64 Display Window and Trace

The C structures, variables, and routines detailed in this section control the type and location of certain displays on the INTERVIEW. These displays can be grouped into three categories.

The first display area is the prompt line, the second line on all Run-mode screens.

The second type of display utilizes the Display Window, available as a selection on the Display Setup portion of the Line Setup menu, or conditionally accessible via softkey during Run mode. To write to the Display Window, use the *pos_cursor* (or *restore_cursor*) and *displayc*, *displayf*, or *displays* routines. When using Display Window, you may position the cursor before output is generated on the screen.

The third type of display utilizes one or a combination of the eight available trace buffers. Trace screens are conditionally accessible via softkey during Run mode. Seven user-traces appear as choices under the User Trace selection on the Display Setup menu. The remaining trace is Program Trace, also an option on Display Setup. Program Trace enables you to track any or all layers, one or all tests, and movement between states. To write to any of the eight trace-screens, use the *tracec*, *tracef*, and *traces* routines.

> NOTE: You may not use the *pos_cursor* routine to position the cursor on any trace screen. New lines (or blank lines) may be generated via the "\n" specifier.

Attributes—color, underlining, and font, for example—may be assigned to characters in the Display Window and all of the Trace buffers.

> NOTE: Color attributes are applied to the RGB output signal, not to the plasma screen.

64.1 Current Display Mode

A group of variables keeps track of softkey movement from one display screen to another (see Table 64-1). When you move from the Display Window to the Program Trace screen, for example, the *fast-event* variable *display_screen_changed* indicates the change of display. The coded value for Display Window now is stored in *prev_display_screen*, and the value for Program Trace can be found in *crnt_display_screen*.

These variables also distinguish between Run mode and Freeze mode. This distinction is important since some keys on the keyboard are mode-dependent. In Freeze mode, for instance, cursor keys automatically become operational for scrolling through the buffer. The programmer will want to avoid using these keys as user-input when *crnt_display_screen* indicates that the unit is in Freeze mode.

Table 64-1
Current Display Variables

Type	Variable	Value (hex/decimal)	Meaning
extern fast_event	display_screen_changed		True when Run-mode display-screen is changed, or when Run/Freeze mode is changed. Value in *crnt_display_screen* is stored in *prev_display_screen*, and *crnt_display-screen* is updated. Line Setup configured for emulate or monitor mode.
extern unsigned short	crnt_display_screen		Contains current display screen (low byte) and indicates whether unit is in Run mode or Freeze mode (high byte). Line Setup configured for emulate or monitor mode.
			display-screen
		0	no display
		1	single-line data
		2	dual-line data
		3	single-line data with leads
		4	dual-line data with leads
		11/17	tabular statistics
		12/18	graphic statistics
		21/33	Display Window
		31/49	Program Trace
		41/65	Layer 1 Protocol Trace
		42/66	Layer 2 Protocol Trace
		43/67	Layer 3 Protocol Trace
		44/68	Layer 4 Protocol Trace
		45/69	Layer 5 Protocol Trace
		46/70	Layer 6 Protocol Trace
		47/71	Layer 7 Protocol Trace
		51/81	User Trace 1
		52/82	User Trace 2
		53/83	User Trace 3
		54/84	User Trace 4
		55/85	User Trace 5
		56/86	User Trace 6
		57/87	User Trace 7
		61/97	TIM package standard stats
		62/98	TIM package aux
			Run/Freeze mode (bit 9)
		100/256	Freeze mode
		0	Run mode

Table 64-1 (continued)

Type	Variable	Value (hex/decimal)	Meaning
extern unsigned short	prev_display_screen		Contains previous display screen (low byte) and indicates whether unit was in Run mode or Freeze mode (high byte). Line Setup configured for emulate or monitor mode.
			display-screen
		0	no display
		1	single-line data
		2	dual-line data
		3	single-line data with leads
		4	dual-line data with leads
		11/17	tabular statistics
		12/18	graphic statistics
		21/33	Display Window
		31/49	Program Trace
		41/65	Layer 1 Protocol Trace
		42/66	Layer 2 Protocol Trace
		43/67	Layer 3 Protocol Trace
		44/68	Layer 4 Protocol Trace
		45/69	Layer 5 Protocol Trace
		46/70	Layer 6 Protocol Trace
		47/71	Layer 7 Protocol Trace
		51/81	User Trace 1
		52/82	User Trace 2
		53/83	User Trace 3
		54/84	User Trace 4
		55/85	User Trace 5
		56/86	User Trace 6
		57/87	User Trace 7
		61/97	TIM package standard stats
		62/98	TIM package aux
			Run/Freeze mode (bit 9)
		100/256	Freeze mode
		0	Run mode

64.2 Prompt Line

Access to the prompt line is always available via the *display_prompt* routine, or its softkey equivalent, the PROMPT action. Attributes may not be assigned to a prompt created via either of these methods. (To create a prompt with attributes, use the *pos_cursor* and *displayf* routines.) Prompts appear on whatever screen is active at the time the prompt is written, including trace screens. With one exception, movement to another display erases the prompt. The only screen which retains the most recent prompt is the Display Window.

You may also position the cursor to the prompt line in the Display Window via the *pos_cursor* routine. The initial position of the cursor in the Display Window is at the beginning of the prompt line—row zero, column zero. Anything written to this cursor

position in the Display Window will appear as a prompt on any one of the other display screens (assuming one of them is active at the time the message is written). Position the cursor below the prompt line for messages intended for the Display Window only.

Trace buffers retain no record of prompts. When you write to a trace screen, the initial position of the cursor is the line immediately below the prompt line—row one. Since you may not position the cursor in trace buffers, all messages written to trace buffers are appended at the end of the buffer. You may never access the prompt line via *tracef* (or *tracec* or *traces*) routines.

64.3 Display Window

The Display Window preserves one screen, including the prompt line, of user-entered messages. When the end of the display screen is reached, the previous messages are overwritten, beginning at row one (the line below the prompt line).

> NOTE: Use the keyboard variables and the *send_key* routine explained in Section 72, Other Library Tools, to program the Run-mode use of ⊡ and ⊡ in the Display Window. (For other Run-mode screens, these keys control the playback speed of disk data.)

(A) Variables

There are variables accessible to the user which provide information about the Display Window. Table 64-2 lists the variables and their possible values. Two variables indicate the current position of the cursor: *current_line* stores the row number and *current_col* stores the column number. To find out which attributes are active in the Display Window, check the values stored in *window_color* and *window_modifier*. Color is stored in the high byte of the two-byte variable *window_color*. Enhancements are stored in the low byte. The current font code can be found in *window_modifier*.

> NOTE: Attributes assigned via the *%m* conversion specifier (refer to *tracef*-routine input) to characters in trace buffers will not alter the values of *window_color* and *window_modifier*. These variables refer to the Display Window only.

The variable *display_window_buffer* provides the user with access to the display-window buffer. This variable is an array of 1,088 *longs*. Each element in the array contains one byte of character data and three bytes of attributes. The attributes are determined by the current values of *window_color* and *window_modifier*.

Table 64-2
Display Window Variables

Type	Variable	Value (hex/decimal)	Meaning
extern unsigned short	current_line	*0-1010-16*	Contains the current row number of the cursor position in the Display Window. Line Setup configured for emulate or monitor mode.
extern unsigned short	current_col	*0-3f10-63*	Contains the current column number of the cursor position in the Display Window. Line Setup configured for emulate or monitor mode.
extern unsigned short	window_color		Two-byte variable. Current color selections are indicated in the low byte. Current enhancements are indicated in the high byte. Written to by *%m* conversions. Attributes are copied into data words in Display Window. Line Setup configured for emulate or monitor mode.
			Isolate bits of interest via *bitwise anding* (&) of mask with variable. Compare result to value column. For example, underline attribute mask = 0x100. Therefore *window_color & 0x100* equals *0* (underline off) or *0x100* (underline on). Line Setup configured for emulate or monitor mode.
			background color mask = 7 (bits 1-3):
		0	black
		1	blue
		2	green
		3	cyan
		4	red
		5	magenta
		6	yellow
		7	white
			foreground color mask = 0x38 (bits 4-6):
		0	black
		8	blue
		10/16	green
		18/24	cyan
		20/32	red
		28/40	magenta
		30/48	yellow
		38/56	white

Table 64-2 (continued)

Type	Variable	Value (hex/decimal)	Meaning
	(window_color continued)		*color blink mask = 0x40 (bit 7):*
		0	no blink
		40/64	blink
			color strike-thru mask = 0x80 (bit 8):
		0	no strike-thru
		80/128	strike-thru
			overline mask = 0x100 (bit 9):
		0	no overline
		100/256	overline
			blank mask = 0x200 (bit 10):
		0	no blank
		200/512	blank
			underline mask = 0x400 (bit 11):
		0	no underline
		400/1024	underline
			reverse image mask = 0x800 (bit 12):
		0	no reverse image
		800/2048	reverse image
			hex mask = 0x1000 (bit 13):
		0	no hex
		1000/4096	hex
			low intensity mask = 0x2000 (bit 14):
		0	no low intensity
		2000/8192	low intensity (RS-170 output)
			monochrome blink mask = 0x4000 (bit 15):
		0	no monochrome blink
		4000/16384	monochrome blink
			monochrome strike-thru *mask = 0x8000 (bit 16):*
		0	no monochrome strike-thru
		8000/32768	monochrome strike-thru

Table 64-2 (continued)

Type	Variable	Value (hex/decimal)	Meaning
extern unsigned char	window_modifier		Contains the current modifiers. Line Setup configured for emulate or monitor mode.
			font mask = 7 (bits 1-3):
		0	ASCII
		1	special graphic character set (refer to Table 64-5)
		2	primary font—code selected on Line Setup
		3	alternate font—current implementation is for call-setup phase in X.21 (ASCII)
		7	hexadecimal
extern unsigned long	display_window_buffer [1088]		Array of 32-bit words that make up the one-screen Display Window. Each word contains three bytes of attributes and a one-byte character. Refer to Table 64-4. Line Setup configured for emulate or monitor mode.

(B) Structures

Once the data word is written to the buffer as an element in the *display_window_buffer* array, it can be accessed and written to—and therefore changed—the same as any other location in memory. There is an *extern* array, *display_window_index_buffer[17]*, which provides a method of informing the display controller on the CPM card that the display needs to be updated. The structure of this array is shown in Table 64-3.

Each element in the *display_window_index_buffer* array represents a horizontal row or line in the Display Window. Each element is a structure with two variables, *mpm* and *cpm*. The first variable in the structure, *mpm*, increments automatically whenever a line in the display-window buffer is updated by a display routine. (If you write to the buffer directly without using one of the display routines, you must increment this variable "manually.") Its particular value at any moment is not important. What is significant is whether or not the value of the second variable in the structure, *cpm*, is the same as *mpm*. The processor on the CPM compares these two variables (for each line) periodically to determine if a line in the Display Window needs to be rewritten. If the values of the two variables do not match, it means that a line updated in memory now needs to be updated by the CPM display-controller software. After the display is changed, the value of *mpm* is copied automatically into *cpm*.

