program

Entries in C are made on the Protocol Spreadsheet, accessed from the Main Program screen. All editing functions available on the spreadsheet can be applied to C coding. Refer to Section 29 for a description of these editing functions.

59.3 Error Reporting in C

Most syntax errors made on the Protocol Spreadsheet are indicated by strike-through of the text where the error occurs. This facilitates correction of entries as you create a test.

Errors which appear in C coding are not indicated by the editor. However, when the program is compiled (when you press [RUN]), the errors will be noted. If there are errors in the program, the INTERVIEW will automatically revert to the Protocol Spreadsheet rather than run the program.

(A) Locating Errors

The cursor is automatically positioned near the first error when the INTERVIEW reverts to the Protocol Spreadsheet. A diagnostic message about the error will be displayed at the top (second line) of the screen. Errors pertaining to the general syntax of the spreadsheet are explained in text. Errors noted by the C pre-processor or compiler are displayed as numbers, with explanatory text if the filename *sys/error_text* is accessible at the moment on a disk. (The file should always be accessible in units with hard disks.) These numbered messages are listed in Appendix A4.

Press GO-ERR again to move down through the spreadsheet to the next error. When you press GO-ERR and there are no more errors, the message "No More Errors" will be displayed.

59.4 Preprocessor Directives

The INTERVIEW supports preprocessor directives *#define* and *#include*. The full set of ANSI preprocessor directives are supported on the INTERVIEW. Included among these directives are *#if*, *#else*, *#ifdef*, *#ifndef*, and *#undef*. (Refer to the ANSI Recommendation for a discussion of these directives.) Implementation-defined *#pragmas* are also preprocessor directives. *#pragma object* and *#pragma hook* are two of the AR *#pragmas*. As the name implies, preprocessor directives are processed before the program in which they appear is compiled.

Preprocessor directives are easy to recognize, since they are always preceded by a pound sign (#). Spaces are significant to the meaning of the directives, since other delimiters are generally not used. Note also that a semi-colon cannot be used to terminate a preprocessor directive. Instead, a directive is terminated by a hard Carriage Return or some indicator of line continuation. Press ⏎ to terminate the directive (no indication of the Return will appear on the screen). Type \ (backslash) and press ⏎ at the end of the line on the screen to indicate that the directive continues on the next line. You may also allow text to wrap to the next line by continuing to type. (Wrapped lines are indicated on the screen by the highlighted symbol ⊞.)

(A) #define

The *#define* directive gives you the convenience of replacing frequently referenced items with a text string of any length.

1. *Placement.* A *#define* directive may be placed at the beginning of a logical line anywhere in a legal C region. The eight valid positions for C regions on the Protocol Spreadsheet are shown in Figure 56-4. The *#define* directive may also be placed in a separate *#include* file. Use the *#include* directive as explained in (B) to invoke the file and make the macro-substitutions it indicates in your main program file.

2. *Format.* The directive follows this format:

   #define identifier string

   For example, if you enter the following line of code,

   #define message The quick brown fox 12345

   the identifier message (wherever it appears exactly as written in the file being acted upon) is replaced in subsequent lines of code by the string The quick brown fox ....12345. The replacement, the macro-substitution, is performed before the code is compiled. When you enter the *#define* directive, leave a space between the directive (*#define*) and the identifier. There should be no spaces in the identifier. The space following the identifier indicates that the next ASCII character (or blank) starts the replacement string. Spaces are allowed and are considered part of the string. Terminate the string (and the directive) as described at the beginning of this sub-section.

3. *Nesting.* *#define* substitutions may be nested. Of course, the nested replacements must be described by a *#define* directive which precedes the *#define* for the replacement text which contains them.

   There is one exception to nesting identifiers—the macro substitution will not be performed when the identifier occurs in a string. In the example below, the programmer tries to nest MAXTRIES within the definition of MESSAGE:

```
#define MAXTRIES 3
#define MESSAGE "Maximum retransmissions is MAXTRIES."
```

   A call to *displayf(MESSAGE);* causes the following to be displayed:

```
Maximum retransmissions is MAXTRIES.
```

   This is certainly not what the programmer intended.

(B) #include

#include files, when invoked in a program, are read into the program file before the program is compiled. As a result, your program has access to commonly used items such as subroutines (input/output and string operations, for example), global variables, constants, and structures without your having to enter or modify the required code repeatedly.

1. *Format.* The format for the directive is as follows:

   #include <filename>

   or

   #include "filename"

#include files follow standard naming conventions. You many also include a single period (.) or double periods (..) in a filename to indicate the current or parent directory. See Section 14.2. As an added convention, the suffix *.h* is appended to the end of the name (as in the filename *stdio.h*).

2. *Search rules for #include files.* The delimiters you use to surround the filename determine how the INTERVIEW searches its filing system for the file.

   - The <> delimiters are intended for files which are supplied by AR. When these delimiters are used, the following directories—and only the following directories—are searched, in the order given:

     1. */sys/include* on current drive (indicated on File Maintenance screen)
     2. The directory named as the current directory on the File Maintenance screen (provided that the current directory is not the root directory for FD1, FD2, or hard disk)
     3. */usr/include* on current drive (indicated on File Maintenance screen)
     4. *FD1/sys/include*
     5. *FD2/sys/include*
     6. *HRD/sys/include*
     7. *FD1/usr/include*
     8. *FD2/usr/include*
     9. *HRD/usr/include*

     NOTE: The directory names are given in the format which the INTERVIEW interprets as the absolute path from the root directory of the disk named before the first slash. So *HRD/sys/include* means */sys/include* on the hard disk.

   - The " " delimiters are intended for user-created files. The same directories are searched for the filename, but they are searched in the following order:

     1. The directory named as the current directory on the File Maintenance screen (provided that the current directory is not the root directory for FD1, FD2, or hard disk)
     2. */usr/include* on current drive (indicated on File Maintenance screen)
     3. */sys/include* on current drive (indicated on File Maintenance screen)
     4. *FD1/usr/include*
     5. *FD2/usr/include*
     6. *HRD/usr/include*

7. *FD1/sys/include*
8. *FD2/sys/include*
9. *HRD/sys/include*

If you have used the same filename for an include file in more than one directory, the file which is actually read in as a result of an *#include* directive will be from the first directory searched which contains that filename. The delimiters you use, then, can make a difference in the file selected for inclusion.

The filename enclosed in <> or " " delimiters may be a relative pathname. The highest directory in the pathname must reside in the current directory or in one of the */include* directories. In response to an *#include "disk_io/stdio.h"* directive, for example, the INTERVIEW first looks for a *disk_io* subdirectory in the current directory on the File Maintenance screen and then for an *stdio.h* file in that subdirectory. If the file is not found, the search for the relative pathname continues according to the sequence designated for " " delimiters.

If the file is not located in any of these directories, an error message is returned to the operator.

(C) #pragma object

Use the *#pragma object* directive to access the compiled routine definitions in a linkable-object file. The OBJECT block-identifier discussed in Section 27.4 may also be used for this purpose. (Also see Section 14.3(P) on creating a linkable-object file—displayed as type LOBJ in the directory listings on the File Maintenance screen).

1. *Placement.* Place the *#pragma object* directive inside any legal C region on the Protocol Spreadsheet. Except for those containing the *static* attribute, routine definitions from an LOBJ file always have global scope. It makes sense, therefore, to position the directive at the top of your spreadsheet program along with other global declarations and definitions.

2. *Format.* The format for the *#pragma object* directive is as follows:

   #pragma object "filename.o"

   A *#pragma object* directive references only one LOBJ filename, but you may include as many directives as you wish.

   The relative or absolute pathname of the linkable-object file is enclosed in quotation marks.

3. *Search rules for linkable-object files.* As your spreadsheet program compiles, the INTERVIEW's filing system is searched for the linkable-object files referenced in *#pragma object* directives.

- If the referenced LOBJ filename begins with *FD1/*, *FD2/*, or *HRD/*, the INTERVIEW interprets it as the absolute pathname and makes only that one search.
- Pathnames beginning with a / indicate that the root directory on each drive should be the beginning point of the search. The drives are searched in the following order: current drive, FD1, FD2, and HRD.
- Otherwise, the name may be a one-word filename, or a relative pathname which includes the directories leading to the file. The highest directory in a relative pathname must reside in the current directory or in one of the */lib* subdirectories. The following directories—and only the following directories—are searched, in the order given:

  1. current directory on the current drive (indicated on the File Maintenance screen)
  2. */usr/lib* on the current drive
  3. */sys/lib* on the current drive
  4. *FD1/usr/lib*
  5. *FD2/usr/lib*
  6. *HRD/usr/lib*
  7. *FD1/sys/lib*
  8. *FD2/sys/lib*
  9. *HRD/sys/lib*

If the pathname is not located in any of these directories, the program will not compile and an error message will be returned to the operator.

4. *How #pragma object works.* When the source of code for the Compile command is FILE, the LOBJ which results usually defines user-created routines. These routine definitions may be "linked," or combined, as needed with your spreadsheet program. This means that routines called within your active program do not always have to be defined on the Protocol Spreadsheet or in *#include* files.

   NOTE: An LOBJ file may also contain *#pragma hook* directives. See Section (D) below. If a *#pragma object* directive references an LOBJ file which contains *#pragma hook* directives, the "hooks" within that file are ignored. Since Compile SPREADSHEET always generates *#pragma hook*s, use the OBJECT block-identifier to reference the resulting LOBJ file.

   (a) *Referenced linkable-object files searched for routine definitions.* If a spreadsheet program calls a routine for which no definition is provided, the LOBJ files referenced in *#pragma object* directives are searched in the order in which they appear on the Protocol Spreadsheet. If a routine is defined in more than one referenced LOBJ file, the definition in the first LOBJ file listed on the Protocol Spreadsheet will be used.

If the routine definition is not found in the spreadsheet program or in any referenced linkable-object file, the compilation will abort. When you go to the Protocol Spreadsheet and look for error messages, the routine name will appear as an unresolved reference.

(b) *Compiled routine definition combined with compiled spreadsheet.* When the routine's definition is located, the compiled code is copied from the LOBJ file and combined with the compiled code of the spreadsheet program.

Routine definitions in an LOBJ file may reference additional routines not defined within the same file. If these indirectly-referenced routines also are not defined on the Protocol Spreadsheet, the LOBJ files are searched again.

Routine definitions containing the *static* attribute are local to the LOBJ file. A *static* routine will be copied from the file only if it is included in the definition of another routine.

NOTE: Use *#pragma object* directives in your active spreadsheet program only. Do no incorporate them in code that will be compiled and saved as an LOBJ file. Although the code will compile, no search for routine definitions in referenced LOBJ files will be performed.

(c) *Efficiently uses memory.* Using *#pragma object* to reference routines in linkable-object files, assists in using the INTERVIEW's memory and spreadsheet buffer efficiently.

- Only the definitions for routines actually called within the current spreadsheet program are copied into memory from the LOBJ file. All other code within the file is ignored.
- When commonly utilized routines are defined in linkable-object files, space in the spreadsheet buffer otherwise dedicated to this purpose can be used for additional programming.
- Since the code in LOBJ files has already been compiled, the INTERVIEW can support a larger program without a corresponding increase in compilation time.

NOTE: Additional *#pragma* preprocessor directives utilized by the INTERVIEW are discussed in other sections of the manual. Refer to Section 64 on Display Window and Trace, for example, for information on the *#pragma tracebuf* directive. Except for *#pragma hook* (below), these other *#pragmas* should be part of the active spreadsheet program, not part of a linkable-object file.

(D) #pragma hook

The *#pragma hook* directive allows compiled C code within a referenced linkable-object file to be automatically combined with the compiled code of an active spreadsheet program. There are eight types of *#pragma hook* directives—hook_types zero through seven. All types may be system-generated during the Compile operation when the source of code is SPREADSHEET, but the resulting linkable-object file always contains at least one hook_type zero.

The programmer also uses hook_type zero (*#pragma hook 0*). For this reason, *#pragma hook 0* will be the focus of the following discussion. The primary purpose of *#pragma hook 0* is to "force" a routine to be called and executed as part of a spreadsheet program, even though no explicit call to the routine is made on the Protocol Spreadsheet. The spreadsheet program *may* also call the routine, but keep in mind that it will be executed twice—once because of the call on the spreadsheet and once because of the call made via the *#pragma hook 0* directive.

1. *Format.* Create hooks on the Protocol Spreadsheet and then write them to a file using the WRITE/U editor command. Before typing your hook on the spreadsheet, press EDIT to prevent the editor from placing a strike-through over the text.

   The format for the *#pragma hook 0* directive is as follows:

   > *#pragma hook hook_type "routine_name();"*

   Follow the directive with a space and enter a decimal (not hexadecimal) constant to identify the *hook_type.*

   After the hook_type, enter another space, and then the *hook text*—C code that calls the routine you want combined with your spreadsheet program. The call to the routine is placed inside quotation marks and includes required syntax—parentheses for the arguments and a semi-colon to complete statement punctuation.

   > NOTE: Task names are always local to a linkable-object file and never *directly* copied from it. The hook text, therefore, cannot reference a task. The rule for exporting tasks from a linkable-object file is to let the *#pragma hook 0* directive call a routine which starts the task(s). See Section 5. following and Section 55 for examples.

   More than one *#pragma hook 0* directive may be present in a single LOBJ file, but each directive calls only one routine.

2. *Routine definitions.* Typically, the definition for the routine called in the directive is located within the same linkable-object file. It may, however, be in another LOBJ file as long as both files are referenced via OBJECT block-identifiers on the Protocol Spreadsheet.