Table 64-3
Display Window Buffer Structures

Type	Variable	Value (hex/decimal)	Meaning
Structure Name:	**display_window_index_buffer [17]**		An array of structures used for detecting changes to the display-window buffer. There are seventeen elements in the array, one for each line in the Display Window. When a change is made to a line in the display-window buffer, the corresponding element in the array is accessed. If a *displayf* routine changes line 3, *display_window_ index_buffer[3].mpm* is automatically incremented. The CPM detects the difference between *display_window_index_buffer [3].mpm* and *display_window_index_buffer [3].cpm* and updates line 3 in the Display Window. Declared as type *extern struct*. You must increment an *mpm* variable manually when you write *directly* (not via a *displayf* routine) to the Display Window.
unsigned char	mpm	*0-ff/0-255*	When the MPM updates a line in the display-window buffer, this variable is incremented.
unsigned char	cpm	*0-ff/0-255*	The CPM checks the value of this variable against the value of *mpm*. If they are different, the value in *mpm* is copied into *cpm*. The updated line in MPM is then presented on the display-window screen.

(C) Routines

You may position the cursor before output is generated on the screen via the *pos_cursor* and *restore_cursor* routines. The *pos_cursor* routine positions the cursor at the row and column you specify. The *restore_cursor* routine returns the cursor to a previous location.

One routine, *displayf*, allows you to add attributes to messages in the Display Window, including the prompt line. These attributes are listed in Table 64-4.

Additional routines control the labeling of Display Window softkeys: *set_dw_fkey_label*, *show_dw_fkey_labels*, *highlight_dw_fkey_label*, and *unhighlight_dw_fkey_label*.

displayc

Synopsis

```
extern void displayc(character);
const char character;
```

Description

The *displayc* routine outputs a single ASCII character to the Display Window screen. The placement of the output on the screen may be controlled via the *pos_cursor* routine. Attributes may not be used in *displayc*.

Inputs

The parameter value may be given as a hexadecimal, octal, or decimal constant; as an alphanumeric constant inside of single quotes; or as a variable. A hexadecimal value must be preceded by the prefix 0x or 0X; an octal value must be preceded by the prefix 0. If no prefix appears before the input, the number is assumed to be decimal. Valid numeric entries are 00 to 127, decimal. An alphanumeric character placed between single quotes will be output as is to the display.

Example

The *displayc* entries on the left output the character given on the right, at the cursor location on the Display Window screen:

```
displayc('a');      a
displayc(65);       A
displayc(0x65);     e
displayc(065);      5
```

displayf

Synopsis

```
extern int displayf(format_ptr, . . . );
const char * format_ptr;
```

Description

The *displayf* routine writes output to the Display Window screen, under control of the string pointed to by *format_ptr* that specifies how subsequent arguments are converted for output. If there are insufficient arguments for the format, the behavior is undefined. If the format is exhausted while arguments remain, the excess arguments are evaluated but otherwise ignored. The *displayf* routine returns when the end of the format string is encountered. The placement of the output on the screen may be controlled via the *pos_cursor* routine.

Inputs

The format is composed of zero or more directives: ordinary characters (not %), which are copied unchanged to the output stream; and conversion specifications, each of which results in fetching zero or more subsequent arguments. Each conversion specification is introduced by the character %. After the %, the following appear in sequence:

- Zero or more *flags* that modify the meaning of the conversion specification. The flag characters and their meanings are:

  - The result of the conversion will be left-justified within the field.

  + The result of a signed conversion will always begin with a plus or minus sign.

  space If the first character of a signed conversion is not a sign, a space will be prepended to the result. If the *space* and + flags both appear, the *space* flag will be ignored.

  # The result is to be converted to an "alternate form." For d and i conversions, the flag has no effect. For o conversion, it increases the precision to force the first digit of the result to be a zero. For x (or X) conversion, a nonzero result will have 0x (or 0X) prepended to it. For u conversions, the argument is displayed in small hex characters. For example, displayf ("%#u", 258); yields %1%2. For c and s conversions, if the argument contains a newline character, it is displayed as ↳.

- An optional decimal integer specifying a minimum *field width*. If the converted value has fewer characters than the field width, it will be padded on the left (or right, if the left adjustment flag, described above, has been given) to the field width. The padding is with spaces unless the field width integer starts with a zero, in which case the padding is with zeros.

- An optional *precision* that gives the minimum number of digits to appear for the d, i, o, u, x, and X conversions, the maximum number of characters to be written from an array in an s conversion, or the number of characters to be written from an array in an H conversion (overriding the usual null-termination of strings). The precision takes the form of a period (.) followed by an optional decimal integer; if the integer is omitted, it is treated as zero. The amount of padding specified by the precision overrides that specified by the field width.

- An optional h specifying that a following d, i, o, u, x, or X conversion specifier applies to a *short int* or *unsigned short int* argument (the argument will have been promoted according to the integral promotions, and its value shall be converted to *short int* or *unsigned short int* before printing); or an optional l specifying that a following d, i, o, u, x, or X conversion specifier applies to a *long int* or *unsigned long int* argument. If an h or l appears with any other conversion specifier, it is ignored.
- A character that specifies the type of *conversion* to be applied. (Special AR extensions have been added.) The conversion specifiers and their meanings are:

  d, i, o, u, x, X

  The *int* argument is converted to signed decimal (d or i), unsigned octal (o), unsigned decimal (u), or unsigned hexadecimal notation (x or X); the letters abcdef are used for x conversion and the letters ABCDEF for X conversion. The precision specifies the minimum number of digits to appear; if the value being converted can be represented in fewer digits, it will be expanded with leading zeros. The default precision is 1. The result of converting a zero value with a precision of zero is no characters.

  c The *int* argument is converted to an *unsigned char*, and the resulting character is written.

  s The argument shall be a pointer to a null-terminated array of 8-bit *chars*. Characters from the string are written up to (but not including) the terminating null character: if the precision is specified, no more than that many characters are written. The string may be an array into which output was written via the *sprintf* routine. (If the string pointed to is an array which has been written via the *stracef* routine, you must use *%b* rather than *%s* to display it.)

  p The argument shall be a pointer to void. The value of the pointer is converted to a sequence of printable characters, in this format: *0000:0000*. There are always exactly 4 digits to the right of the colon. The number of digits to the left of the colon is determined by the pointer's value and the precision specified. Use this conversion to display 80286 memory addresses. The 16-bit segment number will appear to the left of the colon and the 16-bit offset to the right.

  % A % is written. No argument is converted.

  \n Displays ↳. No argument is converted. When \n is *not* preceded by a %, it is not a conversion specifier. Instead of a ↳ being displayed, a newline (␤↳) will be executed.

  H displays a character array (pointed to by the argument) as small hex characters. If precision is specified, it is used as the length of the array (overriding the usual null-termination of strings).

b The argument shall be a pointer to an array of 32-bit words. Characters from the string are written up to (but not including) the terminating word containing a null character: if the precision is specified, no more than that many words are written. If the string pointed to is an array into which output was written via the *stracef* routine, you must use *%b* rather than *%s* to display it. (To display the information in an array written to via *sprintf*, use *%s*.)

m The argument is a *long* integer that indicates attributes to be assigned to subsequent characters. Attributes stay "on" until they are specifically turned "off" with another *%m* conversion specifier. The lowest-order byte contains primarily font code. The next higher byte is not used to set attributes. (In the display-window buffer, this second byte is reserved for character coding.) The third byte holds color information. The high byte indicates which enhancements should be invoked.

Attributes are written automatically to *window_color* and *window_modifier* variables, then copied into subsequent 32-bit data words in the Display Window. Table 64-4 shows the format of this 32-bit word.

Attributes may not be assigned as a one-byte value. Even if you want to alter only one attribute setting, color for example, you must include it as part of a *long*. Append an "L" at the end of the hexadecimal code specifying attributes to indicate the value is a *long*.

NOTE: If you are specifying an attribute in a lower-order byte of the *long*, color for example, and you want the high byte (or bytes) to be zero, you may omit the high byte provided you have the "L" appended at the end of the hexadecimal code. The high byte (or bytes) will be left-padded with zeroes. For example, 0x200000L is converted to 0x00200000L. Associated characters will be displayed on a color monitor as green on a black background, as dictated by the hexadecimal 20 in the third byte. Enhancements are controlled in the high byte, now assigned a value of zero. Any enhancements previously turned "on" will be turned "off."

If a conversion specification is invalid, the behavior is undefined.

If any argument is or points to an aggregate (except for an array of characters using *%s* conversion or any pointer using *%p* conversion), the behavior is undefined.

In no case does a nonexistent or small field width cause truncation of a field; if the result of a conversion is wider than the field width, the field is expanded to contain the conversion result.

Returns

The *displayf* routine returns the number of characters displayed.

Example

To display a date and time in the form "Sunday, July 3, 10:02," where weekday and month are pointers to strings:

```
LAYER: 1
{
 unsigned char weekday [10];
 unsigned char month [10];
 unsigned short day;
 unsigned char hour;
 unsigned char min;
}
     STATE: output_to_display_window
        CONDITIONS: KEYBOARD " "
        ACTIONS:
        {
         displayf( "%s, %s %d, %.2d:%.2d\n", weekday, month, day, hour, min);
        }
```

sprintf

The *sprintf* routine is similar to the *displayf* routine. *displayf* writes output with or without attributes directly to the *Display Window*. *sprintf*, fully documented in Section 67.3, writes output to a *character array* in which attributes are not supported. This routine is useful for writing formatted output to a display, printer, or file.

See also *stracef* in Section 64.4(C).

Table 64-4
Display Window/Trace Buffer 32-Bit Data Word

Bit	Mask (hex)†	Input (hex)††	Meaning
			Modifier attributes, font for example, are contained in the low byte of the 32-bit word.
1-3	00000007L		Font:
		00000000L	ASCII
		00000001L	special graphic character set (refer to Table 64-5)
		00000002L	primary font—code selected on Line Setup
		00000003L	alternate font—current implementation is for call-setup phase in X.21 (ASCII)
		00000007L	hexadecimal
4	00000008L		Special character indicator: (used in trace buffer only; should not be altered by user)
		00000000L	only value in *modifier* in trace buffer header
		00000008L	Character is not displayable but contains control info used internally by the trace logic. When a "\n" is included in a *tracef* routine, for example, a new line is generated, but nothing is displayed on the trace screen. The *tracef* routine automatically sets this bit before the 32-bit word is written into *trace_buf.array*.
5-8	000000f0L	00000000L	unused, but should be zero
9-16	0000ff00L	00000000L	*Character data* is contained in the second byte of the *long* word. Input should be 00 in all %*m* conversions.