   The definition of the hook-text routine may also reference a task (which must be defined in the same file) or it may reference additional routines not defined within the same file. The rules in Section (C) above for indirectly referencing routines apply.

   Definitions for most of the *extern* routines included in this manual are not strictly required.

3. *Accessing hooks.* If you want the hook text combined with your program, use the OBJECT block-identifier to reference the LOBJ file. If you use the *#pragma object* directive to reference the file, the "hooks" within that file will be ignored.

4. *Hooks are added to task list of program main.* As your program compiles, referenced linkable-object files are searched for hooks. When a hook_type zero directive is found in the file, the hook text is automatically added to the bottom of the task-list in the top-level *main.* If a referenced LOBJ file contains more than one "hook," they will be added to the task list in the order in which they appear in the file.

   NOTE: The order of tasks and hooks in the task-list indicates the order in which *main* initiates tasks and executes hook routines. It does not necessarily indicate the order in which the actions in tasks or hooks are processed.

5. *Execution of hooks.* Recall that the *main* function is system-created during compilation. Refer to Section 55, Program Main. Because *main* simply initiates the execution of each task listed, the (hook-text) routine essentially runs concurrently with the tests in your spreadsheet program.

   Since the hook text is a routine, and not a task, it must actually be executed by *main*, not simply started. The definition of the routine determines when, or whether, any subsequent hooks will be executed by *main.*

   - If the routine's definition references a task, as in the example below, *main* returns quickly, leaving the routine to execute the task. Then *main* begins execution of the next hook in the task list.

```
#pragma hook 0 "example();"
extern fast_event fevar_time_of_day;
extern volatile unsigned short crnt_time_of_day;
```

```
task
 {
  main()
   {
    state_alarm_at_one:
    waitfor
     {
        fevar_time_of_day && (crnt_time_of_day == 1300):
         {
          sound_alarm();
         }
     }
   }
 } example_task;
example()
 {
  example_task();
 }
```

- If the routine's purpose is *not* to start a task (or tasks), then *main* has to execute all the code. The more code there is, the longer it will be before *main* can return to execute the next hook.

  If the definition includes a *waitfor*, as in the following example, any subsequent hooks will never get executed. Instead, *main* will continue to wait for the specified event.

```
#pragma hook 0 "example();"
extern fast_event fevar_time_of_day;
extern volatile unsigned short crnt_time_of_day;
example()
 {
  waitfor
   {
    fevar_time_of_day && (crnt_time_of_day == 1300):
     {
        sound_alarm();
     }
   }
 }
```

59.5 Data Types

(A) Precisions

When a variable is declared, the compiler allocates space in memory according to the *type* declaration that precedes the variable name. There are three sizes (or *precisions*) of data allowable in 80286 memory, and three corresponding data types. A *char* is allotted one byte of memory. A *short* is given two bytes, while a *long* reserves four bytes of memory. Shorts and longs are varieties of *int* or integer, and the type descriptions *short int* and *long int* are permitted. The type *int* used by itself is the same as *short int*.

(B) Signed and Unsigned Types

All three precision types may be *signed* or *unsigned*. Signed and unsigned data types are *stored* identically, but treated differently in arithmetic operations. Specifically, they differ in the way they undergo type conversion, comparison, division, and right shifting.

1. *Type conversion.* The following declarations store the same value in memory:

```
signed char a = -6;
unsigned char b = -6;
```

In both cases, the byte stored in memory will be the two's complement of 00000110, or 11111010. (The two's complement is the one's complement + 1.) This bit pattern translates as hex *fa* or ASCII *z*. The *displayf* routine in the following program will write two *z*'s to the screen:

```
{
  signed char a = -6;
  unsigned char b = -6;
}
  STATE: data_type
     CONDITIONS: ENTER_STATE
     ACTIONS:
     {
        displayf ("%c%c" , a, b);
     }
```

When you lengthen the *chars* to *shorts*, however, they behave differently. The *unsigned char* is left-padded with zeroes. The *signed char*, having a leftmost bit equaling 1, is left-padded with ones. This left-padding with ones is called "sign extension."

A *char* is converted to a *short* automatically when a *%d*, *%u*, or *%x* conversion is applied to it, so the following example illustrates the difference between the conversion of *signed* and *unsigned* types:

```
{
  signed char a = -6;
  unsigned char b = -6;
}
  STATE: data_type
     CONDITIONS: ENTER_STATE
     ACTIONS:
     {
        displayf ("%x%x ", a, b);
     }
```

The variable *a* will be seen to extend to hex *fffa*, which is *fa* left-padded with eight ones. The unsigned variable *b* will have been extended by eight zeroes and will appear unchanged as *fa*.

If the *%x* conversion specifiers in the example above are replaced by *%d*, the resulting signed-decimal conversion will show *a* equaling -6, *b* equaling 250. The *signed char* will have survived the type-lengthening with its original negative value intact.

Because they can be lengthened without changing their values, signed variables should be used for any arithmetic operations. Other differences between signed and unsigned variables, not reflected in Table 59-1, are the following:

2. *Comparison.* If the leftmost bit of a signed variable is 1, then the variable has a negative value and the expression *variable > 0* is false. If the leftmost bit of an unsigned variable is 1, the variable is positive and *variable > 0* is true.

3. *Division and modulus.* If the leftmost bit of a signed variable is 1, the two's complement of the variable rather the stored value will be used in any division or modulus operation.

4. *Right shifting.* When a right-shift (>>) operator is used on a signed variable, a 1-bit is shifted in at the left. When the same operation is performed on an unsigned variable, a 0-bit is shifted in.

Table 59-1 shows the ranges of values that are produced by *displayf* and *printf* routines when the valid conversion specifiers—*%c*, *%d*, *%ld*, and so on—are applied to the vàrious *signed* and *unsigned* data types. Frequently it makes no difference whether a variable is declared as *signed* or *unsigned*. When a variable undergoes type conversion, however, as in the case of a *char* given a decimal or hex conversion, there is a significant difference.

Table 59-1

Data Types: Ranges of Values Displayed and Printed

type	char conversion	signed decimal conversion		unsigned decimal conversion		hex conversion	
	(%c)	short (%d)	long (%ld)	shor%u)	long (%lu)	short (%x)	long (%lx)
char[1]	␀ to ▨	0 to 255	–	0 to 255	–	0 to ff	–
signed char[1]	␀ to ▨	-128 to 127	–	0 to 127 and 65408 to 65535	–	0 to 7f and ff80 to ffff	–
unsigned char[1]	␀ to ▨	0 to 255	–	0 to 255	–	0 to ff	–
int	–	-32768 to 32767	–	0 to 65535	–	0 to ffff	–
signed int	–	-32768 to 32767	–	0 to 65535	–	0 to ffff	–
unsigned int	–	-32768 to 32767	–	0 to 65535	–	0 to ffff	–
short	–	-32768 to 32767	–	0 to 65535	–	0 to ffff	–
signed short	–	-32768 to 32767	–	0 to 65535	–	0 to ffff	–
unsigned short	–	-32768 to 32767	–	0 to 65535	–	0 to ffff	–
long	–	–	-2147483648 to 2147483647	–	0 to 4294967295	–	0 to ffffffff
signed long	–	–	-2147483648 to 2147483647	–	0 to 4294967295	–	0 to ffffffff
unsigned long	–	–	-2147483648 to 2147483647	–	0 to 4294967295	–	0 to ffffffff

[1] Through "integral promotion," *char* is converted automatically to *int* in a %*d*, %*u*, or %*x* conversion.

(C) Static Storage Class

A variable must be of the *static* storage class to pass its value into a *waitfor* statement. Declarations at the Program, Layer, or Test level (Level 1 in the source code diagram in Figure 52-4) are *static* even if they are not explicitly declared so. The same is true of a character array initialized by a string (see Section 59.7).

A variable that is initialized at the State level must be declared as *static* by the programmer if the initialized value is to be used inside a *waitfor*.

The following program will display a value of 8 on the prompt line when the operator presses the spacebar:

```
STATE: pass_initialized_value
{
  static int initialized = 8;
}
  CONDITIONS: KEYBOARD " "
  ACTIONS:
  {
     displayf ("%d ", initialized);
  }
```

If you removed the word *static* from the declaration, the initialized value would not be passed into the condition clause and the program would display 0 or a "garbage" number instead of 8.

59.6 Operator Precedence

In an expression with more than one operator, operations are prioritized according to the ranking of operator precedence in Table 59-2. The operator with the highest precedence is at the top of the table. Precedence decreases as you move down.

Consider this example:

```
STATE: precedence
{
  int a;
  a = 3 * 4 + 2;
  displayf ("%d", a);
}
```

Because multiplicative operators (*, /, and %) have higher precedence than additive operators (+ and –), the *3 * 4* operation is performed first. Then 2 is added to the product of 3 and 4, and finally the sum is assigned to the variable *a*. (Assignment operators have very low precedence.) The result of the program is that *a* is displayed as 14. Compare this example:

```
STATE: precedence
{
  int a;
  a = 3 * (4 + 2);
  displayf ("%d", a);
}
```

Table 59-2
Operator Precedence[1]

Operator	Type of Operator	Associativity
()	primary expression	left to right
[] . -> ++ --	postfix	left to right
++ -- sizeof & * + - ~ !	unary	right to left
(type)	cast	left to right
* / %	multiplicative	left to right
+ -	additive	left to right
<< >>	bitwise shift	left to right
< > <= >=	relational	left to right
== !=	equality	left to right
&	bitwise AND	left to right
^	bitwise exclusive OR	left to right
|	bitwise inclusive OR	left to right
&&	logical AND	left to right
||	logical OR	left to right
? :	conditional	right to left
= *= /= %= += -= <<= >>= &= ^= |=	assignment	right to left
,	comma	left to right

[1] Operators on the same line have the same precedence; rows are in order of decreasing precedence.

Here the additive operation is performed before the multiplicative, since the parentheses that denote a primary expression (see Table 59-2) have the highest precedence of all. The result of this program is that decimal 18 is displayed.

Given operations with the same precedence, left-to-right or right-to-left "associativity" (see the right column in Table 59-2) indicates which is performed first. This order of processing is significant for an expression such as *36 / 6 / 2*, where the associativity is left to right.

Associativity is very important in assignment operations, which are always interpreted in a right-to-left direction. Consider this example:

```
STATE: right_to_left_associativity
{
  int a = 4;
  int b = 1;
  a = b;
  displayf ("%d", a);
}
```

The result of this program is that 1 is displayed, not 4. Right-to-left associativity also explains why the following program does not compile.

```
STATE: right_to_left_associativity
{
   int a = 3;
   3 = a;
   displayf ("%d", a);
}
```

A constant never can have a value assigned to it, even if the value equals the constant.

59.7 Strings

A string is a sequence of characters enclosed in double quotes. This is an example of a string:

```
"hello"
```

A string is an expression of the type *pointer*, and may be used anyplace in the program that is appropriate for a pointer. For example, a pointer is appropriate as the argument of a *displays* routine:

```
displays ("hello");
```

The string in this statement does two things during compilation: it writes the character string "hello" in memory, and it *points* to the first character in the string. The string "hello" becomes a 4-byte address that you can examine by displaying it as a *long* hexadecimal:

```
displayf ("%lx", "hello");
```

(A) Using a String to Initialize an Array

Note that the pointer represented by "hello" in the examples above is not stored anywhere and therefore can be used only once. The string pointer "hello" could have been stored as a pointer to the first character in an array, as in this example:

```
char string_array [] = "hello";
displayf (string_array);
```

Stored in this manner, the pointer can be used repeatedly.

An array like *string_array* that has been "initialized" by a string shares many of the traits of standard arrays, but it has unique characteristics as well.

1. *Data type.* A string may only initialize an array whose elements are of the type *char*.
2. *Null termination.* A string is always terminated by a null character. This null terminator is appended by the compiler, not the programmer.

3. *Size.* All arrays must declare their size, in any of three ways. The programmer may declare the length inside of brackets, as in this example:

   ```
   char array [5];
   ```

   Or he may leave the brackets empty and provide a list of initializers, inside of curly braces, from which the compiler can determine the size of the array:

   ```
   char initializer_list_array [] = {'h', 'e', 'l', 0x6c, 'o'} ;
   ```

   The third method of indicating size is to leave the brackets empty and initialize the array with a string, as in our original example of a string initializer:

   ```
   char string_array [] = "hello";
   ```

   The compiler will add a terminating null-character to this string, and calculate an array size of six. To verify that the compiler counts one more character than the user has entered, you may try the following test. Note that the *sizeof* operator will return the length of any array:

   ```
   STATE: display_size_of_string
   {
     char string_array [] = "hello";
     int compiler_count = sizeof(string_array);
     displayf ("%d",compiler_count );
   }
   ```

4. *One-dimensional array.* Whereas arrays in general can be multidimensional, a string-initialized array always has one dimension.

(B) Valid Strings

1. *ASCII and control.* With a few exceptions, all ASCII characters, including control characters, are valid in a string. The exceptions are ␀, ␊, ", and \. These characters are liable to be misinterpreted by the compiler. Null (␀) and linefeed (␊) will be taken to indicate a new logical line in the program. Double-quote (") will be mistaken for the end of the string. Backslash (\) will be misinterpreted as the start of an escape sequence.

   If one of these characters is included in a string, the program may not compile. If not, you will be returned to the Protocol Spreadsheet. The following message will be displayed for nulls or linefeeds: *"Error 718: Newline inside string."* For quotation marks, the message is *"Unclosed AR "C" region."* Depending on their placement in the string, backslashes may or may not generate an error. Even when compilation succeeds, however, they will not be interpreted correctly.

Table 59-3
C String Non-Literals

Non-literal	Meaning	ASCII character	Hex character
\a	bell	${}^{B}{}_{L}$	${}^{0}{}_{7}$
\b	backspace	${}^{B}{}_{S}$	${}^{0}{}_{8}$
\f	form feed	${}^{F}{}_{F}$	${}^{0}{}_{C}$
\n	linefeed †	${}^{L}{}_{F}$	${}^{0}{}_{A}$
\r	carriage return	${}^{C}{}_{R}$	${}^{0}{}_{D}$
\t	horizontal tab	${}^{H}{}_{T}$	${}^{0}{}_{9}$
\v	vertical tab	${}^{V}{}_{T}$	${}^{0}{}_{B}$
\'	single quote	'	${}^{2}{}_{7}$
\"	double quote †	"	${}^{2}{}_{2}$
\\\\	backslash †	\\	${}^{5}{}_{C}$
\\###	octal representation	any ASCII character	${}^{0}{}_{0}$ – ${}^{F}{}_{F}$
\x###	hex representation	any ASCII character	${}^{0}{}_{0}$ – ${}^{F}{}_{F}$

† These characters require non-literal entries in INTERVIEW strings. The others may be entered as ASCII characters, non-literals, or hexadecimal characters.

2. *Non-literals.* Most characters in strings are interpreted literally. Each of the invalid characters listed above, therefore, needs a non-literal representation. Non-literals are preceded by a backslash. The compiler converts these non-literals to their one-byte numeric value.

   To include a null (or any ASCII) character in a string, use the octal or hexadecimal representation shown in Table 59-3. Hex and octal numbers take up to three digits, so use leading zeroes if necessary. Otherwise, a subsequent digit may be interpreted as part of the value. Suppose, for example, you want to create the string "${}^{N}{}_{U}$abc". You initialize an array as follows:

   ```
   char string[] = "\x0abc";
   ```

   The string will be stored as "+c" (hexadecimal characters ${}^{A}{}_{B}{}^{6}{}_{3}$). The correct declaration was *char string[] = "\x000abc"*. In octal, the null would be written *\000*.

   Please note that a string that has a null character somewhere other than at the end will be difficult to display or print completely. Display and print routines that take strings as input typically begin at the pointer position and continue until they encounter a terminating null. If, as in the last example, a null is encountered at the beginning of the string, execution of the routine will end before anything has been displayed or printed.

   Provide precision to the *%H* conversion specifier to override null termination of a string while displaying a string in hex: see Section 60.3(C).

3. *Constants.* Spreadsheet constants may be included in strings. An example of a spreadsheet constant is the fox message represented as ⟪FOX⟫. See Section 25 on Constants.

   The C translator expands constants both inside and outside of C regions before the code is preprocessed.

4. *Hexadecimal characters.* ASCII characters, including the control characters, may be entered in strings as hexadecimal characters via the HEX key. Hex representation is considered literal. That is, you may not enter ASCII characters which require non-literal representation in strings as hexadecimal characters. The sequence of characters comprising a non-literal may be entered as hexadecimal characters. Double backslash (\\), for example, may be entered as $^{5}{}_{c}{}^{5}{}_{c}$.

(C) String Routines

There are several C routines in the INTERVIEW that display or print strings. See Section 63 on "Print" and Section 60 on "Display Window and Trace" for detailed descriptions of the *prints*, *displays*, and *traces* routines, as well as other display and print routines that use the *%s* conversion specifier.

There is also a pair of routines, *index* and *rindex*, that search inside of strings for particular characters. These routines are defined (with examples) in Section 67.

59.8 Recommended Sources

The following sources provide accurate, in-depth information on C Programming Language:

1. *ANSI Document X3J11/86-098.* Proposed American National Standard for Information Systems—Programming Language C.

   NOTE: When approved, the number for the ANSI document will change to: *ANSI Standard X3.159-198X.*

2. Darnell, Peter A., and Margolis, Philip E. *Software Engineering in C.* New York: Springer-Verlag, 1988.

3. Harbison, Samuel P., and Steele, Guy L., Jr. *C: A Reference Manual.* 2d ed. Englewood Cliffs: Prentice-Hall 1987.

4. Kernighan, Brian W., and Ritchie, Dennis M. *The C Programming Language.* 2d ed. Englewood Cliffs: Prentice-Hall, 1988.

60 Variables

60.1 Creating or Accessing C Variables

Softkey-selectable programming "tokens" entered by the user on the Protocol Spreadsheet are translated automatically into C during the initial compiler phases after $\boxed{\text{RUN}}$ is pressed. (Then the C code in turn is compiled into object code.) The C variables used by the translator are documented throughout this volume.

C regions available to the user at every level of spreadsheet programming (see Section 56) provide direct access to these variables.

An example of a user-accessible variable is *keyboard_new_key*, used in the following program to sound an alarm whenever any ASCII-keyboard key is pressed.

```
{
   extern fast_event keyboard_new_key;
}
      STATE: anykey
         CONDITIONS:
            {
               keyboard_new_key
            }
         ACTIONS: ALARM
```

The C regions also allow the user to work with variables of his own creation.

Here is an example of a user-created variable named *minutes* that is used to count minutes elapsed since the beginning of Run mode. The C program displays this "counter" on the prompt line of the Run-mode screen.

```
{
   extern fast_event fevar_time_of_day;
   short minutes;
}
      STATE: run_mode_minutes
         CONDITIONS:
         {
            fevar_time_of_day
         }
         ACTIONS:
         {
            minutes++;
            pos_cursor (0,0);
            displayf ("Duration of run = %4d minutes", minutes);
         }
```

The first C region in the example "declares" the variables *fevar_time_of_day* and *minutes*. The first of these variables is an event variable that is built into the system software. All event variables in an active State-block are polled constantly. Once every minute, *fevar_time_of_day* returns true.

The second variable, *minutes*, is created by the program itself—that is, by the user. The declaration in effect creates the variable: it causes 16 bits in memory ("short" = 16 bits) to be dedicated to information stored under the name *minutes*.

The second C region in the example is placed inside the Actions block. The statement *minutes++* causes the value that is stored in the 16 bits dedicated to *minutes* to increment. The function *pos_cursor (0,0)* places the cursor in the leftmost column on the second line of the display screen (the Prompt line). The *displayf* function writes a text message to the display screen, beginning at the current cursor position. In the text message itself, "%" will be replaced by the current value of the variable *minutes*. "4" means that four columns on the screen will be dedicated to the value, and "d" means that the value will be expressed in a decimal number.

60.2 Declaring Variables

Declare your variables and routines in a C region, delimited by curly braces { and }, at the top of your program or at the top of a Constants, Layer, Test, State, or Actions block. Declare a variable preceded by its type descriptors and followed by a semicolon, as in these examples:

```
{
  extern fast_event keyboard_new_key;
  extern fast_event keyboard_new_any_key;
  extern fast_event fevar_time_of_day;
  short minutes;
}
```

A variable may have its scope limited to a particular Test, State, or Actions block. A variable also may be redeclared at different levels. (In software revision 5.00 or earlier, it may not be redeclared at the same level.) Given more than one valid declaration, the lower or *nearer* one applies.

The rules governing the placement of variable declarations are laid out in detail in Section 56.5(A).

(A) Naming Variables

1. *Legal names*. The first letter of a variable name may be either a letter or an underscore. Following characters may be letters, numbers, underscores, or dollar signs.

   Reserved words (indicated in boldface type in Appendix K) may not be used as variable names.

2. *Naming conventions.* Generally speaking, variables that begin with *dte_* or *dce_* are used by the software to test DTE and DCE conditions. Variables that begin *rcvd_* are used to test RECEIVE (or RCV) conditions. Variables that begin *m_* are used by the layer packages to construct the protocol traces.

(B) Modifiers

1. *Data type.* The data type for each variable precedes the variable name in the declaration. All standard data types except *float* are supported in the INTERVIEW 7000 Series. Standard data types and their sizes and ranges are given in Table 59-1.

2. *Preassigned modifiers.* When you declare a user-accessible external variable, be sure to use the modifiers which precede the data type for that variable as listed in variable tables throughout this volume.

60.3 Comparing a Variable to a Value

User-accessible and user-created variables may be tested as part of any standard C expression.

The following is an example of a user-invented variable called *anykey* that is declared with a default value of zero, incremented by the operator pressing any ASCII-keyboard key, and checked for a value of 3 by an *if* statement after each depression of a key. An alarm will sound on the third keystroke.

```
{
  extern fast_event keyboard_new_key;
  short anykey;
}
    STATE: press_key
      CONDITIONS:
      {
        keyboard_new_key
      }

      ACTIONS:
      {
        anykey++;
        if (anykey == 3) sound_alarm ();
      }
```

The next example uses a built-in, user-accessible variable called *crnt_time_of_day* and checks it for a particular value. This 16-bit variable stores the time of day in hours and minutes. The Condition in the program (the event variable *fevar_time_of_day*) is true once per minute. The Action each time the condition is true is to check *crnt_time_of_day* for a value of 1129. At 11:29 AM, an alarm will sound.

```
{
  extern fast_event fevar_time_of_day;
  extern volatile unsigned short crnt_time_of_day;
}
    STATE: alarm_clock
      CONDITIONS:
      {
        fevar_time_of_day
      }
      ACTIONS:
      {
        if (crnt_time_of_day == 1129) sound_alarm();
      }
```

60.4 Checking a Variable in a *Waitfor* Clause

Please note that the following variation on the preceding example does not produce the same result. *The alarm will never sound* if this version of the program is run:

```
{
  extern volatile unsigned short crnt_time_of_day;
}
    STATE: alarm_clock
      CONDITIONS:
      {
        crnt_time_of_day == 1129
      }
      ACTIONS:
      {
        sound_alarm();
      }
```

Note that the time-of-day condition that was lodged in an *if* statement in the previous example has now been placed in a Conditions block. Conditions blocks on the Protocol Spreadsheet are converted to *waitfor* clauses (see Section 56.3), not *if* statements, when the program is translated automatically into C coding.

Waitfor clauses work very differently from *if* statements and other conditional control structures in C.

(A) Event vs. Nonevent Variables

Two kinds of variables may be used inside of these *waitfor* clauses—event variables and nonevent variables. When a state is active, event variables in that state are checked regularly during routine polling by the CPU. When an event variable (such as *fevar_time_of_day*) is polled and returns a value of true, conditional statements containing nonevent variables (such as *crnt_time_of_day*) also are checked for truth or falsity. *In the absence of an event variable being polled and returning a value of true, a statement about a nonevent variable inside of a Conditions block* (waitfor *clause) never can be true.*

Since there is no event variable in the Conditions block (*waitfor* clause) above, the nonevent variable *crnt_time_of_day* is never even checked.

(B) Translation of Softkey Tokens Into Variables

You could have written the "alarm clock" program using only softkey entries, as follows:

```
STATE: alarm_clock
   CONDITIONS: TIME 1129
   ACTIONS: ALARM
```

In this case, the C translator will convert the Conditions block into a *waitfor* clause that uses the event variable *fevar_time_of_day* to check the nonevent variable *crnt_time_of_day* once a minute. Here is the translator's version of the Conditions and Actions blocks:

```
{
   waitfor
   {
      fevar_time_of_day && (crnt_time_of_day == 1129):
      {
         sound_alarm();
      }
   }
}
```

(C) Example of A Nonevent Condition "Waiting For" An Event

The next example illustrates the interplay of event variables and nonevent variables in a *waitfor* clause.

```
{
   extern fast_event keyboard_new_key;
   short anykey;
}
      STATE: press_key
         CONDITIONS:
         {
            keyboard_new_key
         }
         ACTIONS:
         {
            anykey++;
         }
         CONDITIONS:
         {
            anykey == 3
         }
         ACTIONS: ALARM
```

This program looks similar to a previous one in which the operator hit three keys and the alarm sounded. Here, however, the alarm does not sound until the fourth keystroke. The variable *anykey* begins the test at zero, and increments (*anykey++*) with every keystroke. But remember what a condition such as *anykey == 3* in a *waitfor* clause really means. It means that the condition will be true when the variable equals three and an event (such as a keystroke) occurs that causes the variable to be checked. On these terms, the condition is not satisfied until the fourth event.

(D) User-Created Event Variables

The user can create his own event variable simply by declaring a new variable with the modifiers *extern event*. Once the event variable has been declared, he can use the *signal* function to indicate that the event has occurred. Here is an example of an event variable called *check_number* that causes the nonevent variable *number* to be checked—and sounds the alarm when the value of *number* satisfies the condition.

```
{
   short number = 3;
   extern event check_number;
}
      STATE: user_created_event
      {
         signal (check_number);
      }
         CONDITIONS:
         {
            check_number && (number == 3)
         }
         ACTIONS: ALARM
```

(E) Rules and Cautions

To sum up the discussion of event and nonevent variables, here are a few rules of thumb:

1. *If* statements, *for* loops, *while* loops, and other conditional control structures may not be used in Conditions blocks (that is, in *waitfor* clauses). They may be used in State blocks, above (or in the absence of) Conditions blocks; and they may be used in Actions blocks.

   (Placing an *if* statement at the top of the State block, above any *waitfor* clauses, is how the translator converts ENTER_STATE softkey conditions into C.)

2. Event variables are designed for use in Conditions blocks (*waitfor* clauses) only. It makes no sense to use an event variable in an *if* statement, *while* loop, etc., since there is no possibility that the event will be true at the precise moment the statement is being processed.