† Use the masks to change attributes of characters in the Display Window or trace buffer. In the Display Window, characters are represented in the second byte of the *longs* that comprise the 1,088 array elements in *display_window_buffer*. In the *trace_buf* structure, the characters are represented in the second byte of the *longs* that make up the *trace_buf.array*. To change one attribute of a character while leaving the others unchanged:

display_window_buffer[position] = ((display_window_buffer[position] & (~attribute-mask)) | input);

To change only the font of the twenty-first character in the trace buffer from its current setting to the special graphic font, for example:

l2_trbuf.array[20] = ((trace_buf.array[20] & (~0x00000007L)) | 0x00000001L);

*And*ing the character with the mask will indicate the current setting of an attribute:

If *(l2_trbuf.array[20] & 0x00000007L)* equals 2, then the 21st character in the Trace 2 user-trace buffer is being displayed in the font selected on the Line Setup menu.

†† In *displayf* routines, the %*m* conversion specifier writes input to the *window_color* and *window_modifier* variables. These variables are copied into subsequent data words in the Display Window. In *tracef* routines, the %*m* conversion specifier writes input to *trace_buffer_header*. The header is then copied into each subsequent data word in the buffer. Combine attributes via hexadecimal addition.

Table 64-4 (continued)

Bit	Mask (hex)	Input (hex)	Meaning
			Color is contained in the third byte of the *long*. Combine color attributes via hexadecimal addition.
17-19	00070000L		Background color:
		00000000L	black
		00010000L	blue
		00020000L	green
		00030000L	cyan
		00040000L	red
		00050000L	magenta
		00060000L	yellow
		00070000L	white
20-22	00380000L		Foreground color:
		00000000L	black
		00080000L	blue
		00100000L	green
		00180000L	cyan
		00200000L	red
		00280000L	magenta
		00300000L	yellow
		00380000L	white
23	00400000L		Color blink:
		00000000L	no blink
		00400000L	blink
24	00800000L		Color strike-thru:
		00000000L	no strike-thru
		00800000L	strike-thru
			Enhance attributes, underlining for example, are contained in the high byte of the *long*. Combine enhancements via hexadecimal addition.
25	01000000L		Overline: (for monochrome and color)
		00000000L	no overline
		01000000L	overline
26	02000000L		Blank:
		00000000L	monochrome display, color display
		02000000L	monochrome no display, color display
27	04000000L		Underline: (for monochrome and color)
		00000000L	no underline
		04000000L	underline

Table 64-4 (continued)

Bit	Mask (hex)	Input (hex)	Meaning
28	08000000L		Monochrome reverse image:
		00000000L	no reverse image
		08000000L	reverse image
29	10000000L		Hex:
		00000000L	no hex
		10000000L	hex
30	20000000L		Monochrome low intensity:
		00000000L	no low intensity
		20000000L	low intensity (RS-170 interface)
31	40000000L		Monochrome blink:
		00000000L	no blink
		40000000L	blink
32	80000000L		Monochrome strike-thru:
		00000000L	no strike-thru
		80000000L	strike-thru

Table 64-5
Special Graphic Character Set†

Display	Input (hex/decimal)	Display	Input (hex/decimal)
⌜	0	─	1c/28
⌜	1	│	1d/29
⎺	2	┬	1e/30
─	3	┴	1f/31
⎽	4	┤	20/32
├	5	├	21/33
┼	6	▒	22/34
⎺	7	▒	23/35
⎽	8	▒	24/36
⌜	9	▒	25/37
⌜	a/10	▒	26/38
⌜	b/11	▒	27/39
⌜	c/12	▒	28/40
⌜	d,11/13,17	▒	29/41
⌜	e/14	▒	2a/42
⌜	f/15	▒	2b/43
⌜	10/16	▒	2c/44
⌜	12/18	▒	2d/45
⌜	13/19	▒	2e/46
⌜	14/20	▒	2f/47
⌜	15/21	▒	30/48
⌜	16/22	*(space)*	31/49
⌜	17/23	↑	32/50
⌜	18/24	↓	33/51
⌜	19/25	←	34/52
⌜	1a/26	→	35/53
┼	1b/27	▒	36/54

† Written to the Display Window or a trace buffer when low (modifier) byte of 32-bit data word = 0x01.

Table 64-5 (continued)

Display	Input (hex/decimal)	Display	Input (hex/decimal)
¥	80/128	コ	9a/154
｡	81/129	サ	9b/155
｢	82/130	シ	9c/156
｣	83/131	ス	9d/157
、	84/132	セ	9e/158
・	85/133	ソ	9f/159
ヲ	86/134	タ	a0/160
ァ	87/135	チ	a1/161
ィ	88/136	ツ	a2/162
ゥ	89/137	テ	a3/163
ェ	8a/138	ト	a4/164
オ	8b/139	ナ	a5/165
ャ	8c/140	ニ	a6/166
ュ	8d/141	ヌ	a7/167
ョ	8e/142	ネ	a8/168
ッ	8f/143	ノ	a9/169
ー	90/144	ハ	aa/170
ア	91/145	ヒ	ab/171
イ	92/146	フ	ac/172
ウ	93/147	ヘ	ad/173
エ	94/148	ホ	ae/174
オ	95/149	マ	af/175
カ	96/150	ミ	b0/176
キ	97/151	ム	b1/177
ク	98/152	メ	b2/178
ケ	99/153	モ	b3/179

Table 64-5 (continued)

Display	Input (hex/decimal)	Display	Input (hex/decimal)
ﾔ	b4/180	Ä	ce/206
ﾕ	b5/181	Å	cf/207
ﾖ	b6/182	É	d0/208
ﾗ	b7/183	æ	d1/209
ﾘ	b8/184	Æ	d2/210
ﾙ	b9/185	ô	d3/211
ﾚ	ba/186	ö	d4/212
ﾛ	bb/187	ò	d5/213
ﾜ	bc/188	û	d6/214
ﾝ	bd/189	ù	d7/215
ﾞ	be/190	ÿ	d8/216
•	bf/191	Ö	d9/217
Ç	c0/192	Ü	da/218
ü	c1/193	¢	db/219
é	c2/194	£	dc/220
â	c3/195	ß	dd/221
ä	c4/196	₧	de/222
à	c5/197	ƒ	df/223
å	c6/198	á	e0/224
ç	c7/199	í	e1/225
ê	c8/200	ó	e2/226
ë	c9/201	ú	e3/227
è	ca/202	ñ	e4/228
ï	cb/203	Ñ	e5/229
î	cc/204	ª	e6/230
ì	cd/205	º	e7/231

Table 64-5 (continued)

Display	Input (hex/decimal)	Display	Input (hex/decimal)
¿	e8/232	ï	ed/237
¬	e9/233	··	ee/238
¬	ea/234	§	ef/239
½	eb/235	•	f0/240
¼	ec/236		

displays

Synopsis

```
extern void displays(string_ptr);
const char * string_ptr;
```

Description

The *displays* routine writes output to the Display Window screen, under control of the string that is pointed to by *string_ptr*. The *displays* routine returns when the end of the string is encountered. The placement of the output on the screen may be controlled via the *pos_cursor* routine. Attributes may not be used in *displays*.

Inputs

The input is a pointer to a string composed of zero or more ordinary characters. Octal or hexadecimal values also may be included in the string, with octal preceded by \ and hex by \x. Pad each value to three integers with leading zeroes.

Example

The following entry

```
pos_cursor( 0, 0 );
displays("End of test.");
```

produces the following output on the prompt line:

```
End of test.
```

The following coding produces the same output:

```
pos_cursor( 0, 0 );
const char * string_ptr;
string_ptr = "End of test.";
displays (string_ptr);
```

display_prompt

Synopsis

```
extern void display_prompt(string_ptr);
const char * string_ptr;
```

Description

The *display_prompt* routine displays a designated string at the beginning of the prompt line. The cursor is automatically positioned at row zero, column zero. Once the prompt is written, the cursor is returned to its previous position. The softkey equivalent of this routine is the PROMPT action. The prompt is visible on whichever display screen is active at the time the prompt is written. The most recent prompt is retained in the Display Window. Attributes may not be used in *display_prompt*.

Inputs

The input is a pointer to a string composed of zero or more ordinary characters. Octal or hexadecimal values also may be included in the string, with octal preceded by \ and hex by \x. Pad each value to three integers with leading zeroes.

Example

Refer to the example provided for the *displays* routine. The same string could be output to the same position without calling the *pos_cursor* routine:

```
display_prompt("End of test.");
```

or

```
const char * string_ptr;
string_ptr = "End of test.";
display_prompt (string_ptr);
```

pos_cursor

Synopsis

```
extern unsigned int pos_cursor(row, column);
unsigned char row;
unsigned char column;
```

Description

This routine positions the cursor on the Display Window screen by row and column numbers.

> NOTE: The *pos_cursor* routine may not be used to position the cursor on trace screens.

Inputs

The first parameter is the row number. Possible values: 0-16. (The top line of the screen is reserved for header information and cannot be written to.)

The second parameter is the column number. Possible values: 0-63.

Returns

The *pos_cursor* routine returns the previous cursor position in the form of an *unsigned int*. The high byte contains the row number; the low byte identifies the column number.

Example

To position the cursor at the far left edge of the prompt line on the Display Window, enter zero for both parameters.

```
LAYER: 4
    STATE: write_to_display
        CONDITIONS: KEYBOARD " "
        ACTIONS:
        {
         pos_cursor(0,0);
         displays("Display on prompt line.");
        }
```

restore_cursor

Synopsis

```
extern void restore_cursor(position);
unsigned int position;
```

Description

The *restore_cursor* routine returns the cursor to a previous position.

> NOTE: The *restore_cursor* routine may not be used to position the cursor on trace screens.

Inputs

The only input is an *unsigned int* in the same form that is used by the returned value of the *pos_cursor* routine. The high byte identifies the row number. The low byte identifies the column number.

Example

Suppose the cursor is located in the middle of the Display Window. You want to write a message to the prompt line, but return to your previous location on the screen to continue your display.

```
{
 unsigned int previous;
}
     STATE: display
        CONDITIONS: KEYBOARD " "
        ACTIONS:
        {
         pos_cursor(8,0);
         displays("This line begins on row 8, column 0 of the Display Window.");
         previous = pos_cursor(0,0);
         displays("This sentence is on the prompt line.");
         restore_cursor(previous);
         displays("This sentence begins on row 8, column 58 of the Display Window, the
         position of the cursor at the time pos_cursor(0,0) was called.");
        }
```

set_dw_fkey_label

Synopsis

```
extern void set_dw_fkey_label(fkey, label_ptr);
unsigned int fkey;
const char * label_ptr;
```

Description

The *set_dw_fkey_label* routine assigns a user-defined label to a specified Display Window softkey. A call to *set_dw_fkey_label* does not automatically update the label on the Display Window screen. You must press the Run-mode DSP WND softkey *at least once* to access the new rack of softkey labels. After that, you may update the display by calling the *show_dw_fkey_labels* routine.