3. A Conditions block (*waitfor* clause) that lacks an event variable can never come true.

One other word of caution about the importance of event variables: please note that the following program will not sound the alarm even if the operator presses a key while the time is 11:29 AM.

```
{
  extern fast_event keyboard_new_key;
  extern volatile unsigned short crnt_time_of_day;
}
    STATE: alarm_clock
      CONDITIONS:
      {
        keyboard_new_key && (crnt_time_of_day == 1129)
      }
      ACTIONS: ALARM
```

The reason this program doesn't "work" is that all variables begin Run mode at zero. Often a particular event variable must return true before a particular nonevent variable will be updated. The nonevent variable *crnt_time_of_day* is updated only when the event variable *fevar_time_of_day* is entered in the *waitfor* clause and returns true. In the example above, the operator pressing the key will cause *crnt_time_of_day* to be checked; but in the absence of *fevar_time_of_day*, the value of *crnt_time_of_day* remains always at zero.

60.5 Checking and Displaying Equivalent Values of a Variable

Variables may be checked and displayed as octal, decimal, hexadecimal, and ASCII-character values. Decimal comparison and display is the default.

(A) Checking Equivalent Values

To compare a variable to an octal value, precede the value with a zero (0). No prefix is necessary to make a decimal comparison. To compare a variable to a hexadecimal value, precede the value with 0x or 0X. To check whether a variable matches an ASCII character, enter the character in between single quotes.

The alarm will sound in the example below, since all of the values entered to the right of the equal signs are equivalent.

```
{
  char foxtrot = 'f';
}
    STATE: compare_equivalent_values
    {
      if ((foxtrot == 0146) && (foxtrot == 102) && (foxtrot == 0x66) && (foxtrot ==
      'f')) sound_alarm();
    }
```

Note that the data type *char* in the declaration simply means that the variable is composed of 8 bits. The designation *char* does not say anything about the comparison mode or the display mode. (Data types *short* and *int* = 16 bits; *long* = 32 bits.)

(B) Displaying Equivalent Values

Variables may be displayed in a variety of data formats via the *displayf* function. The full set of display conversions is given in Table 64-7. The program below generates a representative sample of display formats. When the program is run, the prompt line on the display screen will look like this: 152 106 6a 6A j $^{6}{}_{A}$.

```
{
   char juliet = 'j';
}
     STATE: display_equivalent_values
     {
        displayf ("%o  %d  %x  %X  %c  %#u ", juliet, juliet, juliet, juliet, juliet,
        juliet);
     }
```

60.6 Isolating Bits from a Variable Value

Some variables are bit-oriented. That is, one bit (or perhaps a small field of bits) may have significance that is independent of the surrounding bit values. The variable *current_eia_leads* (refer to Table 63-1), for example, uses 7 bits to store the on/off status of seven separate EIA leads, plus an eighth bit to store the status of any lead that is patched to the UA input jack (see Section 12.3). If you want to check this variable to determine the status of DTR (for example) you need to determine whether the bit that represents DTR (the fifth bit from the right or the fifth least significant bit in the variable) is set to 1 (DTR off) or zero (DTR on). How can you isolate this bit from the surrounding bits in order to determine its status?

The tool for isolating a bit in a C variable is the "care mask," a group of bits (usually expressed in hexadecimal) in which the bit(s) under scrutiny is set to 1 and all other bits to zero. The care mask for DTR is 0x10 (or 16 in decimal notation). The binary version, 00010000, shows that only the DTR bit is set to 1. When this care mask is *and*ed (via the "&" operator) with the variable *current_eia_leads*, only two results are possible, depending on whether the DTR bit in *current_eia_leads* is 1 or 0.

With DTR on, suppose that the combination of all lead statuses gives *current_eia_leads* a value of e6 in hex—11100110 in binary. The effect of *and*ing this variable with the care mask for DTR will be as follows:

```
  11100110
& 00010000
  00000000
```

Now turn DTR off, and the result of the *and*ing will be this:

```
  11110110
& 00010000
  00010000
```

The seven "don't care" zeroes in the care mask guarantee seven zero-bits in the result (because 0 & 1 = 0 and 0 & 0 = 0). So the result of the *and*ing must be either 0 if the DTR bit is 0 (on), or hex 10 (decimal 16, binary 00010000) if the DTR bit is 1 (off).

This C program will detect DTR on:

```
{
   extern fast_event fevar_eia_changed;
   extern const volatile unsigned short current_eia_leads;
}
      STATE: check_dtr_on
         CONDITIONS:
         {
            fevar_eia_changed
         }
         ACTIONS:
         {
            if ((current_eia_leads & 0x10) == 0) sound_alarm();
         }
```

If you try to run this program, make sure of the following:

1. The Front-End Buffer Setup menu should be configured to buffer control leads.

2. If you are not connected to a device that provides clock, the Line Setup menu should be configured to provide internal clock. EIA leads are clocked through the front-end buffer before they reach the program logic.

3. After the program enters Run mode, use a single-wire patch cord to connect the +12V output pin on the test-interface module to the DTR lead. The alarm should sound as soon as the patch is made.

A slightly different condition inside of the *if* statement will detect DTR off:

```
if ((current_eia_leads & 0x10) == 0x10) sound_alarm();
```

The DSR bit is the fourth least significant bit in the *current_eia_leads* variable, so the care mask for DSR is 0x08 (binary 00001000). The following *if* statement will detect DSR on:

```
if ((current_eia_leads & 0x08) == 0) sound_alarm();
```

This *if* statement will detect DTR on and DSR on:

```
if ((current_eia_leads & 0x18) == 0) sound_alarm();
```

This *if* statement will detect DTR off and DSR on:

```
if ((current_eia_leads & 0x18) == 0x10) sound_alarm();
```

The last condition simply means that you care (1=care) about DTR and DSR and you want DTR to be 1 (off) and DSR to be 0 (on).

60.7 Pointing to an Address

Some routines require an address as input. The *displays* (display-string) routine, for example, requires a CPU memory address as its argument. When executed, the routine will begin to display characters that it finds at the specified address and at subsequent addresses, one by one, until a null is encountered. A memory address is four bytes (32 bits) and is declared as a *long*.

```
{
   long any_cpu_address;
}
   STATE: display_string
   {
      displays (any_cpu_address);
   }
```

Many of the important addresses needed by the user and by the program can be found inside of interlayer ("IL") message buffers. When BOP-framed data is monitored, it is copied automatically into IL buffers. Each time a frame is buffered, a data primitive is created automatically and the event variable *m_lo_ph_prmtv* is signaled. The segment number of the IL buffer is recorded in the variable *m_lo_ph_il_buff*. This segment number can be converted into an address.

Here, for example, is a program that looks for a DTE data packet, converts *m_lo_ph_il_buff* into a four-byte address that points to the first data position, and displays the data contents of the packet.

```
{
   long first_data_address;
   extern volatile unsigned short m_lo_ph_il_buff;
}
LAYER: 3
   STATE: display_data
      CONDITIONS: DTE DATA
      ACTIONS:
      {
         first_data_address = ((long) m_lo_ph_il_buff << 16) + 37;
         displays (first_data_address);
      }
```

The IL buffer is illustrated in Section 66 of this manual, and the procedure for converting the buffer-segment number into a memory address is explained in detail in Section 66.1(C). Briefly, we have cast the segment number (a *short*, 16 bits) into a *long* and moved the number over to its high-order position in the CPU address, sixteen bits to the left. Then we added 37 to the number to bypass the header information for the buffer (32 bits) and the frame and packet headers (5 bits).

Each address in memory stores 8 bits, so the second byte in the data field of the data packet would be *first_data_address + 1*, the fourteenth byte would be *first_data_address + 13*, and so on.

60.8 Creating a Character Pointer

For most of the variables in a C program, the address is not important to the user or to the program. The user does not need to know the address in order to declare the variable, perform operations on it, and compare its value to other values. In general, addresses of variables are solely the concern of the compiler.

In the case of a routine such as *displays*, the address is what is important. The value that is stored at the address is not so important, since the routine will go to the address and begin displaying the data whatever the value (as long as the value is displayable).

There is another kind of variable for which both the address and the value stored at the address are important. These variables are called pointers. The user creates a pointer by typing an asterisk (*) just following the data type in a declaration, as in this example:

```
char * packet_type_ptr;
```

The variable *packet_type_ptr* is a four-byte memory address just as *first_data_address*, declared as a *long* in the previous example, was a four-byte address—even though *packet_type_ptr* is declared as a *char*. The data type *char* preceding the asterisk simply means that the amount of data pointed to is eight bits.

Once you use an asterisk to declare the variable a pointer, you can access the address directly as *packet_type_ptr* or you can access the value stored at that address as ** packet_type_ptr*. A *displays* routine would accept *packet_type_ptr* as input, while a *displayc* or *displayf* routine would expect ** packet_type_ptr*.

With the X.25 personality package loaded at Layers 2 and 3 (via the Layer Setup screen), the following program goes to the memory location pointed to by *packet_type_ptr* and checks its value to determine whether the packet in the buffer is a Clear request.

```
{
  extern volatile unsigned short m_lo_ph_il_buff;
  extern event dte_packet;
  char * packet_type_ptr;
}
  STATE: search_for_dte_clear
     CONDITIONS:
     {
        dte_packet
     }
     ACTIONS:
     {
        packet_type_ptr = (void *) (((long) m_lo_ph_il_buff << 16) + 36);
        if (*packet_type_ptr == 0x13) sound_alarm();
     }
```

The pointer *packet_type_ptr* is a *char*, but you could just as easily point to a *short* (16 bits) or a *long* (32 bits). If you increment an address, you get the next address, 8 bits farther in memory. If you increment a *char* pointer, you also get the next address. If you increment a *short* pointer, you add two increments to the memory address. In effect you move the pointer two places. If you increment a *long* pointer, you move the pointer by four addresses, 32 bits.

In the example above, the integer *m_lo_ph_il_buff* is cast as a pointer (*void* *) after it is cast as a *long*. This is to avoid a compiler error ("Warning 31: Illegal implicit integer-to-pointer conversion") when the new value of *m_lo_ph_il_buff* is assigned to *packet_type_ptr*.

60.9 Pointing with Subscripts

When it is preceded by an asterisk (*), the pointer *packet_type_ptr* returns the character value that it points to, as we have just seen. Another way to return this value is to omit the asterisk and add a subscript: *packet_type_ptr[0]*. This mechanism allows you to access an array of values without moving the pointer.

For example, the transmission header ("TH") in a FID2 SNA information field is six bytes long. If you establish a pointer to the first TH byte (TH0), you can use subscripts to access any other byte in the field without moving the pointer. The following program checks the values of two bytes in the TH field (corresponding to "DAF" and "OAF") before freezing the data display and sounding an alarm.

```
{
  extern volatile unsigned short m_lo_ph_il_buff;
  char * th;
}
LAYER: 2
  STATE: th_pointer
    CONDITIONS: DTE INFO
      ACTIONS:
      {
        th = (void *) (((long) m_lo_ph_il_buff << 16) + 34);
        if ((th[2] == 5) && (th[3] == 1))
        {
          ctl_capture_td (0x10);
          ctl_capture_rd (0x100);
          sound_alarm ();
        }
      }
```

60.10 Creating a String

Strings are used in INTERVIEW programming mainly for transmissions and for messages to the operator ("prompts"). In the following program, the compiler decodes the string "QWERTYUIOP" from ASCII to hex, stores it in memory as a series of contiguous values, adds a null to it, returns the address of the first character, "Q," and then assigns this address to the variable *keyrow*:

```
{
   long keyrow;
}
STATE: assign_string_address_to_variable
{
   keyrow = "QWERTYUIOP";
}
```

The variable *keyrow* now is the four-byte address of "Q" in the string. You can see this address for yourself by using either *"QWERTYUIOP"* or *keyrow* as the argument in a *displayf* routine:

```
displayf ("%lx ", "QWERTYUIOP");
```

or

```
displayf ("%lx ", keyrow);
```

Either version will display a CPU address (hex 04400000) on the second line of the Run-mode screen.

The string can be displayed in a simple *displays* routine, since that routine expects a four-byte address as input:

```
displays ("QWERTYUIOP");
```

or

```
displays (keyrow);
```

If you want to access individual characters in the string, declare a pointer:

```
char * keyrow = "QWERTYUIOP";
```

With a pointer you can display the entire string or a single character—the seventh character, "U," in this example:

```
displays (keyrow);
displayc (keyrow[6]);
```

Declaring the string an array has virtually the same effect as declaring it a pointer:

```
char keyrow [] = "QWERTYUIOP";
```

The name of the array still is the address of the first character in the string and so may be used in a *displays* routine; and individual characters still may be specified by a subscript:

```
displays (keyrow);
displayc (keyrow[6]);
```

The only difference is that the array name is a constant whose value is assigned in a declaration and cannot be changed, while the pointer is a variable and may be incremented, assigned a new value, and so forth, while the program is running.

60.11 Comparing Strings

A string comparison in C may be conducted as follows. First, create a pointer in the manner described in Section 60.8, or else simply declare one of the pointers to line data that is provided in the set of user-accessible variables. Example: *extern volatile unsigned char * m_packet_ptr.*

Next, create an array that represents the search string you will try to match against the line data. For example:

```
char search_string [] = "\xa";
```

Create a trigger to look for a line event (such as the event variable *dte_packet*) that will initialize the pointer.

```
{
   extern volatile unsigned char * m_packet_ptr;
   char search_string [] = { 0x10, 0x04, 0x0b };
   extern event dte_packet;
}
LAYER: 3
     STATE: match_packet_string
        CONDITIONS:
        {
           dte_packet
        }
```

Compare the pointer-value with the first element of the search string. If a match is found, increment the pointer and compare the new value to the second element of the search string; and so on. If a match is found for every element of the string, take an appropriate action.

```
ACTIONS:
{
   if (search_string [0] == * m_packet_ptr)
   {
      m_packet_ptr ++;
      if (search_string [1] == * m_packet_ptr)
      {
         m_packet_ptr ++;
         if (search_string [2] == * m_packet_ptr) sound_alarm ();
      }
   }
}
```

Here is the same Actions block, only this time the variable *element* replaces the numeral in the subscript to *search_string*, and the same variable is added as a subscript to *m_packet_ptr*. This coding may be modified easily for any length string. For a 9-byte string, for example, simply change the 3 in the *if* statement to 9.

```
ACTIONS:
{
   element = 0;
   while (search_string [element] == m_packet_ptr [element])
   {
      if (search_string[element++] == 3)
      {
         sound_alarm();
         break;
      }
   }
}
```

60.12 Accessing a Variable Inside of a Structure

A structure is a mechanism that makes repetitive declarations of similar variables unnecessary. For example, there are twelve variables associated with any given counter created in the program. One variable is the current value of the counter, one is the last sampled value, another is the highest sampled count, another the total of all the sampled values, another the number of samples taken, and so forth. If the user creates four counters via the spreadsheet softkeys, the C translator does not declare 48 separate variables (4 x 12). Instead the translator declares a structure for counters—called *counter_struct*—that declares each of the twelve variables once, as follows:

```
{
   struct counter_struct
   {
      unsigned long current;
      unsigned long last;
      unsigned long maximum;
      unsigned long minimum;
      unsigned short sample_count;
      unsigned long total_high;
      unsigned short total_low_low;
      unsigned short total_low_high;
      unsigned short out_of_range;
      unsigned short changed;
      unsigned long prev;
      unsigned long old;
   };
```

Then the translator declares each of the user's four counters as having the structure *counter_struct*:

```
struct counter_struct dte_good_bcc, dte_bad_bcc, dce_good_bcc, dce_bad_bcc;
```

In effect the translator has declared all 48 variables. Suppose the user wants to access one of these variables. He may wish to display the total value of a counter whose current value no longer is the total value (since the counter may have been sampled—and therefore cleared—several times). As long as the total is less than 65,536, the entire number will reside in the seventh variable in the *counter_struct* structure, *total_low_low*. If the counter in question is *dce_good_bcc*, he will access this "total" variable under the name *dce_good_bcc.total_low_low*.

Here is a sample trigger that displays this variable whenever the operator presses T:

```
STATE: display_total_dce_good_bcc
   CONDITIONS: KEYBOARD "Tt"
   ACTIONS:
      {
         displayf ("Total DCE good BCC's = %d", dce_good_bcc.total_low_low);
      }
```

Refer to Section 65.1 for more detail on the structure of counters.

60.13 Creating a Structure Pointer

We have just seen how a structure can be created to store and access data conveniently. A structure can also be used as a multibyte pointer that is superimposed on data that has been stored previously.

In our example we will declare the structure of an IL buffer and then point this structure at a newly received IL buffer.

The precise structure of an IL buffer is given in the following declaration. Note that there are 32 bytes devoted to header information and the remaining 4K bytes are available for data.

```
{
   struct il_buffer
   {
      unsigned short lock;
      unsigned short maintain_bits;
      unsigned short buffer_size;
      unsigned short transmit_tag;
      unsigned short receive_tag;
      unsigned long char_buff_frame_start;
      unsigned long char_buff_frame_end;
      unsigned short tick_count_high;
      unsigned short tick_count_mid;
      unsigned short tick_count_low;
      unsigned short available_space_offset;
      unsigned short bytes_remaining;
      unsigned long bcc_indicator;
      unsigned char data [4064];
   };
```

The next step is to create a pointer that has the structure of *il_buffer*. First, declare the structure of *il_buffer*, as indicated above. Then declare *buffer_ptr* as a structure-pointer, as follows:

```
struct il_buffer * buffer_ptr;
```

The next step is to wait for an INFO frame to be monitored. When the the frame data has been buffered and *m_lo_ph_il_buff* has been updated with the new buffer-segment number, assign the first address of this buffer to *buffer_ptr*.

```
buffer_ptr = (void *) ((long) m_lo_ph_il_buff << 16);
```

Now a structure has been created around the most recent upward-moving IL buffer. This means that rather than moving a pointer around in the IL buffer, you can access elements in the buffer directly. The *tick_count_low* variable, for example, would be called *buffer_ptr->tick_count_low*. (The -> operator is used in place of the dot operator in structure-pointers.)

The first element of the *data* string would be called *buffer_ptr ->data[0]*. Here is a program that displays on the prompt line the fifth data element (the packet-*type* byte) in the IL buffer for Info frames monitored on DTE.

```
{
  extern volatile unsigned short m_lo_ph_il_buff;
  struct il_buffer
  {
     unsigned short lock;
     unsigned short maintain_bits;
     unsigned short buffer_size;
     unsigned short transmit_tag;
     unsigned short receive_tag;
     unsigned long char_buff_frame_start;
     unsigned long char_buff_frame_end;
     unsigned short tick_count_high;
     unsigned short tick_count_mid;
     unsigned short tick_count_low;
     unsigned short available_space_offset;
     unsigned short bytes_remaining;
     unsigned long bcc_indicator;
     unsigned char data [4064];
  };
  struct il_buffer * buffer_ptr;
}
  LAYER: 2
     STATE: monitor_il_buffers
        CONDITIONS: DTE INFO
        ACTIONS:
        {
           buffer_ptr = (void *) ((long) m_lo_ph_il_buff <<16);
           pos_cursor (0,0);
           displayf ("%02x ", buffer_ptr->data[4]);
        }
```

61 Routines

This manual documents the C routines that are "external" to the C program—that is, defined elsewhere than in the program. Most of these routines are used by the C translator when it converts softkey-selectable programming "tokens"—most commonly those tokens that are appropriate to Actions blocks—entered by the user on the Protocol Spreadsheet. Some, like the Disk I/O routines, are associated with no spreadsheet conditions or actions and can be accessed only in C regions on the spreadsheet.

61.1 Declarations

In most of the examples in the manual, we have not bothered to declare routines since it is not necessary. In the absence of a declaration, the compiler assumes that the routine is external and that it returns an integer. In nearly all cases, this assumption works. In those rare cases when the routine returns another data type (the stats-display routine *get_68k_phys_addr*, for example, returns a *long*) it must be declared.

61.2 Arguments

An argument is an input that the user provides when he calls a routine. Arguments are placed inside of parentheses just following the routine name, as in this call to the *pos_cursor* routine: *pos_cursor (1,5);*

This routine requires two arguments in order to position the cursor in one of 1,088 possible character positions. The first argument selects one of the seventeen horizontal rows. The second argument selects one of the sixty-four vertical columns.

Many routines in the INTERVIEW library have arguments whose names end in the letters *ptr* or *pointer*. If you look at the synopsis for the *displays* routine, for example, you will see that the only argument is something called *string_ptr*. This is an address argument. The user enters a four-byte address as argument when he calls the displays routine, and the routine goes to this address and begins displaying data until a null (or other nondisplayable character) is encountered.

Pointers are four-byte addresses. The following call to the *displays* routine will go to the location of *m_packet_info_ptr* (the first byte of user data in a packet) and begin displaying data until a nondisplayable character is encountered:

```
displays (m_packet_info_ptr);
```

Array names also are four-byte addresses. The following example will display the characters in the array *string*:

```
char string [] = "QWERTY";
displays (string);
```

A string of characters declared inside of double-quotation marks is really a four-byte address that points to the first character in the string. In the function call *displays ("qwertyuiop")*, *"qwertyuiop"* qualifies as a string pointer and therefore satisfies the formal definition of the routine.

Many routines have no arguments and are called with empty parentheses:

```
sound_alarm();
```

Do not omit the parentheses. Without them, *sound_alarm* is a variable instead of a routine.

61.3 Returns

In addition to performing various operations, many routines include a *return* function that, at the end of the routine, stores a user-defined value in a memory location. As an example, we will look at an X.25 routine called *l3_window_full*.

The *l3_window_full* routine is declared automatically by the translator after the user has made a WINDOW FULL softkey entry. The synopsis for *l3_window_full* shows how it is declared:

```
extern unsigned char l3_window_full (path_number);
```

The routine is declared as a *char* because at the end of the routine, a return function will store a *char*-sized value (8 bits) in memory. If the packet window is full, the stored value will be nonzero. If the packet window is not full, the value will be zero.

The stored value is accessed any time you call the routine in your program. If you want to test for the window being full, you can enter this line of code:

```
if (l3_window_full(path_number) != 0) sound_alarm();
```

Here is a simpler coding for the same test:

```
if (l3_window_full(path_number)) sound_alarm();
```

This coding works for the same reason that *if (1) sound_alarm();* or *if (!0) sound_alarm();* will sound the alarm. Nonzero constants, variables, and expressions are true in C and cause statements to be executed inside of *if*, *while*, and other control constructions. Constants, variables, and expressions that equal zero are false and prevent statements in control structures from being executed.

If a routine is declared as a *short*, a *short* will be set aside in memory and any value returned by the routine (via a *return* function) will be stored there. If the routine is declared a *long*, a *long* will be reserved. If the routine is declared *void*, no space will be reserved in memory and a call to return a value will not be successful.

61.4 User-Defined Routines

The following coding will blank out the prompt line near the top of the INTERVIEW run-mode display.

```
pos_cursor(0,0);
displays ("                                        ");
```

If you code these two routines each time you display a user-prompt, you can always be sure that the prompt line will be blank and that each prompt will overwrite the previous prompt completely. The only problem is that the two routines are laborious to type in.

A better way is to declare a routine that executes the two "subroutines" automatically.

Declare a routine with its arguments inside parentheses and its body—the list of statements or subroutines that the routine is intended to perform—inside a pair of curly braces.

```
void blank_prompt_line()
{
   pos_cursor(0,0);
   displays ("                                        ");
}
```

Now you can blank out the line simply by typing this:

```
blank_prompt_line();
```

Suppose you wanted a routine that blanked the prompt line and generated a new prompt. The new prompt will be the argument for the routine:

```
void new_prompt (string_pointer)
char string_pointer [];
{
   pos_cursor(0,0);
   displays ("                                        ");
   pos_cursor(0,0);
   displays (string_pointer);
}
```

Now you can generate a prompt against a blank background with this simple routine:

```
new_prompt ("This prompt will overwrite any previous prompt");
```

NOTE: User routines may be declared and defined outside of the current spreadsheet program—in include files or linkable-object files. See Section 59.4.

61.5 Example Routines

We will provide three examples that will help illustrate how routines are created.

(A) Example Routine: Temporary Prompt

Here is a user-defined routine that blanks the prompt line, displays a new user-defined prompt, and then waits a user-defined interval before blanking the prompt line again. The routine is called *temporary_prompt*. The two inputs are 1) the new prompt, and 2) the number of seconds that you want the prompt to remain on the display.

The routine incorporates one external routine, *timeout_restart_action*, discussed in Section 72.3 of the section titled "Other Library Tools," and one internal routine, *blank_prompt_line*, discussed above.

```
{
  struct
  {
     unsigned long event_id;
     unsigned short event_id_uid;
  }
  timeout_prompt;
  void blank_prompt_line()
  {
     pos_cursor(0,0);
     displays ("                                            ");
  }
  void temporary_prompt (string_pointer, seconds)
  char string_pointer [];
  char seconds;
  {
     blank_prompt_line();
     pos_cursor(0,0);
     displays(string_pointer);
     timeout_restart_action (&timeout_prompt, seconds * 1000, blank_prompt_line);
  }
}
STATE: test_temporary_prompt
  CONDITIONS: KEYBOARD " "
  ACTIONS:
  {
     temporary_prompt("This prompt will self-destruct in 4 seconds.", 4);
  }
```

Note that the *blank_prompt_line* routine is embedded inside the *timeout_restart_action* routine, which in turn is embedded inside the *temporary_prompt* routine.

Note also:

The structure *timeout_prompt* is needed by the *timeout_restart_action* routine. The structure is explained in Table 72-1.

The two arguments in the *temporary_prompt* routine are declared outside the body of the routine (that is, outside of the curly braces). As a result, they are not redeclared each time the routine is called.

Timeout timers increment in milliseconds, so the user's *seconds* entry is multiplied by 1,000.

(B) Example Routine: Display Binary Value of Byte

The next sample routine takes a user-defined 8-bit value as input and expands it into a binary display of ASCII 1's and 0's. The routine, called *display_binary*, uses the & ("and") operator to isolate each bit and turn it into a "1" or "0" in an ASCII string called *binary_string*. See Section 60.6 for a discussion of the & operator.

The condition-and-action program that follows the declaration of *display_binary* uses the routine to expand the packet-type byte in each DCE packet.

```
{
 extern volatile unsigned char * m_packet_ptr;
 extern event dce_packet;
 char binary_string [8];
 void display_binary (hex_value)
 char hex_value;
 {
  if ((hex_value & 0x80) == 0) binary_string[0] = '0';
  else binary_string[0] = '1';
  if ((hex_value & 0x40) == 0) binary_string[1] = '0';
  else binary_string[1] = '1';
  if ((hex_value & 0x20) == 0) binary_string[2] = '0';
  else binary_string[2] = '1';
  if ((hex_value & 0x10) == 0) binary_string[3] = '0';
  else binary_string[3] = '1';
  if ((hex_value & 0x08) == 0) binary_string[4] = '0';
  else binary_string[4] = '1';
  if ((hex_value & 0x04) == 0) binary_string[5] = '0';
  else binary_string[5] = '1';
  if ((hex_value & 0x02) == 0) binary_string[6] = '0';
  else binary_string[6] = '1';
  if ((hex_value & 0x01) == 0) binary_string[7] = '0';
  else binary_string[7] = '1';
  displayf ("\n%s", binary_string);
 }
}
  STATE: binary
   CONDITIONS: { dce_packet }
   ACTIONS:
   {
    display_binary (m_packet_ptr[2]);
   }
```

(C) Example Routine: Compare String Against Line Data

Here is a routine called *strcmp* that matches a user-entered string to line data, beginning at a point in the line data that the user specifies. The arguments are the string itself and a pointer to the beginning of the line data.