You may monitor the softkeys associated with your labels only when the user-defined rack of softkeys is active, i.e., the labels are displayed. When the labels are displayed and a function key pressed, the fast-event variable *keyboard_new_any_key* comes true and the variable *keyboard_any_key* is updated according to the values listed below. See Section 72.2 for more information on these variables.

Hex Value	Key Pressed
197	F1
198	F2
199	F3
19a	F4
19b	F5
19c	F6
19d	F7
19e	F8

There is no Protocol Spreadsheet softkey equivalent of this routine.

Inputs

The first parameter identifies the number of the function key to be labeled. Integers from 1 through 8 are valid values. If the specified value is out of the valid range, the label is not assigned to any softkey.

The second parameter is a pointer to a null-terminated string, i.e., the label that should appear below the designated softkey. The label string has a maximum length of seven characters. If it has fewer than seven characters, it is padded to the right with spaces. If it has more than seven characters, only the first seven are used.

Example

In the example below a label is assigned to each of the softkeys in the Display Window. To see the labels displayed, press DSP WND.

```
LAYER: 1
    STATE: define_labels
        CONDITIONS: ENTER_STATE
        ACTIONS:
        {
         set_dw_fkey_label(1, "  one");
         set_dw_fkey_label(2, "  two");
         set_dw_fkey_label(3, " three");
         set_dw_fkey_label(4, "  four");
         set_dw_fkey_label(5, "  five");
         set_dw_fkey_label(6, "  six");
         set_dw_fkey_label(7, " seven");
        }
```

show_dw_fkey_labels

Synopsis

```
extern void show_dw_fkey_labels();
```

Description

The *show_dw_fkey_labels* routine updates the display of all user-assigned softkey labels in the Display Window. For this routine to have any effect, the DSP WND softkey must have been pressed *at least once* and the user-assigned labels must be currently displayed. There is no Protocol Spreadsheet softkey equivalent of this routine.

Example

Enter the Display Window by pressing DSP WND in Run mode. Then alternate between two defined softkey rack by pressing [F8] (labeled MORE) from either rack.

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

```
LAYER: 1
    STATE: first_rack
        CONDITIONS: ENTER_STATE
        ACTIONS:
        {
         set_dw_fkey_label(1, "  one");
         set_dw_fkey_label(2, "  two");
         set_dw_fkey_label(3, " three");
         set_dw_fkey_label(4, "  four");
         set_dw_fkey_label(5, "  five");
         set_dw_fkey_label(6, "  six");
         set_dw_fkey_label(7, " seven");
         set_dw_fkey_label(8, "  MORE");
         show_dw_fkey_labels();
        }
        NEXT_STATE: second_rack
    STATE: second_rack
        CONDITIONS:
        {
         keyboard_new_any_key && (keyboard_any_key == 0x19e)  /* MORE pressed on rack 1 */
        }
        ACTIONS:
        {
         set_dw_fkey_label(1, " eight");
         set_dw_fkey_label(2, "  nine");
         set_dw_fkey_label(3, "  ten");
         set_dw_fkey_label(4, " eleven");
         set_dw_fkey_label(5, " twelve");
         set_dw_fkey_label(6, " thirtn");
         set_dw_fkey_label(7, " fourtn");
         set_dw_fkey_label(8, "  MORE");
         show_dw_fkey_labels();
        }
        NEXT_STATE: wait_for_more
    STATE: wait_for_more
        CONDITIONS:
        {
         keyboard_new_any_key && (keyboard_any_key == 0x19e)  /* MORE pressed on rack 2 */
        }
        NEXT_STATE: first_rack
```

highlight_dw_fkey_label

Synopsis

```
extern void highlight_dw_fkey_label(fkey);
unsigned int fkey;
```

Description

The *highlight_dw_fkey_label* displays a specified user-defined softkey label in reverse video. This routine applies to the Display Window only. There is no Protocol Spreadsheet softkey equivalent of this routine.

Inputs

The only parameter identifies the number of the function key whose label is to be highlighted. Integers from 1 through 8 are valid values. Values outside this range are ignored.

Example

This example is similar to the one for *show_dw_fkey_labels* except that each time a softkey is pressed, its label is highlighted and any previous highlighted label is returned to normal video.

```
{
 extern fast_event keyboard_new_any_key;
 extern volatile unsigned short keyboard_any_key;
 unsigned short current_fkey;  /* currently highlighted fkey label */
}
LAYER: 1
     STATE: first_rack
        CONDITIONS: ENTER_STATE
        ACTIONS:
        {
         unhighlight_dw_fkey_label(current_fkey);
         set_dw_fkey_label(1, "  one");
         set_dw_fkey_label(2, "  two");
         set_dw_fkey_label(3, " three");
         set_dw_fkey_label(4, "  four");
         set_dw_fkey_label(5, "  five");
         set_dw_fkey_label(6, "  six");
         set_dw_fkey_label(7, " seven");
         set_dw_fkey_label(8, "  MORE");
         current_fkey = 0; /* 0 not in range — no fkey highlighted */
         show_dw_fkey_labels();
        }
        NEXT_STATE: second_rack
     STATE: second_rack
        CONDITIONS:
        {
         keyboard_new_any_key && (keyboard_any_key == 0x19e)  /* MORE pressed on rack 1 */
        }
        ACTIONS:
        {
         unhighlight_dw_fkey_label(current_fkey);
         current_fkey = 0; /* no highlight on initial display of rack 2 */
         set_dw_fkey_label(1, " eight");
         set_dw_fkey_label(2, "  nine");
         set_dw_fkey_label(3, "  ten");
         set_dw_fkey_label(4, " eleven");
         set_dw_fkey_label(5, " twelve");
         set_dw_fkey_label(6, " thirtn");
         set_dw_fkey_label(7, " fourtn");
         set_dw_fkey_label(8, "  MORE");
         show_dw_fkey_labels();
        }
```

```
    NEXT_STATE: wait_for_more
    CONDITIONS:
    {
     /* key other than MORE pressed on rack 1 */
     keyboard_new_any_key && ((keyboard_any_key >= 0x197) && (keyboard_any_key <=
       0x19d))
    }
    ACTIONS:
    {
     unhighlight_dw_fkey_label(current_fkey);
     current_fkey = keyboard_any_key - 0x196;
     highlight_dw_fkey_label(current_fkey);
    }
  STATE: wait_for_more
    CONDITIONS:
    {
     /* key other than MORE pressed on rack 2 */
     keyboard_new_any_key && ((keyboard_any_key >= 0x197) && (keyboard_any_key <=
       0x19d))
    }
    ACTIONS:
    {
     unhighlight_dw_fkey_label(current_fkey);
     current_fkey = keyboard_any_key - 0x196;
     highlight_dw_fkey_label(current_fkey);
    }
    CONDITIONS:
    {
     keyboard_new_any_key && (keyboard_any_key == 0x19e)  /* MORE pressed on rack 2 */
    }
    NEXT_STATE: first_rack
```

unhighlight_dw_fkey_label

Synopsis

```
extern void highlight_dw_fkey_label(fkey);
unsigned int fkey;
```

Description

The *unhighlight_dw_fkey_label* displays a specified user-defined softkey label in normal video. This routine applies to the Display Window only. There is no Protocol Spreadsheet softkey equivalent of this routine.

Inputs

The only parameter identifies the number of the function key to be unhighlighted. Integers from 1 through 8 are valid values. Values outside this range are ignored.

Example

See *highlight_dw_fkey_label*.

64.4 Program and User Traces

Unless their sizes are increased, Program Trace and the User Traces retain a maximum of 4096 characters, equivalent to four full screens when every character space is used. (See Section (B)2. below on increasing the size of trace buffers.) When a buffer's limit is reached, new characters written to the end of the buffer force out the same number of characters from the beginning of the buffer. The prompt line is not part of these buffers. Messages are appended to the end of the buffers. In Freeze mode you may scroll through the buffer using the cursor keys.

You write messages to the User Traces only by using C routines. The Run-mode softkeys for User Traces—USER TR, TRACE 1, TRACE 2, TRACE 3, TRACE 4, TRACE 5, TRACE 6, TRACE 7—appear when the buffers are used in a program.

(A) Variables

There are no *extern* variables associated exclusively with Traces.

(B) Structures

1. *Declaring trace buffers.* The trace routines that write to any of the trace buffers require a pointer to the appropriate trace buffer as input. To point to one of the trace buffers, you must first have declared it as a structure. The structure that is common to trace buffers is named *trace_buf*. This structure is already declared in a file called *trace_buf.h* located in the *HRD/sys/include* directory. The *trace_buf* structure contains another structure, *trace_buffer_header*, which also is declared in the *trace_buf.h* file. (These structures are explained in Table 64-6.) Use the *#include* pre-processor directive to include both declarations in your program.

   There are eight trace buffers available (including the Program Trace), each one having its own display screen. All are instances of the *trace_buf* structure. Declare each one you use as an *extern struct*, as in this example:

   ```
   extern struct trace_buf l1_trbuf;
   ```

   The names of all the trace buffers are listed in Table 64-6.

2. *Sizing trace buffers.* There is a preprocessor *#pragma* which allows the user to configure the size of the data array in each user trace buffer. The syntax is TRACE-NUMBER SIZE TRACE-NUMBER SIZE. . . . Trace number 0 refers to the Program Trace buffer, and trace-number "*" allows all trace-buffer arrays to be set at once. All sizes are given in terms of four-byte array elements.

The example below first sets all trace-buffer arrays to 16,000 elements, and then down-sizes array number 3 to 2,048 elements.

```
#pragma tracebuf * 16000  3 2048
```

When a trace buffer is declared, its array will have the size specified in the *#pragma tracebuf* directive. If the buffer was not referenced in a *#pragma tracebuf* directive, its array size will default to 4,096. The maximum size for a trace-buffer array is 16,381 elements. If you specify a size that is too small or too large, an error message will be displayed.