When the user enters his string inside double quotes, the compiler writes the string into memory, appends a zero (null), and returns a pointer to the first character in the string. The *strcmp* routine uses this zero to determine when the match is complete.

If a complete match is found, the *return(1)* routine breaks out of the while loop, so the *return(0)* never is executed. A routine that returns 1 (or nonzero) inside of an *if* condition will make the condition true.

The sample program that uses the *strcmp* routine looks on the DCE side for a data packet with a user-data field that begins "␍␊PASSWORD." This string occurs on the "HDLC/X.25 Data Sample" diskette, DSK-951-007-1, shipped with your INTERVIEW. Be sure to load in the Layer 2 and Layer 3 X.25 packages if you try out this program. The Layer 3 package will provide you with your line-data pointer (*m_packet_info_ptr*).

```
{
 extern volatile unsigned char *m_packet_info_ptr;
 int element;
 int strcmp (user_string_ptr, line_data_ptr)
 char user_string_ptr [];
 char * line_data_ptr;
 {
  element = 0;
  while (user_string_ptr[element] == line_data_ptr[element])
     {
      if (user_string_ptr[++element] == 0)
      return (1);
     }
  return(0);
 }
}
LAYER: 3
   STATE: match_user_data_field
     CONDITIONS: DCE DATA
     ACTIONS:
     {
      if (strcmp("\x0d\x0aPASSWORD", m_packet_info_ptr))
      sound_alarm();
     }
```

62 Monitor/Transmit Line Data

The external variables and routines in this section are available for use by the programmer to monitor and transmit data. Their use on the Protocol Spreadsheet is not limited to any particular layer, though normally they belong at Layer 1.

The variables and routines approximate Layer 1 spreadsheet-generated conditions and actions. Refer to Section 31 for more detailed explanations of the purposes of specific conditions and actions. Sometimes the name of the variable or routine is sufficient for identifying its related spreadsheet token. When this is not the case, the information is provided below.

62.1 Structures

Use the structure *xmit_list*, shown in Table 62-1, when transmitting line data via the *l1_transmit* routine. Refer to *l1_transmit* in Section 62.3(B) for an example of how to use this structure.

Table 62-1
Transmit Structures

Type	Variable	Value (hex/decimal)	Meaning
Structure Name:	xmit_list		Structure of a transmit list for *l1_transmit* routine. Declared as type *struct*. Reference member variables of the structure as follows: *xmit_list.string_length*.
unsigned char *	string		pointer to the location of the transmit string—the transmit string is declared separately
unsigned short	string_length	*0-ffff/0-65535*	length of the transmit string

62.2 Variables

(A) Monitoring Events

1. *Emulate or monitor mode.* Layer 1 events include characters received, good or bad BCC's, aborts, parity errors, and framing errors. All event variables in Table 62-2 containing a *_td* or *_rd* suffix are valid in either emulate or monitor mode. These event variables are *fevar_rcvd_char_rd, fevar_rcvd_char_td, fevar_gd_bcc_rd, fevar_gd_bcc_td, fevar_bd_bcc_rd, fevar_bd_bcc_td, fevar_abort_rd, fevar_abort_td, fevar_parity_rd, fevar_parity_td, fevar_frm_error_rd, fevar_frm_error_td,* and *fevar_rcv_buffer_full.* The variable *fevar_frm_error_rd,* for example, equates to DCE FRAMING_ERROR (or RECEIVE FRAMING_ERROR when you are emulating DTE).

   You can use both *td* and *rd* variables relating to the same event in one conditions block. Suppose you want count all bad BCC's, from either side of the line. Enter the following CONDITIONS/ACTIONS block:

```
CONDITIONS:
{
 fevar_bd_bcc_td || fevar_bd_bcc_rd
}
ACTIONS: COUNTER bad_bcc INC
```

   Using spreadsheet tokens, the same test needs two CONDITIONS/ACTIONS blocks:

```
CONDITIONS: DTE BAD_BCC
ACTIONS: COUNTER bad_bcc INC
CONDITIONS: DCE BAD_BCC
ACTIONS: COUNTER bad_bcc INC
```

   Use *fevar_rcv_buffer_full* and its associated status variable, *rcv_buffer_full,* to monitor the status of the character buffer. The moment the buffer is full, *fevar_rcv_buffer_full* comes true and the value of *rcv_buffer_full* transitions from zero to a non-zero value. Then, new data begins to overwrite the old data. The softkey equivalent of *fevar_rcv_buffer_full* is the layer-independent condition BUFFER_FULL when it appears alone in a conditions block. When BUFFER_FULL is combined with another condition, in most cases the other condition will supply the event variable and only the status test will be used. See Section 30 for a discussion of this and other layer-independent conditions and actions.

Table 62-2
Monitor/Transmit Variables

Type	Variable	Value (hex/decimal)	Meaning
extern fast_event	fevar_rcvd_char_rd		True for each character received on RD. Line Setup configured for emulate or monitor mode.
extern fast_event	fevar_rcvd_char_td		True for each character received on TD. Line Setup configured for emulate or monitor mode.
extern fast_event	fevar_gd_bcc_rd		True when a good BCC is calculated for an RD block or frame. Line Setup configured for emulate or monitor mode.
extern fast_event	fevar_gd_bcc_td		True when a good BCC is calculated for a TD block or frame. Line Setup configured for emulate or monitor mode.
extern fast_event	fevar_bd_bcc_rd		True when a bad BCC is calculated for an RD block or frame. Line Setup configured for emulate or monitor mode.
extern fast_event	fevar_bd_bcc_td		True when a bad BCC is calculated for a TD block or frame. Line Setup configured for emulate or monitor mode.
extern fast_event	fevar_abort_rd		True when an abort is detected in an RD frame. Line Setup configured for emulate or monitor mode.
extern fast_event	fevar_abort_td		True when an abort is detected in a TD frame. Line Setup configured for emulate or monitor mode.
extern fast_event	fevar_parity_rd		True when a parity error is detected for an RD byte. Line Setup configured for emulate or monitor mode.
extern fast_event	fevar_parity_td		True when a parity error is detected for a TD byte. Line Setup configured for emulate or monitor mode.
extern fast_event	fevar_frm_error_rd		True when an async framing error is detected for an RD byte. Line Setup configured for emulate or monitor mode.
extern fast_event	fevar_frm_error_td		True when an async framing error is detected for a TD byte. Line Setup configured for emulate or monitor mode.

Table 62-2 (continued)

Type	Variable	Value (hex/decimal)	Meaning
extern fast_event	fevar_xmit_cmplt		True when the INTERVIEW puts a transmission out onto the link. Line Setup configured for emulate mode only.
extern fast_event	fevar_rcv_buffer_full		Returns true at the moment the character buffer fills with data and will begin to overwrite existing data. Line Setup configured for emulate or monitor mode.
extern volatile unsigned short	rcv_buffer_full	0	not full
		1	full
			Line Setup configured for emulate or monitor mode.
extern unsigned short	rcvd_char_td		Most recent TD character is stored in this variable. Line Setup configured for emulate or monitor mode.
		0-ff/0-255	data character (lower byte in 16-bit data word in data buffer)
		100/256	good or bad BCC
		101/257	flag
		102/258	sync
		103/259	abort
extern unsigned short	rcvd_char_rd		Most recent RD character is stored in this variable. Line Setup configured for emulate or monitor mode.
		0-ff/0-255	data character (lower byte in 16-bit data word in data buffer)
		100/256	good or bad BCC
		101/257	flag
		102/258	sync
		103/259	abort
extern unsigned char	td_modifier		Most recent modifier byte for a TD data character. This is the upper byte in the 16-bit data word reserved for each data character in the data buffer. Line Setup configured for emulate or monitor mode.
		1	data—initial value (always included in value of *td_modifier*)
		2	alternate code set
		4	underline (rd character)
		8	reverse image
		10/16	hexadecimal
		20/32	low intensity
		40/64	blink
		80/128	strike-thru (parity error)

Table 62-2 (continued)

Type	Variable	Value (hex/decimal)	Meaning
extern unsigned char	rd_modifier		Most recent modifier byte for an RD data character. This is the upper byte in the 16-bit data word reserved for each data character in the data buffer. Line Setup configured for emulate or monitor mode.
		1	data (always included in value of *rd_modifier*)
		2	alternate code set
		4	underline (rd character)—initial value of *rd_modifier*
		8	reverse image
		10/16	hexadecimal
		20/32	low intensity
		40/64	blink
		80/128	strike-thru (parity error)

2. *Emulate mode only.* One variable is valid in emulate mode only, since it monitors an emulate action. "SENDing" a transmission means queuing a transmission to send. The layer protocol (the RTS-CTS handshake, for example, at Layer 1) may delay the actual transmission. The fast-event variable *fevar_xmit_cmplt* will not come true until the transmission actually has been sent. Use this condition to start accurate response-time measurements.

   If you try to use *fevar_xmit_cmplt* in monitor mode, you will be returned to the main program menu. When you go to the Protocol Spreadsheet and search for errors, the following message will be displayed: "*Error 140: Unresolved reference fevar_xmit_cmplt.*"

(B) Status Variables

Status variables are those in Table 62-2 that do not include *event* in the Type column. Their associated event variables guarantee that they are updated and tested.

1. *Distinguishing character types.* Suppose you're monitoring the DCE side of the link. Every time a character is detected, the event *fevar_rcvd_char_rd* comes true, regardless of whether or not the character will be stored in the character buffer. Not all characters are "data" characters. A character also may be a flag or the second byte in a block-check, for example. *fevar_rcvd_char_rd* (or *fevar_rcvd_char_td*) does not distinguish character types.

   Character type is stored in the high byte of *rcvd_char_rd* or *rcvd_char_td.* For data characters, the high byte is zero. The low byte contains the actual value of the character.

For a "non-data" character, hereafter referenced as a special symbol, the high byte of *rcvd_char_rd* is a non-zero value. The low byte specifies a special symbol to be displayed on the data screen, overwriting or replacing the character. The special symbols are 🅂 (sync), 🄶 (good BCC), 🅱 (bad BCC), 🅰 (abort), and 🄵 (flag). See Table 62-2.

Notice on Table 62-2 that the value for good BCC and bad BCC is the same. Use *fevar_gd_bcc_rd* and *fevar_bd_bcc_rd* event variables to distinguish between good and bad BCC's (or data BCC's in DDCMP). Likewise, use *fevar_gd_bcc2_rd* and *fevar_bd_bcc2_rd* to differentiate between good and bad header BCC's in DDCMP. Refer to Section 78 for DDCMP variables.

Aborts are not automatically reflected in *rcvd_char_rd* and *rcvd_char_td.* When seven consecutive 1-bits are received in 7E-framed protocols, the controller chip generates an interrupt. The bits, however, are not stored in memory. In this case, use *fevar_abort_rd* or *fevar_abort_td* to detect the interrupt. When this event variable transitions to true, it updates *rcvd_char_rd* (or *rcvd_char_td*) to indicate an abort.

Use *rcvd_char_td* and *rcvd_char_rd* to monitor received characters, independent of whether or not they will be buffered. The following condition detects RD data characters only:

```
CONDITIONS:
{
 fevar_rcvd_char_rd && (!(rcvd_char_rd & 0x100))
}
```

2. *Attributes*. Data characters and special symbols in the character buffer are available for normal or enhanced display on the data display-screen. Access the data display by pressing DATA on the first rack of Run-mode softkeys, or by selecting it as the initial Run-mode display on the Display Setup menu.

   The current attributes for RD data are stored in *rd_modifier*. Table 62-2 shows how the various attributes are coded. The initial value of *rd_modifier* is always five. This value means that the character is data (1) on the RD (4) side. RD data is always underlined. TD data is never underlined. The initial value of *td_modifier*, therefore, is one.

   You may change some attributes by using spreadsheet tokens (or their equivalent C routines). The Layer 1 ENHANCE action allows you to control reverse-image, blink, hexadecimal, and low intensity enhancements. This action also updates *rd_modifier, td_modifier*, or both.

   When an RD data character is written to the character buffer, the value of *rd_modifier* is written to the high byte of a two-byte data event-word. The data character, found in *rcvd_char_rd*, is written to the low byte. See Section 62.3(C) on the format of character-buffer event words.

NOTE: The attributes in *rd_modifier* and *td_modifier* do not apply to special symbols. *rd_modifier* and *td_modifier* always reflect the attributes last assigned to data. Underlining applied to (RD) special symbols on the data display-screen comes from a bit in the special receive-event word. See Table 62-3.

62.3 Routines

Unless noted otherwise, the routines discussed below apply when the Line Setup menu shows either emulate or monitor mode.

(A) Controlling Data Display

ctl_enhance_td

Synopsis

```
extern void ctl_enhance_td(enhance_type_status);
unsigned short enhance_type_status;
```

Description

This routine turns various enhancements of the data display on and off on the DTE side. It also updates the variable *td_modifier*. The softkey equivalent of this routine is the ENHANCE DTE action on the Protocol Spreadsheet.