Table 64-6
Trace Buffer Structures

Type	Variable	Value (hex/decimal)	Meaning
Structure Name:	**trace_buffer_header**		Structure of a header for trace buffers. Declared as type *extern struct*. Declared automatically if a softkey-entered TRACE action is taken. Contained in the structure of the trace buffer. Declaration contained in file named *HRD/sys/include/trace_buf.h*. Written to by *%m* conversion specifier. Because it is an extern structure, values of component variables should not be altered directly by the user. In some instances, e.g., altering array size, the result could be a crash.
unsigned short	logical_end	*0-ffff0-4095*	end of data within the buffer. Maximum value is one less than the *array_size*.
unsigned short	logical_end_wrap_count	0 *non-zero*	trace buffer is not full trace buffer is full. As new lines are written to the end of the trace buffer, lines at the beginning of the buffer are removed.
unsigned char	modifier		Special-character indicator bit and bit 8 must be zero. For other specific values and their meanings,see Table 64-4.
unsigned char	color	*0-ff/0-255*	For specific values and their meanings, see Table 64-4.
unsigned char	enhance	*0-ff/0-255*	For specific values and their meanings, see Table 64-4.
unsigned short	write_lock	*0-ffff/0-65535*	prevents two processes from writing to the same buffer at the same time. Should not be altered by user or future access to the trace buffers may be locked out.
unsigned short	array_size	1000/4096	size of buffer; at present only one value
unsigned char	line_size	*0-3f/0-63*	number of characters in last line in buffer
unsigned char	spare	0	reserved for future use
Structure Name:	**trace_buf**		Structure of a trace buffer. Declared as type *extern struct*. Declared automatically if a softkey-entered TRACE action is taken. Declaration contained in file named *HRD/sys/include/trace_buf.h*.
struct trace_buffer_header	hdr		structure of the trace-buffer header described above
unsigned long	array [4096]		array of data words in the buffer

Table 64-6 (continued)

Type	Variable	Value (hex/decimal)	Meaning
Structure Name:	**prog_trbuf**		Structure of the Program Trace buffer, an instance of the *trace_buf* structure declared in file named *HRD/sys/include/trace_buf.h*. Declared as type *extern struct trace_buf*. Declared automatically if a softkey-entered TRACE action is taken. Writing to this buffer makes the Run-mode PROG TR softkey appear.
struct trace_buffer_header	hdr		structure of the trace-buffer header described above
unsigned long	array [4096]		array of data words in the buffer
Structure Name:	**I1_trbuf**		Structure of one of seven user trace buffers, an instance of the *trace_buf* structure declared in file named *HRD/sys/include/trace_buf.h*. Declared as type *extern struct trace_buf*. Writing to this buffer causes the Run-mode TRACE 1 softkey appear.
struct trace_buffer_header	hdr		structure of the trace-buffer header described above
unsigned long	array [4096]		array of data words in the buffer
Structure Name:	**I2_trbuf**		Structure of one of seven user trace buffers, an instance of the *trace_buf* structure declared in file named *HRD/sys/include/trace_buf.h*. Declared as type *extern struct trace_buf*. Writing to this buffer causes the Run-mode TRACE 2 softkey appear.
struct trace_buffer_header	hdr		structure of the trace-buffer header described above
unsigned long	array [4096]		array of data words in the buffer
Structure Name:	**I3_trbuf**		Structure of one of seven user trace buffers, an instance of the *trace_buf* structure declared in file named *HRD/sys/include/trace_buf.h*. Declared as type *extern struct trace_buf*. Writing to this buffer causes the Run-mode TRACE 3 softkey appear.
struct trace_buffer_header	hdr		structure of the trace-buffer header described above
unsigned long	array [4096]		array of data words in the buffer
Structure Name:	**I4_trbuf**		Structure of one of seven user trace buffers, an instance of the *trace_buf* structure declared in file named *HRD/sys/include/trace_buf.h*. Declared as type *extern struct trace_buf*. Writing to this buffer causes the Run-mode TRACE 4 softkey appear.
struct trace_buffer_header	hdr		structure of the trace-buffer header described above
unsigned long	array [4096]		array of data words in the buffer

Table 64-6 (continued)

Type	Variable	Value (hex/decimal)	Meaning
Structure Name: I5_trbuf			Structure of one of seven user trace buffers, an instance of the *trace_buf* structure declared in file named *HRD/sys/include/trace_buf.h*. Declared as type *extern struct trace_buf*. Writing to this buffer causes the Run-mode TRACE 5 softkey appear.
struct trace_buffer_header	hdr		structure of the trace-buffer header described above
unsigned long	array [4096]		array of data words in the buffer
Structure Name: I6_trbuf			Structure of one of seven user trace buffers, an instance of the *trace_buf* structure declared in file named *HRD/sys/include/trace_buf.h*. Declared as type *extern struct trace_buf*. Writing to this buffer causes the Run-mode TRACE 6 softkey appear.
struct trace_buffer_header	hdr		structure of the trace-buffer header described above
unsigned long	array [4096]		array of data words in the buffer
Structure Name: I7_trbuf			Structure of one of seven user trace buffers, an instance of the *trace_buf* structure declared in file named *HRD/sys/include/trace_buf.h*. Declared as type *extern struct trace_buf*. Writing to this buffer causes the Run-mode TRACE 7 softkey appear.
struct trace_buffer_header	hdr		structure of the trace-buffer header described above
unsigned long	array [4096]		array of data words in the buffer

(C) Routines

Most routines defined below are valid for either the Program Trace or the user traces. One, however, applies to the user traces only. *set_utrace_fkey_label* allows the programmer to modify the current softkey labels for the user traces.

The other four trace routines—*tracec*, *tracef*, *stracef*, and *traces*—apply to both the Program Trace and the user traces. The softkey TRACE action is built on the *tracef* routine.

The first argument in three of these trace routines is the address of the trace buffer into which you want output written. Each time you call a trace routine, *tracef* for example, variables in the named trace-buffer structure are updated. Those variables which store attributes are updated when the *%m* conversion specifier is used in the *tracef* routine parameter. When *%m* is not present, the routine applies the attributes currently stored in the *color*, *modifier*, and *enhance* variables.

The second argument in all four of these trace routines is the character, string, or format pointer to the data that will be written to the selected trace buffer.

The *tracef* routine allows you to add attributes to messages on the Program Trace screen and User Traces. These attributes are listed in Table 64-4.

Each trace operation appends output to the end of the trace buffer. You may not use the *pos_cursor* routine to position the cursor on any trace screen. New lines (or blank lines) may be generated via the "\n" nonliteral. Put the "\n" nonliteral at the end of the string to generate a leading blank line on the selected trace screen:

```
tracef(&prog_trbuf, "This trace message will generate a leading blank line.\n");
```

During real-time display, this line moves just ahead of the freshest trace message and continuously overwrites the oldest one. If you put the "\n" sequence at the beginning of the format string, no leading blank line will help you distinguish new messages from the old:

```
tracef(&prog_trbuf, "\nThis message will not generate a leading blank line.");
```

tracec

Synopsis

```
extern void tracec(trace_buffer_ptr, character);
extern struct trace_buf * trace_buffer_ptr;
const char character;
```

Description

The *tracec* routine outputs a single ASCII character to the trace screen indicated.

Inputs

The first parameter is a pointer to the trace buffer into which the character will be written.

For the second parameter, see the *displayc* routine.

Example

In this instance, output will be written to the Program Trace screen.

```
{
 #include <trace_buf.h>
 extern struct trace_buf prog_trbuf;
}
LAYER: 2
      STATE: display_to_prog_tr
         CONDITIONS: KEYBOARD " "
         ACTIONS:
         {
          tracec(&prog_trbuf, 'a');
          tracec(&prog_trbuf, '\n');
          tracec(&prog_trbuf,65);
          tracec(&prog_trbuf, '\n');
          tracec(&prog_trbuf,0x65);
          tracec(&prog_trbuf, '\n');
          tracec(&prog_trbuf,065);
         }
```

When the user views the PROG TR screen, the output will look like this:

a
A
e
5

tracef

Synopsis

```
extern int tracef(trace_buffer_ptr, format_ptr, . . . );
extern struct trace_buf * trace_buffer_ptr;
const char * format_ptr;
```

Description

The *tracef* routine writes output to a specified trace screen, under control of the string, pointed to by *format_ptr*, that specifies how subsequent arguments are converted for output. If there are insufficient arguments for the format, the behavior is undefined. If the format is exhausted while arguments remain, the excess arguments are evaluated but otherwise ignored. The *tracef* routine returns when the end of the format string is encountered.

Inputs

The first parameter is a pointer to the trace buffer into which the output will be written.

For the second parameter, see the *displayf* routine. Placement of "\n" in the format string of a call to *tracef* generates a blank new line on the selected trace screen. (In a *displayf* routine, the newline character does not blank the new line.)

Attributes are written via the *%m* conversion specifier to *trace_buf.hdr.modifier*, *trace_buf.hdr.color*, and *trace_buf.hdr.enhance*. The attributes are copied from these variables into subsequent 32-bit data words in the Program Trace and User Traces. Table 64-4 shows the format of this 32-bit word.

Returns

The *tracef* routine returns the number of characters displayed, or a negative value if the unit is in freeze mode.

Example

This program traces X.29 PAD-control messages in DTE and DCE data packets. The letters "DCE" are underlined in the DCE trace lines.

```
LAYER: 3
{
   #include <trace_buf.h>
   extern struct trace_buf l3_trbuf;
   extern unsigned char * m_packet_info_ptr;
   extern unsigned short m_packet_lcn;
   unsigned char pad_ctrl_msg;
}
   STATE: packet_type
      CONDITIONS: DTE DATA Q= 1
      ACTIONS:
      {
         pad_ctrl_msg = m_packet_info_ptr[0];
         tracef (&l3_trbuf, "DTE LCN:%.3x PAD MSG:%.2x\n", m_packet_lcn,
            pad_ctrl_msg);
      }
      CONDITIONS: DCE DATA Q= 1
      ACTIONS:
      {
         pad_ctrl_msg = m_packet_info_ptr[0];
         tracef (&l3_trbuf, "%mDCE%m LCN:%.3x PAD MSG:%.2x\n", 0x04000000L,
            0x00000000L, m_packet_lcn, pad_ctrl_msg);
      }
```

stracef

Synopsis

```
extern void stracef(array_ptr, string_ptr);
unsigned long array_ptr;
const char * string_ptr;
```

Description

The *stracef* routine is similar to the *tracef* routine, except that *stracef* writes output to a *variable*, while *tracef* writes output to a trace screen. The output is under control of the string pointed to by *string_ptr* that specifies how subsequent arguments are converted for output. If there are insufficient arguments for the

format, the behavior is undefined. If the format is exhausted while arguments remain, the excess arguments are evaluated but otherwise ignored. The *stracef* routine returns when the end of the format string is encountered.

The *stracef* routine differs from *sprintf* in that it generates an array of *longs*, whereas *sprintf* generates an array of *chars*. When the *stracef* array is written to a trace buffer (or to the Display Window) it carries its predefined attributes along with it. An *sprintf* array, by contrast, will receive the attributes that are active in the buffer at the moment.

At the end of the output string, there will be a null character with the Special Character Indicator bit set in its *modifier* attribute-byte.

Inputs

The first parameter is a pointer to the variable into which output will be written. The array which will hold output must be declared as a *long*. By allocating 32 bits for each element, the array may accommodate attributes assigned via the *%m* conversion specifier. Attributes comprise 24 bits of the *long*. The preferred form of the declaration is *unsigned long name [100]*. The size and name of the array are user-determined.

For the second parameter, see the *displayf* routine.

Example

This program traces X.29 PAD-control messages for DTE and DCE data packets. The resulting trace is identical to the one generated by the example under *tracef*. Note that attributes that are turned on in an *stracef* array do not need to be turned off after the array has been brought, via the *%b* conversion specifier, into a *tracef* format string.

```
LAYER: 3
{
   #include <trace_buf.h>
   extern struct trace_buf l3_trbuf;
   extern unsigned char * m_packet_info_ptr;
   extern unsigned short m_packet_lcn;
   unsigned char pad_ctrl_msg;
   unsigned long source[4];
}
   STATE: packet_type
      CONDITIONS: DTE DATA Q= 1
      ACTIONS:
      {
         stracef (source, "%s", "DTE");
      }
      NEXT_STATE: pad_msg_trace
```

```
    CONDITIONS: DCE DATA Q= 1
    ACTIONS:
    {
       stracef (source, "%m%s", 0x04000000L, "DCE");
    }
    NEXT_STATE: pad_msg_trace
  STATE: pad_msg_trace
    CONDITIONS: ENTER_STATE
    ACTIONS:
    {
       pad_ctrl_msg = m_packet_info_ptr[0];
       tracef (&l3_trbuf, "%b LCN:%.3x PAD MSG:%.2x\n", source, m_packet_lcn,
          pad_ctrl_msg);
    }
    NEXT_STATE: packet_type
```

traces

Synopsis

```
extern void traces(trace_buffer_ptr, string_ptr);
extern struct trace_buf trace_buffer_ptr;
const char * string_ptr;
```

Description

The *traces* routine writes output to a specified trace screen, under control of the string that is referenced by *string_ptr*. The *traces* routine returns when the end of the string is encountered.

Inputs

The first parameter is a pointer to the trace buffer into which the output will be written.

For the second parameter, see the *displays* routine.

Example

In this instance, output will be written to the TRACE 1 screen.

The following entry

```
{
#include <trace_buf.h>
extern struct trace_buf l1_trbuf;
}
LAYER: 1
     STATE: any
        CONDITIONS: KEYBOARD " "
        ACTIONS:
        {
         traces(&l1_trbuf, "End of test.");
        }
```

produces the following output on the TRACE 1 trace screen:

End of test.

The following coding produces the same output:

```
{
#include <trace_buf.h>
extern struct trace_buf l1_trbuf;
}
LAYER: 1
     STATE: any
        CONDITIONS: KEYBOARD " "
        ACTIONS:
        {
         const char * string_ptr;
         string_ptr = "End of test.";
         traces (&l1_trbuf, string_ptr);
        }
```

set_utrace_fkey_label

Synopsis

```
extern void set_utrace_fkey_label(trace_buffer, label_ptr);
unsigned int trace_buffer;
const char * label_ptr;
```

Description

Use the *set_utrace_fkey_label* routine to modify the labels which identify the seven user-trace buffers. The default labels are TRACE 1, TRACE 2, TRACE 3, TRACE 4, TRACE 5, TRACE 6, TRACE 7. These labels correspond to the user-trace buffer with the same number. There is no Protocol Spreadsheet softkey equivalent of this routine.

Inputs

The first parameter identifies the user-trace function key whose label is to be replaced. Integers from 1 through 7 are valid values. The buffer number must correspond to a user-trace buffer that is written to in the program. If it does not or if the specified value is out of the valid range, the label is not assigned to any softkey.

The second parameter is a pointer to a null-terminated string, i.e., the label that should replace the current one for the specified trace buffer. The label string has a maximum length of seven characters. If it has fewer than seven characters, it is padded to the right with spaces. If it has more than seven characters, only the first seven are used.

Example

In the following example, new labels are assigned to the softkeys for user-trace buffers 2 and 3. If you press the USER TR softkey in Run mode, the labels TRACE 1 and TRACE 2 should be replaced with FRAME and PACKET.

```
{
 #include <trace_buf.h>
 extern struct trace_buf l2_trbuf;
 extern struct trace_buf l3_trbuf;
}
LAYER: 1
     STATE: define_labels
        CONDITIONS: ENTER_STATE
        ACTIONS:
        {
         set_utrace_fkey_label(2, " FRAME");
         set_utrace_fkey_label(3, "PACKET");
        }
        NEXT_STATE: write_to_buffers
     STATE: write_to_buffers
        CONDITIONS: KEYBOARD "2"
        ACTIONS:
        {
         tracef(&l2_trbuf, "Frame Level Information");
        }
        CONDITIONS: KEYBOARD "3"
        ACTIONS:
        {
         tracef(&l3_trbuf, "Packet Level Information");
        }
```

64.5 Attributes

Attributes are written to the Display Window and to the trace buffers in 32-bit words that include 8 bits of character data (the second-lowest byte) and 24 bits of attributes. The format of the 32-bit data word, given in Table 64-4, is the same for the Display Window and for the trace buffers.

In *displayf* routines, the *%m* conversion specifier writes input to *window_color* and *window_modifier* variables. These variables are then copied into data words written to the Display Window by string pointers in this and subsequent *displayf* routines. See Figure 64-1.

In *tracef* routines, the *%m* conversion specifier writes input to the *trace_buffer_header* structure for a particular user-trace buffer. The header is then copied into each data word written to the particular user buffer by string pointers in this and subsequent *tracef* routines. See Figure 64-2.

(A) Applying Attributes As Data Is Buffered

There are two ways an attribute may be assigned to a character in the Display Window. One way uses the *%m* conversion specifier to assign attributes to the *window_color* and *window_modifier* variables. This program, for example, includes a *displayf* routine that uses the *%m* conversion specifier to write underlined data to the Display Window:

```
STATE: apply_attribute_to_window_color_variable
   CONDITIONS: ENTER_STATE
   ACTIONS:
   {
    pos_cursor (1,0);
    displayf ("%mThis data is underlined in the Display Window.", 0x04000000L);
   }
```

The chart in Table 64-4 shows the hex value 04000000L in the "input" column alongside the underline attribute. This means that when the value 0x04000000L is input to the conversion specifier *%m*, an underline attribute is applied to the current *displayf* string and any that follow until the attribute is turned off. The underline attribute actually is applied to the external *window_color* variable. See Table 64-2. The *window_color* and *window_modifier* variables lend their attributes to every character that is written in a format string to the Display Window. In Run mode if the user presses the softkey for DSP WND, he will see his underlined string. Subsequent characters or strings written to the Display Window also will be underlined.

The same attribute could be applied to a string in any of the user-trace buffers, as follows:

```
{
 #include <trace_buf.h>
 extern struct trace_buf l1_trbuf;
}
      STATE: apply_attribute_to_header
         CONDITIONS: ENTER_STATE
         ACTIONS:
         {
          tracef (&l1_trbuf, "%mThis data is underlined.", 0x04000000L);
         }
```

Only the header for the TRACE 1 display is affected by this *%m* conversion. Only the TRACE 1 buffer is written to. When other trace buffers are subsequently written to, the strings will not receive underlining as a result of the attributes applied above to the TRACE 1 header.

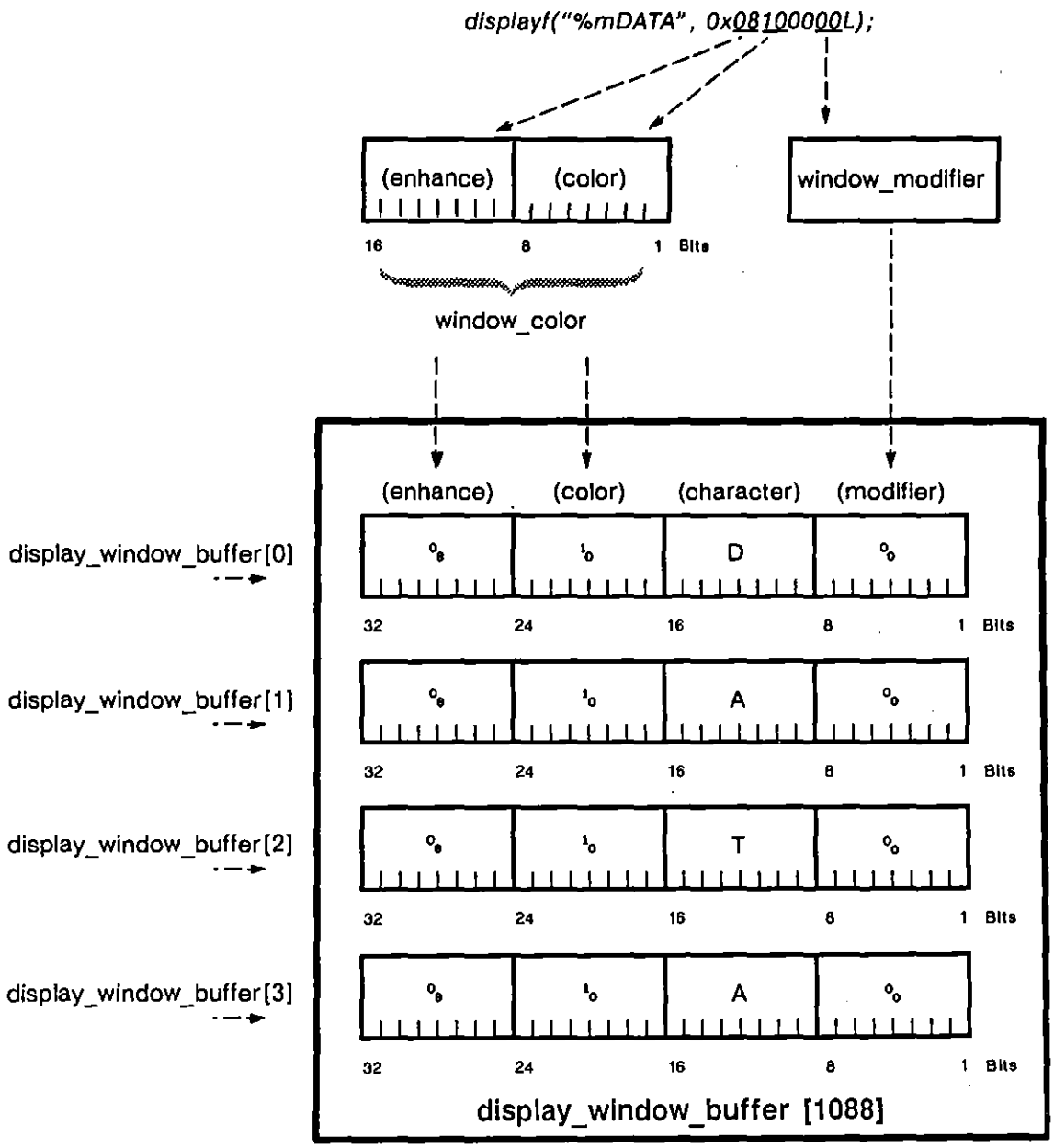

Figure 64-1 When a *displayf* routine is called, the attributes assigned via the *%m* conversion specifier are stored in two *extern* variables, accessible to the user. Both color and enhance attributes are contained in *window_color*. The low byte in *window_color* indicates the color; the high byte contains enhancements. In this example, the following attributes will be assigned to characters written to the Display Window: reverse-image enhancement, green-on-black color, and ASCII font. Before a character is written to the Display Window, it is combined in a *long* with its attributes, as mapped in the figure.