Inputs

There is one two-byte parameter. The high byte identifies the type of enhancement to be controlled: blink (40), low intensity (20), hexadecimal representation (10), and reverse image (08). The low-order byte indicates the status of the enhancement. To indicate a given enhancement is on, the second byte has the same value as the first. If the enhancement is to be turned off, the value of the second byte is zero. For example, if you want to turn blink on, the parameter value is 0x4040. To turn blink off, it is 0x4000.

Multiple enhancements can be controlled with one action by using hexadecimal addition of the parameters, as in the example for *ctl_enhance_rd*.

Example

Assume X.25 protocol for this example. You want to enhance the packet type byte on the DTE side with a blinking, reverse image.

```
LAYER: 1
     STATE: enhance_packet_type
        CONDITIONS:  DTE STRING "🄵🄵((XXXXXXX0))⊠⊠"
        ACTIONS:
        {
         ctl_enhance_td(0x4040);
         ctl_enhance_td(0x0808);
        }
```

CONDITIONS: DTE STRING "$\boxed{F}\boxed{F}$((XXXXXXX0))$\boxed{X}\boxed{X}\boxed{X}$"
ACTIONS:
{
ctl_enhance_td(0x4000);
ctl_enhance_td(0x0800);
}

ctl_enhance_rd

Synopsis

extern void ctl_enhance_rd(enhance_type_status);
unsigned short enhance_type_status;

Description

This routine turns various enhancements of the data display on and off on the DCE side. It also updates the variable *rd_modifier*. The softkey equivalent of this routine is the ENHANCE DCE action on the Protocol Spreadsheet.

Inputs

See *ctl_enhance_td*.

Example

Assume X.25 protocol for this example. You want to enhance the packet type byte on the DCE side with a blinking, reverse image.

LAYER: 1
STATE: enhance_packet_type
CONDITIONS: DCE STRING "$\boxed{F}\boxed{F}$((XXXXXXX0))$\boxed{X}\boxed{X}$"
ACTIONS:
{
ctl_enhance_rd(0x4848);
}
CONDITIONS: DCE STRING "$\boxed{F}\boxed{F}$((XXXXXXX0))$\boxed{X}\boxed{X}\boxed{X}$"
ACTIONS:
{
ctl_enhance_rd(0x4800);
}

ctl_capture_td

Synopsis

extern void ctl_capture_td(status);
unsigned short status;

Description

This routine turns on and off the presentation of DTE data to the screen—that is, it stops or "freezes" the display—and capture of data to the screen buffer (character RAM). Unlike the Manual Freeze mode initiated by the $\boxed{\text{FREEZE}}$ key, however, the "capture off" action does not allow you to scroll through the buffer while the test continues. The softkey equivalent of this routine is the CAPTURE DTE action on the Protocol Spreadsheet.

Inputs

The only parameter is the status of capture, on (0x00) or off (0x10). Turning capture off freezes the display.

Example

Assume X.25 protocol for this example. You want to turn capture off as soon as the cause byte is displayed in a Clear packet on the DTE side. Capture will be resumed when the spacebar is pressed.

```
LAYER: 1
     STATE: find_cause
        CONDITIONS: DTE STRING "FF((XXXXXXX0))☒☒',☒"
        ACTIONS:
        {
         ctl_capture_td(0x10);
        }
        CONDITIONS: KEYBOARD " "
        ACTIONS:
        {
         ctl_capture_td(0x00);
        }
```

ctl_capture_rd

Synopsis

extern void ctl_capture_rd(status);
unsigned short status;

Description

This routine turns on and off the presentation of DCE data to the screen—that is, it stops or "freezes" the display—and capture of data to the screen buffer (character RAM). Unlike the Manual Freeze mode initiated by the FREEZE key, however, the "capture off" action does not allow you to scroll through the buffer while the test continues. The softkey equivalent of this routine is the CAPTURE DCE action on the Protocol Spreadsheet.

Inputs

The only parameter is the status of capture, on (0x00) or off (0x100). Turning capture off freezes the display.

Example

Assume X.25 protocol for this example. You want to turn capture off as soon as the cause byte is displayed in a Clear packet on the DCE side. Capture will be resumed when the spacebar is pressed.

```
LAYER: 1
    STATE: find_cause
        CONDITIONS: DCE STRING "🄵🄵((XXXXXXX0))⊠⊠13⊠"
        ACTIONS:
        {
         ctl_capture_rd(0x100);
        }
        CONDITIONS: KEYBOARD " "
        ACTIONS:
        {
         ctl_capture_rd(0x00);
        }
```

outsync_action

Synopsis

```
extern void outsync_action(side);
unsigned short side;
```

Description

The *outsync_action* routine applies to synchronous format only. This routine sends one of the receivers (TD or RD) out of sync and initiates a search for sync. The softkey equivalent of this routine is the (PROTOCL) OUT_SYN action on the Protocol Spreadsheet.

Inputs

The only parameter identifies which side of the line is to go out of sync, 0 for the DTE side, 1 for the DCE side.

Example

To display DTE protocol information only, initiate sync each time a start-of-text character is found. The results of this routine are similar to turning capture off and on, but here the display does not have to be turned on again. It resumes automatically with sync.

```
LAYER: 1
    STATE: go_out_of_sync
        CONDITIONS: DTE STRING "␂ "
        ACTIONS:
        {
         outsync_action(0);
        }
```

disable_dce

Synopsis

extern void disable_dce();

Description

The *disable_dce* routine applies only to synchronous format in emulate mode. This routine completely disables the monitoring of the DCE side of the line. Once this routine has been executed, the DCE side of the line cannot be monitored until Run mode has been exited.

This reduction (by half) in the receive load enables the INTERVIEW to achieve better speeds for user-implemented BERT.

Example

For this example, configure the Line Setup menu with the following selections: **Mode:** EMULATE DCE**, Source:** LINE**, Format:** SYNC**, Sync Char:** $^{S}_{Y}$$^{S}_{Y}$**, Outsync:** ON**, Display Idle:** ON**, Xmit Idle Char:** °o**, Clock Source:** INTERNAL**.**

By disabling the receipt of DCE data, the following program runs at speeds higher than those possible when the INTERVIEW must process data from both sides of the line.

```
LAYER: 1
    STATE: look_for_errors
       CONDITIONS: ENTER_STATE
       ACTIONS:
       {
        disable_dce();
       }
       CONDITIONS: DTE ONE_OF "[illegible]"
       ACTIONS: COUNTER error INC
       CONDITIONS: KEYBOARD " "
       ACTIONS: SEND "((FOX))" NO_BCC
```

disable_dte

Synopsis

extern void disable_dte();

Description

The *disable_dte* routine applies only to synchronous format in emulate mode. This routine completely disables the monitoring of the DTE side of the line. Once this routine has been executed, the DTE side of the line cannot be monitored until Run mode has been exited.

This reduction (by half) in the receive load enables the INTERVIEW to achieve better speeds for user-implemented BERT.

Example

For this example, configure the Line Setup menu with the following selections: **Mode:** EMULATE DCE **or** EMULATE DTE**, Source:** LINE**, Format:** SYNC**, Sync Char:** $^{5}{}_{S}{}^{5}{}_{S}$**, Outsync:** OFF**, Display Idle:** ON**, Xmit Idle Char:** $^{5}{}_{S}$**, Clock Source:** INTERNAL .

By disabling the receipt of DTE data, the following program runs at speeds higher than those possible when the INTERVIEW must process data from both sides of the line.

```
LAYER: 1
     STATE: look_for_errors
        CONDITIONS: ENTER_STATE
        ACTIONS:
        {
         disable_dte();
        }
        CONDITIONS: DCE ONE_OF "[illegible]"
        ACTIONS: COUNTER error INC
        CONDITIONS: KEYBOARD " "
        ACTIONS: SEND "((FOX))" NO_BCC
```

(B) Transmitting

Use the following routines in emulate mode only. If you try to call one of these routines in monitor mode, you will be returned to the main program menu. When you go to the Protocol Spreadsheet and search for errors, a message like the following will be displayed: "*Error 140: Unresolved reference ll_il_transmit.*"

l1_transmit

Synopsis

```
extern void l1_transmit(count, struct_send_string_ptr, xmit_tag);
unsigned short count;
struct xmit_list
   {
    unsigned char * string_ptr;
    unsigned short string_length;
   };
struct xmit_list * struct_send_string_ptr;
unsigned short xmit_tag;
```

Description

The *l1_transmit* routine sends a specified string with a user-determined BCC.

Inputs

The first parameter is the number of strings to be sent.

The second parameter is a pointer to a structure which in turn identifies the location and length of each string.

The third parameter is a transmit tag which includes a BCC in bits 0–2: good (001), bad (010), or abort (011). Bits 3–7 are reserved for future use. Integers may be used to indicate the value of the transmit tag: good (1), bad (2), and abort (3).

Example

Assume you want to send a fox message at Layer 1 inside of an X.25 data packet with a good block check. You might have 2 strings, one with the Layers 2 and 3 header information, and one with the fox message. You would send these strings as follows:

```
{
 unsigned char headers [] = {0x01, 0x00, 0x10, 0x04, 0x00};
 unsigned char message [] = "((FOX))";
 struct xmit_list
  {
   unsigned char * string;
   unsigned short string_length;
  };
 struct xmit_list send_string [] = {&headers[0], 5, &message[0], sizeof(message) - 1};
}
LAYER: 1
    STATE: send_message
       CONDITIONS: KEYBOARD " "
       ACTIONS:
       {
        l1_transmit(2, &send_string[0], 1);
       }
```

l1_il_transmit

Synopsis

```
extern void l1_il_transmit(il_buffer_number, relay_baton, data_start_offset, transmit_tag);
unsigned short il_buffer_number;
unsigned short relay_baton;
unsigned short data_start_offset;
unsigned short transmit_tag;
```

Description

This routine sends a designated interlayer message buffer out onto the line.

Inputs

The first parameter is the interlayer message buffer number.

The second parameter is the maintain bit used to hold the buffer while the send operation is performed at Layer 1.

The third parameter is the offset from the beginning of the buffer to the service data unit (SDU).

The fourth parameter is a transmit tag which includes a BCC in bits 0-2: good (001), bad (010), or abort (011). Bits 3-7 are reserved for future use. Integers may be used to indicate the value of the transmit tag: good (1), bad (2), and abort (3).

Example

Send the same text as in the example for *l1_transmit*. The softkey equivalent of this routine is the SEND action on the Protocol Spreadsheet. Refer to Section 66.3(A) for a description of the *_get_il_msg_buff*, *_start_il_buff_list*, and *_insert_il_buff_list_cnt* routines.

```
{
 unsigned short il_buffer_number;
 unsigned short relay_baton;
 unsigned short data_start_offset;
 unsigned char message [] = "01\x0001004\x000((FOX))";
}
LAYER: 1
      STATE: send_message
         CONDITIONS: KEYBOARD " "
         ACTIONS:
         {
          _get_il_msg_buff(&il_buffer_number, &relay_baton);
          _start_il_buff_list(il_buffer_number, &data_start_offset);
          _insert_il_buff_list_cnt(il_buffer_number, data_start_offset, &message[0],
               (sizeof(message) - 1));
          l1_il_transmit(il_buffer_number, relay_baton, data_start_offset, 1);
         }
```

idle_action

Synopsis

```
extern void idle_action(character);
unsigned char character;
```

Description

Only for format SYNC, the *idle_action* routine allows you to change the idle-line condition applied by the INTERVIEW. The softkey equivalent of this routine is the (PROTOCL) IDLE_LN action on the Protocol Spreadsheet.

Inputs

The only parameter is a character or numeric value representing the idle character.

Example

X.21 or X.21BIS idles different characters in various states, ␌, ␀, +, for example. To signal a change in protocol state, you might change the idle character to +:

```
LAYER: 1
     STATE: change_idle_character
        CONDITIONS: KEYBOARD " "
        ACTIONS:
        {
         idle_action('+');
        }
```

set_tcr_b

Synopsis

```
extern void set_tcr_b (tcr_register_mask, tcr_register_value);
unsigned char tcr_register_mask;
unsigned char tcr_register_value;
```

Description

This routine clamps the transmit line to 0 (space) or 1 (mark), or unclamps it so that *transmit* routines may be executed. In X.21, steady zero will signal a clear request/indication or a clear confirm, while steady 1 will indicate one of the call-ready or call-setup states. In other contexts, the routine simply initiates and terminates a *break*.

Inputs

The first parameter is the mask that is *and*ed with the current TCR register to turn the current values of bits 3 and 4 (counting 1-8 from the right) to zero. This mask is always 0xf3.

The second parameter contains the new values of bits 3 and 4 that will be written to the register. The three available parameters are 0x08 to clamp the line to zero, 0x0c to clamp the line to 1, and 0x04 to unclamp the line and permit data transmissions.

Example

This program will generate a 250-millisecond break when the operator presses the [BRK] key.

```
{
 extern fast_event keyboard_new_any_key;
 extern volatile unsigned short keyboard_any_key;
}
```

```
STATE: generate_break
  CONDITIONS:
  {
   keyboard_new_any_key && (keyboard_any_key == 0x1e3)
  }
  ACTIONS: TIMEOUT break RESTART 0.250
  {
   set_tcr_b (0xf3, 0x08);
  }
  CONDITIONS: TIMEOUT break
  ACTIONS:
  {
   set_tcr_b (0xf3, 0x04);
  }
```

(C) Writing to Character RAM

For the sake of speed, the 64-Kbyte character buffer uses a shorter data word than the 32-bit word in the Display Window and traces. Refer to Table 64-4. A sixteen-bit event word is reserved for each character in the 64-Kbyte character buffer.