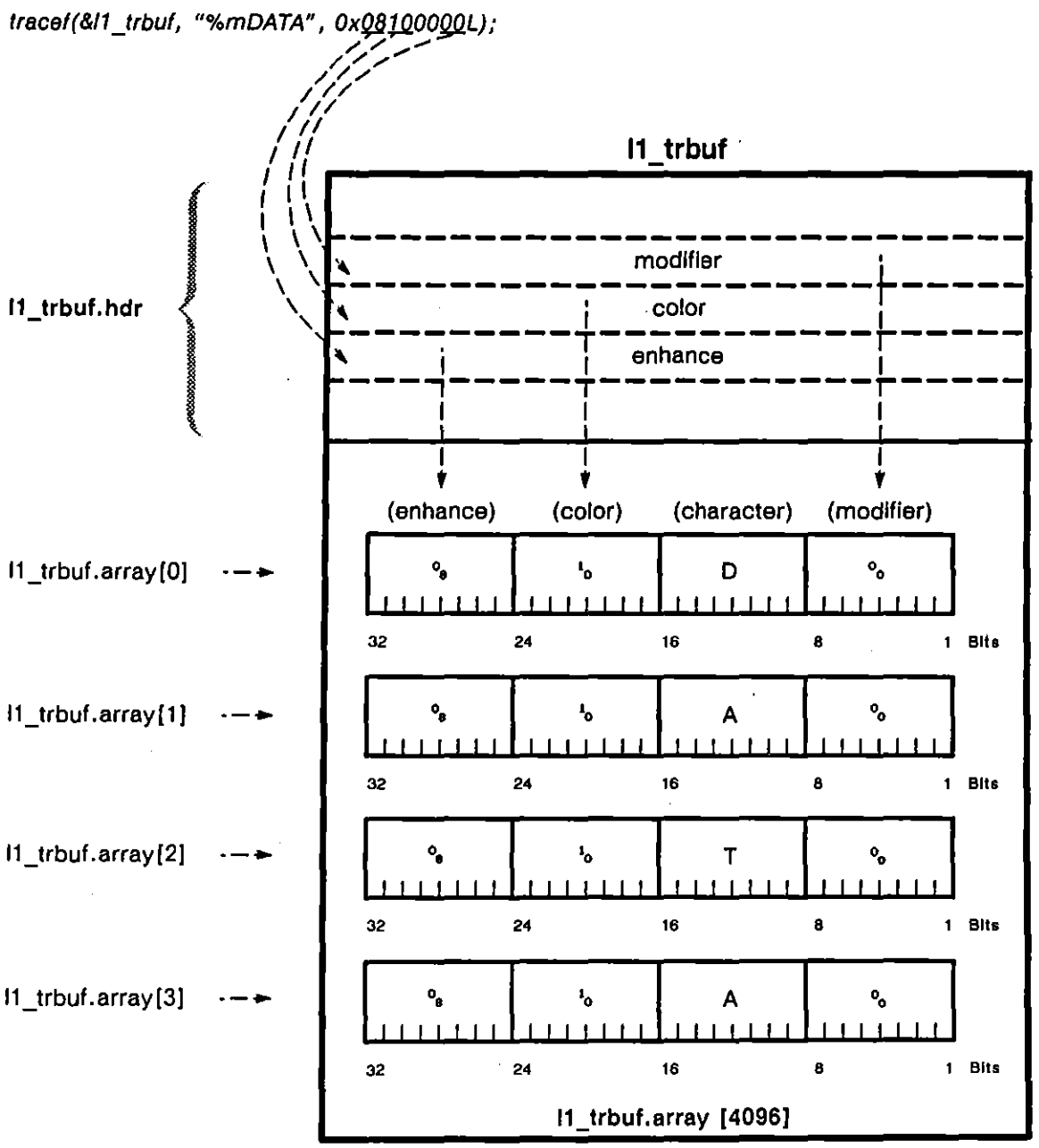

Figure 64-2 When a *tracef* routine is called, the attributes assigned via the *%m* conversion specifier are stored in three variables in the trace-buffer header of a designated buffer. In this example, *l1_trbuf.hdr* holds the following attributes: reverse-image enhancement, green-on-black color, and ASCII font. Before a character is written to the buffer, it is combined in a *long* with its attributes, as mapped in the figure.

(B) Applying Attributes to Buffered Data

The Display Window is an array of 1,088 *long int*egers, each including one byte of character data and three bytes of attributes. The character data is generated by strings in display routines. The attributes for each character are derived from the current *window_color* and *window_modifier* values at the time the character is written to the display-window buffer.

Once the data word is written to the buffer as an element in the array, it can be accessed and written to—and therefore changed—the same as any other location in memory. In the example that follows, a string is written to the Display Window without underlining. Then, as a result of a keyboard input from the operator, the first 32-bit word in the string (containing the first character, the letter "T") is given a new value that includes the underline attribute.

```
{
 extern unsigned long display_window_buffer[1088];
 extern struct
        {
         unsigned char mpm;
         unsigned char cpm;
        }
 display_window_index_buffer[17];
}
     STATE: apply_attribute_directly_to_display_window
        CONDITIONS: ENTER_STATE
        ACTIONS:
        {
         pos_cursor(1,0);
         displayf ("This data is not underlined.");
        }
        CONDITIONS: KEYBOARD " "
        ACTIONS:
        {
         display_window_buffer[64] = ((display_window_buffer[64] & ~0x04000000L) |
           0x04000000L);
         display_window_index_buffer[1].mpm ++;
        }
```

Incrementing *display_window_index_buffer.mpm* is necessary to alert the processor on the CPM card (containing the display-controller software) that the program has changed the contents of the Display Window. Refer to Table 64-3 for an explanation of this structure.

The bitwise *and*ing and *or*ing in the example are necessary if you want to change certain bits in the word without affecting others. Note that the value whose complement (~) is *and*ed with *display_window_buffer* element #64 is the "mask" for the underline attribute in Table 64-4; and the value to the right of the *or* operator (|) is the "input" value for the underline attribute.

Table 64-7
Conversion Specifiers

Specifier	Argument type	Conversion Type
%b	integer-array pointer	array of *long* integers. 2nd byte of each *long* is displayed as character. 1st, 3rd, and 4th bytes interpreted as attributes. Array begins at pointer, ends at element containing null character and Special Character bit = 1.
%l	integer	signed decimal representing 15-bit value
%o	unsigned character	unsigned character
%#c	unsigned character	newline character displayed as ↳ rather than acted on
%d	integer	signed decimal representing 15-bit value
%ld	integer	signed decimal representing 31-bit value
%H	character-array pointer	character array indicated by argument appears as small hex characters. (Precision as to number of characters becomes length of the array, overriding usual null-termination of strings.)
%m	integer	*long* integer not displayed or printed, but written to attribute header-variable for Display Window or for one of the trace buffers
%o	integer	unsigned octal representing 16-bit value
%lo	integer	unsigned octal representing 32-bit value
%#o	integer	unsigned octal representing 16-bit value, preceded by 0
%#lo	integer	unsigned octal representing 32-bit value, preceded by 0
%p	integer	unsigned hexadecimal (lower-case letters) representing 32-bit value, with a minimum 5 digits displayed and a colon between the 4 right-hand digits and the 1-4 left-hand digits. Useful for displaying CPU segment number and offset.
%s	character-array pointer	array of characters beginning at pointer and ending at null terminator or at array-length precision, whichever occurs first
%#s	character-array pointer	newline character displayed as ↳ rather than acted on
%u	integer	unsigned decimal representing 16-bit value
%lu	integer	unsigned decimal representing 32-bit value
%#u	integer	hex *characters* (example: $^{B}{}_{F}{}^{E}{}_{5}$) representing 16-bit value
%#lu	integer	hex *characters* (example: $^{B}{}_{F}{}^{E}{}_{5}{}^{3}{}_{0}{}^{1}{}_{3}$) representing 32-bit value

Table 64-7 (continued)

Specifier	Argument type	Conversion Type
%x	integer	unsigned hexadecimal (lower-case letters) representing 16-bit value
%lx	integer	unsigned hexadecimal (lower-case letters) representing 32-bit value
%#x	integer	unsigned hexadecimal (lower-case letters) representing 16-bit value, preceded by 0x
%#lx	integer	unsigned hexadecimal (lower-case letters) representing 32-bit value, preceded by 0x
%X	integer	unsigned hexadecimal (upper-case letters) representing 16-bit value
%lX	integer	unsigned hexadecimal (upper-case letters) representing 32-bit value
%#X	integer	unsigned hexadecimal (upper-case letters) representing 16-bit value, preceded by 0x
%#lX	integer	unsigned hexadecimal (upper-case letters) representing 32-bit value, preceded by 0x
%\n	none	displays an ↳
%%	none	displays a %

64.6 Protocol Trace Buffers

There are two Protocol Trace buffers, one dedicated to Layer 2 and the other to Layer 3 data. Run-mode softkeys for accessing these traces—PROTOCL, L2TRACE, and L3TRACE—appear when personality packages are loaded in at Layers 2 and 3. The prompt line is not part of these buffers.

The size of each Protocol Trace buffer is 65,536 bytes. Of this total, two bytes are dedicated to the buffer header and two bytes to the trailer. The usable size of a Protocol Trace buffer, therefore, is 65,532 bytes. When a buffer's limit is reached, new characters written to the end of the buffer force out the same number of characters from the beginning of the buffer. In Freeze mode you may scroll through the buffer using the cursor keys.

You cannot write directly to the Protocol Trace buffers. Monitor the position within the buffers, as well as the wrap count, using the variables and structures discussed below.

(A) Variables

The addresses of the variables in Table 64-8 identify the physical location of the beginning and end of each Protocol Trace buffer. The beginning position is at the first data byte in the buffer. The end is just after the last data byte.

Table 64-8
Protocol Trace Buffer Variables

Type	Variable	Value (hex/decimal)	Meaning
extern unsigned char	l2pp_trbuff		First data byte in the Layer 2 Protocol Trace buffer. Address of this variable—segment number plus offset—will indicate the *physical* location of the first data byte, two bytes from the beginning of the buffer. Line Setup configured for emulate or monitor mode.
extern unsigned long	l2pp_trbuff_end		First byte in the two-byte trailer of the Layer 2 Protocol Trace buffer—i.e., after the last data byte. Address of this variable—segment number plus offset—will indicate the *physical* location of the end of the data area, hexadecimal FFFE bytes from the beginning of the buffer. Line Setup configured for emulate or monitor mode.
extern unsigned char	l3pp_trbuff		First data byte in the Layer 3 Protocol Trace buffer. Address of this variable—segment number plus offset—will indicate the *physical* location of the first data byte, two bytes from the beginning of the buffer. Line Setup configured for emulate or monitor mode.
extern unsigned long	l3pp_trbuff_end		First byte in the two-byte trailer of the Layer 3 Protocol Trace buffer—i.e., after the last data byte. Address of this variable—segment number plus offset—will indicate the *physical* location of the end of the data area, hexadecimal FFFE bytes from the beginning of the buffer. Line Setup configured for emulate or monitor mode.