Table 62-3 shows the format of event words. Two kinds of event word should be distinguished: data and special receive.

1. *Data Event-Words.* Data event-words may contain enhancement attributes in the high byte. Whereas attributes comprise 24 bits of a *long* in the Display Window and the traces, in the character buffer they are contained in only 8 bits. Data words in the character buffer, therefore, include a less flexible set of attributes. Color attributes, for example, are not directly available in words written to the character buffer. See Section 17, Color Display, for an explanation of how reverse, blink, and low enhancements in the character buffer may be mapped to colors in the RGB output. Table 62-3 lists the available attributes.

   The character is located in the low 8 bits. Its value can range from hexadecimal 0 through FF.

2. *Special-Receive Words.* The high byte in special-receive words determines the symbol (from the special graphic character font) that will overlay the character contained in the low byte. The symbols that may be written to the character buffer are good BCC's, bad BCC's, aborts, flags, and sync. One bit, the td/rd indicator, controls on which side the symbol will be displayed. Symbols on the RD side are underlined, as all RD data is. Notice in Table 62-3 that the td/rd indicator bit is the same one that controls the underline enhancement in data event-words.

   The value in the low byte is meaningless in the context of special-receive words. The special symbol will overlay or replace the character. Its value, nevertheless, can range from hexadecimal 0 through FF.

Table 62-3
Character Buffer 16-Bit Word

Type	Mask (hex)	Input (hex)	Meaning
data			*data-event word:*
	0100	0100	the low byte contains data
	0500	*add 0100 to the following:*	td/rd indicator:
		0000	td character
		0400	rd character (underlined)
	ff00	*add modified value of td/rd indicator to one (or a combination) of the following:*	enhancements: † (enhancements apply to data indicated in low byte)
		0000	normal
		0200	alternate code set
		0800	reverse image
		1000	hexadecimal
		2000	low intensity
		4000	blink
		8000	strike-thru (parity error on character)
special receive			*special receive-event word:*
	8300	0200	special receive-event word
		8200	reserved
	8700	*add 0200 to the following:*	td/rd indicator:
		0000	td character
		0400	rd character (underlined)
	bf00	*add modified value of td/rd indicator to one of the following:*	special event: (symbols for these events overlay the data indicated in low byte)
		0800	good CRC
		1000	bad CRC
		1800	abort
		2000	flag
		2800	sync
		3000	bad CRC2 (DDCMP)
		3800	good CRC2 (DDCMP)
reserved	0700	0400	reserved
reserved	0f00	0800	reserved

† Selecting rd (0400) for the td/rd indicator results in the data being underlined. The underline enhancement shares the same bit. It has been omitted from the list of enhancements to avoid an error from double counting.

The routines for writing 16-bit event words to the character buffer are *add_event_to_buff* and *add_array_to_buff*. These routines may be used when the Line Setup menu shows either emulate or monitor mode.

add_event_to_buff

Synopsis

```
extern unsigned int add_event_to_buff(event_word);
unsigned int event_word;
```

Description

The *add_event_to_buff* routine writes the specified input to the 64-Kbyte character buffer.

Inputs

The only input is a 16-bit event-word to be written to the buffer. Table 62-3 lists the coding of event words.

Returns

A one is returned if the event was successfully added to the character buffer. If the routine failed, zero is returned.

Example

To display only SDLC frames with an address of hexadecimal c2, enter the following spreadsheet program:

```
LAYER: 1
{
 extern unsigned short rcvd_char_td;
 extern unsigned short rcvd_char_rd;
}
    STATE: Init
       CONDITIONS: ENTER_STATE
       ACTIONS: CAPTURE BOTH OFF
       NEXT_STATE: address
    STATE: address
       CONDITIONS: DTE STRING "FF"
       ACTIONS:
       {
        if(rcvd_char_td == 0xc2)
          {
           add_event_to_buff (((short)td_modifier << 8) + rcvd_char_td);
           ctl_capture_td(0x00);
          }
       }
       CONDITIONS: DTE STRING "FF"
       ACTIONS: CAPTURE DTE OFF
```

```
CONDITIONS: DCE STRING "FP"
ACTIONS:
{
 if(rcvd_char_rd == 0xc2)
   {
    add_event_to_buff (((short)rd_modifier << 8) + rcvd_char_rd);
    ctl_capture_rd(0x00);
   }
}
CONDITIONS: DCE STRING "PF"
ACTIONS: CAPTURE DCE OFF
```

add_array_to_buff

Synopsis

```
extern unsigned int add_array_to_buff(array_ptr, count);
unsigned short * array_ptr;
unsigned char count;
```

Description

The *add_array_to_buff* routine writes specified elements of an array to the 64-Kbyte character buffer.

Inputs

The first parameter is the location of the array to be written to the character buffer. The array consists of 16-bit *shorts.*

The second parameter is the number of elements in the array to be written. The number of elements which can be written to the buffer must be in the range 0–16. Elements in the array must adhere to the format of event words shown in Table 62-3.

Returns

The result of the *add_array_to_buff* routine is all or nothing. A one is returned when all requested elements of the array are successfully added to the character buffer. If the routine fails, zero is returned and nothing is written to the buffer.

Example

To display on the Data Screen only X.25 packets with an LCN of 004, enter the following spreadsheet program. (This program displays the DTE side of the line only. Additional programming similar to that entered would include DCE data.)

```
LAYER: 1
{
 unsigned short dte_array [100];
 unsigned short lcn;
 extern unsigned short rcvd_char_td;
}
    STATE: Init
        CONDITIONS: ENTER_STATE
        ACTIONS: CAPTURE BOTH OFF
        NEXT_STATE: address
    STATE: address
        CONDITIONS: DTE STRING "FF"
        ACTIONS:
        {
         dte_array [0] = (0x0100 + rcvd_char_td);
        }
        NEXT_STATE: frame_type
    STATE: frame_type
        CONDITIONS: DTE STRING "((XXXXXXX0))"
        ACTIONS:
        {
         dte_array [1] = (0x0100 + rcvd_char_td);
        }
        NEXT_STATE: gfi
        CONDITIONS: DTE STRING "((XXXXXXX1))"
        NEXT_STATE: address
    STATE: gfi
        CONDITIONS: DTE STRING "X"
        ACTIONS:
        {
         dte_array [2] = (0x0100 + rcvd_char_td);
         lcn = ((unsigned int)rcvd_char_td & 0x0f) << 8;
        }
        NEXT_STATE: lcn
    STATE: lcn
        CONDITIONS: DTE STRING "X"
        ACTIONS:
        {
         dte_array [3] = (0x0100 + rcvd_char_td);
         lcn += rcvd_char_td;
         if(lcn == 0x0004)
           {
            add_array_to_buff(dte_array, 4);
            ctl_capture_td(0x00);
            current_state = state_eof;
           }
         else
            current_state = state_address;
         break;
        }
    STATE: eof
        CONDITIONS: DTE STRING "FF"
        ACTIONS: CAPTURE DTE OFF
        NEXT_STATE: address
```

63 EIA

The Test Interface Module (TIM) located in the rear of the INTERVIEW determines the EIA leads available for monitoring and control (Section 12). The variables and routines in this section apply to RS-232, V.35, and RS-449 interface modules. The X.21 module is treated separately in Section 73.

To use the C variables and routines explained in this section, enable EIA leads by selecting **Buffer Control Leads: YES** on the FEB Setup menu. See Section 9.1(B). If no other source for clock is provided, use internal clock (Line Setup menu).

The variables and routines approximate Layer 1 EIA spreadsheet-generated conditions and actions. Their use on the Protocol Spreadsheet is not limited to any particular layer, though normally they belong at Layer 1.

63.1 Variables

With an RS-232, V.35, or RS-449 TIM installed, you may monitor RI, DSR, DTR, CD, CTS, RTS, and UA. The lead names in RS-449 are slightly different: see Table 63-1.

The fast-event variable *fevar_eia_changed* detects a change in EIA leads. It does not establish which lead(s) has changed. Two associated variables, *current_eia_leads* and *previous_eia_leads*, indicate the status of the seven leads. These are two-byte (*short*) variables. Each lead is represented by a different bit in the *short*. Some bits are unused. Table 63-1 lists the mask that can be used to isolate each lead.

Whenever a lead changes, the value in *current_eia_leads* is written to *previous_eia_leads*. Then *current_eia_leads* is updated.

(A) Masking To Detect a Change in a Given Lead

To test whether or not a given lead changed, RTS for example, while disregarding its status, enter the following condition on the Protocol Spreadsheet:

```
CONDITIONS:
{
fevar_eia_changed && (((current_eia_leads ^ previous_eia_leads) & 0x80) == 0x80)
}
```

Select a mask value from the list in Table 63-1 to indicate which lead you care about. Specify multiple leads with a mask derived via hexadecimal addition.

Table 63-1
EIA Variables

Type	Variable	Value (hex/decimal)	Meaning
extern fast_event	fevar_eia_changed		True when the status changes for an EIA lead (non-data). Line Setup configured for emulate or monitor mode.
			RS-232/V.35: (RS-449)
extern const volatile unsigned short	current_eia_leads	4	RI (IC)
		8	DSR (DM)
		10/16	DTR (TR)
		20/32	CD (RR)
		40/64	CTS (CS)
		80/128	RTS (RS)
		200/512	UA
			A value in this list, when *anded* (&) with *current_eia_leads*, equals zero if the lead is *on*. Example: STATE: rts_on { *if ((current_eia_leads & 0x80) == 0) sound_alarm(); }*
			Note: This variable will store EIA status if (1) internal or external clock is supplied and (2) EIA leads are enabled on FEB Setup. Line Setup configured for emulate or monitor mode.
extern const volatile unsigned short	previous_eia_leads		Same values as *current_eia_leads*. Updated only after logic has had a chance to compare current and previous leads. Line Setup configured for emulate or monitor mode.

The mask for RTS is 0x80. In the example, the event *fevar_eia_changed* updated *current_eia_leads*. The new *current_eia_leads* was *bitwise-exclusive-OR*ed with *previous_eia_leads* to identify all the leads that changed. Then the result was *bitwise AND*ed with the RTS mask to determine if RTS was among the leads that changed. If this result was equal to the mask, the lead changed.

(B) Masking For the Status of a Lead

You may also test the current status of a lead, independent of any change. *And* the mask with *current_eia_leads*, as in this *if* statement testing for RTS "on":

```
STATE: test_for_rts_on
  {
   if((current_eia_leads & 0x80) == 0)  sound_alarm();
  }
```

If the result is zero, the lead is on. If the result equals the mask, the lead is off. "On" means that a lead is more positive than +3 volts with respect to signal ground. "Off" implies only that a lead is not at or above the "on" threshold, not necessarily that a minus threshold has been attained.

(C) Detect Change and Current Status

The two examples shown above could be combined to test for RTS changing from off to on:

```
CONDITIONS:
{
  (fevar_eia_changed && (((current_eia_leads ^ previous_eia_leads) & 0x80) == 0x80) &&
      ((current_eia_leads & 0x80) == 0))
}
```

This example approximates the translator's version of the spreadsheet-token condition EIA RTS ON when it appears alone in a conditions block. When an EIA condition is combined with another condition, in most cases the other condition will supply the event variable and only the EIA status test will be used.

63.2 Routines

You may control RS-232 EIA leads in emulate mode only. When the Line Setup menu shows **Mode:** EMULATE DCE, you control CTS, CD, and DSR. An EMULATE DTE selection gives you control over RTS and DTR. Entries on the Interface Control menu may be used to set the leads' initial status (Section 12.6).

ctl_eia

Synopsis

```
extern void ctl_eia(on_mask, off_mask);
unsigned short on_mask;
unsigned short off_mask;
```

Description

The *ctl_eia* routine allows you to control the status of up to three of nine possible leads. Which leads you control depends on your emulation mode. The softkey equivalent of this routine is the EIA action on the Protocol Spreadsheet.

Inputs

The first parameter indicates which leads you want to turn on. Each bit in the parameter controls a given lead: RTS/CTS (01), DTR/DSR (02), CD (04), AUX0 (10), AUX1 (20), AUX2 (40), AUX3 (80). Wherever there is a *zero* in the first parameter, the corresponding lead will be turned on. A one in this parameter will *not* cause any lead to be turned off. A value of 0xff will mean *don't care* (no action).

The second parameter indicates which leads you want in the "off" condition. Each bit in the parameter controls a given lead: RTS/CTS (01), DTR/DSR (02), CD (04), AUX0 (10), AUX1 (20), AUX2 (40), AUX3 (80). Wherever there is a *one* in the second parameter, the corresponding lead will be turned off. Zeroes in this parameter do *not* turn leads on. A value of 0 will mean *don't care* (no action).

> NOTE: If both bytes are attempting to control the same lead, the off parameter will override the on parameter.

Example

Suppose your emulate mode is EMULATE DCE. As a DCE, you control the CTS, DSR, and CD leads. (An attempt to control the status of RTS or DTR will fail, since the DTE controls these leads.) When RTS is raised, you want to turn CTS on; when RTS drops, turn CTS off.

```
LAYER: 1
    STATE: control_cts
       CONDITIONS: EIA RTS ON
       ACTIONS:
       {
        ctl_eia(0xfe, 0x00);
       }
       CONDITIONS: EIA RTS OFF
       ACTIONS:
       {
        ctl_eia(0xff, 0x01);
       }
```