(B) Structures

The structure variables in Table 64-9 contain the high and low bytes of a beginning and ending offset and wrap-count in the Layer 2 and Layer 3 Protocol Trace buffers. Create a logical beginning (or ending) offset within a buffer by combining the two offset-variables relating to a beginning (or ending) position into a single, two-byte offset. Add the resulting offset to the address of *l3_trbuff* to identify the *physical* address of a *logical* location.

The example below uses *#define* preprocessor directives for determining beginning and ending offsets in the Layer 3 Protocol Trace buffer. When *get_l3pp_value_end* is encountered in a program, for example, each of the two "end" offset-variables is cast into a *long* and, if necessary, shifted left to its appropriate position in an offset. Then the two variables are added together.

```
#define get_l3pp_value_begin
(((unsigned long)(l3pp_trbuff_ctl.begin_off_hi) << 8) +
((unsigned long)(l3pp_trbuff_ctl.begin _off_lo)))


#define get_l3pp_value_end
(((unsigned long)(l3pp_trbuff_ctl.end_off_hi) << 8) +
((unsigned long)(l3pp_trbuff_ctl.end_off_lo)))
```

When the ending offset, in this example, is added to the address of *l3_trbuff*, the result is the address of the *logical* end in the buffer:

```
unsigned long end_address;
end_address = &l3_trbuff + get_l3pp_value_end;
```

You may also use the offsets and wrap counts to determine how much data is currently in the buffer. Include the wrap count in the high two bytes of a four-byte offset. Then subtract the beginning offset from the ending offset.

```
#define get_l3pp_value_begin
(((unsigned long)(l3pp_trbuff_ctl.begin_wrap_hi) << 24) +
((unsigned long)(l3pp_trbuff_ctl.begin_wrap_lo) << 16) +
((unsigned long)(l3pp_trbuff_ctl.begin_off_hi) << 8) +
((unsigned long)(l3pp_trbuff_ctl.begin_off_lo)))

#define get_l3pp_value_end
(((unsigned long)(l3pp_trbuff_ctl.end_wrap_hi) << 24) +
((unsigned long)(l3pp_trbuff_ctl.end_wrap_lo) << 16) +
((unsigned long)(l3pp_trbuff_ctl.end_off_hi) << 8) +
((unsigned long)(l3pp_trbuff_ctl.end_off_lo)))

unsigned long end, begin, count;
end = get_l3pp_value_end;
begin = get_l3pp_value_begin;
count = end - begin;
```

Table 64-9
Protocol Trace Buffer Structures

Type	Variable	Value (hex/decimal)	Meaning
Structure Name:	**lpp_trbuff_ctl**		Declared as type *struct*. The variables contained in this structure monitor logical location in a Protocol Trace buffer. Reference structure variables as follows: *lpp_trbuff_ctl.begin_off_hl*.
unsigned char	begin_off_hl	*0-ffl0-255*	High byte of a 2-byte offset from the *physical* beginning of the Protocol Trace buffer to a *logical* beginning in the buffer. Range of the two-byte offset is 2 through hexadecimal FFFE.
unsigned char	begin_off_lo	*0-ffl0-255*	Low byte of a 2-byte offset from the *physical* beginning of the Protocol Trace buffer to a *logical* beginning in the buffer. Range of the two-byte offset is 2 through hexadecimal FFFE.
unsigned char	begin_wrap_hl	*0-ffl0-255*	High byte of a 2-byte count of the number of times a *logical* beginning has wrapped through the Protocol Trace buffer.
unsigned char	begin_wrap_lo	*0-ffl0-255*	Low byte of a 2-byte count of the number of times a *logical* beginning has wrapped through the Protocol Trace buffer. It will have a value of zero only once. Once the count reaches hexadecimal FFFF, it will wrap to one.
unsigned char	end_off_hl	*0-ffl0-255*	High byte of a 2-byte offset from the *physical* beginning of the Protocol Trace buffer to a *logical* end in the buffer. Range of the two-byte offset is 2 through hexadecimal FFFE.
unsigned char	end_off_lo	*0-ffl0-255*	Low byte of a 2-byte offset from the *physical* beginning of the Protocol Trace buffer to a *logical* end in the buffer. Range of the two-byte offset is 2 through hexadecimal FFFE.
unsigned char	end_wrap_hl	*0-ffl0-255*	High byte of a 2-byte count of the number of times a *logical* end has wrapped through the Protocol Trace buffer.
unsigned char	end_wrap_lo	*0-ffl0-255*	Low byte of a 2-byte count of the number of times a *logical* end has wrapped through the Protocol Trace buffer. It will have a value of zero only once. Once the count reaches hexadecimal FFFF, it will wrap to one.
Structure Name:	**l2pp_trbuff_ctl**		An instance of the *lpp_trbuff_ctl* structure, declared as type *extern struct lpp_trbuff_ctl*. The variables contained in this structure monitor logical location in the Layer 2 Protocol Trace buffer. Has the same structure as *lpp_trbuff_ctl*. Reference structure variables as follows: *l2pp_trbuff_ctl.begin_off_h*.

Table 64-9 (continued)

Type	Variable	Value (hex/decimal)	Meaning
Structure Name:	**l3pp_trbuff_ctl**		An instance of the *lpp_trbuff_ctl* structure, declared as type *extern struct lpp_trbuff_ctl*. The variables contained in this structure monitor logical location in the Layer 3 Protocol Trace buffer. Has the same structure as *lpp_trbuff_ctl*. Reference structure variables as follows: *l3pp_trbuff_ctl.begin_off_h*.

(C) Routines

The *set_ltrace_fkey_label* routine allows the programmer to modify the current softkey labels for the Layer 2 and 3 Protocol Traces. There is no Protocol Spreadsheet softkey equivalent of this routine.

set_ltrace_fkey_label

Synopsis

```
extern void set_ltrace_fkey_label(layer, label_ptr);
unsigned int layer;
const char * label_ptr;
```

Description

Use the *set_ltrace_fkey_label* routine to modify the labels which identify the two Protocol Trace buffers. The default labels are L2TRACE and L3TRACE. These labels correspond to the Layer 2 and 3 Protocol Traces.

Inputs

The first parameter identifies the Protocol Trace function key whose label is to be replaced. Integers from 1 through 7 are valid values. The number must correspond to a layer package which is currently loaded into the INTERVIEW. If it does not or if the specified value is out of the valid range, the label is not assigned to any softkey.

The second parameter is a pointer to a null-terminated string, i.e., the label that should replace the current one for the specified Protocol Trace. The label string has a maximum length of seven characters. If it has fewer than seven characters, it is padded to the right with spaces. If it has more than seven characters, only the first seven are used.

Example

In the following example, the X.25 Layer 2 and Layer 3 protocol packages have been loaded via the Layer Setup screen. New labels are assigned to the softkeys for both Protocol Traces. If you press the PROTOCL softkey in Run mode, the labels L2TRACE and L3TRACE should be replaced with X25 FRM and X25 PKT.

```
LAYER: 1
    STATE: define_labels
        CONDITIONS: ENTER_STATE
        ACTIONS:
        {
         set_ltrace_fkey_label(2, "X25 FRM");
         set_ltrace_fkey_label(3, "X25 PKT");
        }
```

65 Counters, Timers, and Accumulators

65.1 Counters

The translator declares the following structure for counters that are entered as softkey tokens on the Protocol Spreadsheet:

```
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;
};
struct counter_struct counter_name={0,0,0,-0ul};
```

The first eight counter variables in the structure are used to calculate statistical values whenever the counter is sampled. See Table 65-1. Three of the variables—*counter_name.current*, *counter_name.prev*, and *counter_name.old*—come into play each time the counter is incremented, decremented, or set to a particular value.

Counters are internal program variables and counter interrupts are strictly program-generated signals, so the C programmer is free to ignore this structure and maintain counts and statistics in a different way. Please note, however, that the 68010 CPU expects this counter structure when it polls the 80286 periodically for statistical values to display in columns on the tabular and graphic stats screens.

(A) Current, Previous, and Old Values

When a counter is incremented, decremented, or set to a specific value on the Protocol Spreadsheet, the program does not signal a *counter_name_change* interrupt automatically. First it verifies that the new value of the counter really is a *change* from the previous value. See Table 65-2. For this comparison, the program needs to maintain two variables, *counter_name.current* and *counter_name.prev*.

Table 65-1
Counter Structures

Type	Variable	Meaning
Structure Name:	**counter_struct**	Structure of a counter. Declared as type *struct*. Declared automatically if a program counter is used. Program counters assigned to structure as follows: struct counter_struct counter_*name*. Reference a structure variable as follows: counter_*name*.current.
unsigned long	current	This value of the counter is acted on directly by program actions.
unsigned long	last	Last sampled value; displayed on the tabular statistics screen.
unsigned long	maximum	Maximum value of all samples; displayed on the tabular statistics screen.
unsigned long	minimum	Minimum value of all samples; displayed on the tabular statistics screen. Should be initialized as -0ul.
unsigned short	sample_count	Number of samples.
unsigned long	total_high	High four bytes of an eight-byte counter total.
unsigned short	total_low_low	Low two bytes of an eight-byte counter total. This two-byte variable counts to 65,535.
unsigned short	total_low_high	Bytes 3 and 4 of an eight-byte counter total.
unsigned short	out_of_range	Number is out of range, either incremented beyond the range or decremented below 0; should not be factored into averages.
unsigned short	changed	For future use.
unsigned long	prev	When converting a counter action to C, the translator compares *prev* with *current* to determine whether counter has changed. Then *prev* is updated to *current* and *counter*_name_*change* is signaled.
unsigned long	old	When converting a counter condition to C, the translator compares *old* with *current* to determine whether true condition is new (transitional). After program logic has examined counter, *old* is updated to *prev*.

Here, for example, is the C translation of the simple action COUNTER example SET 5.

```
counter_example.current = 5;
if (counter_example.prev != counter_example.current)
  {
    counter_example.old = counter_example.prev;
    counter_example.prev = counter_example.current;
    signal (counter_example_change);
  }
```

Table 65-2
Counter Variables

Type	Variable	Meaning
extern event	counter_*name*_change	True when the named counter is incremented, decremented, or set to new value. This event will not be triggered unless a spreadsheet *condition* names the counter. Line Setup configured for emulate or monitor mode.

It is clear from the translation that the variable *counter_example.prev* is used to limit the number of *counter_example_change* interrupts to those cases where the current value of the counter really has changed.

What is *counter_name.old* used for? We will preface the answer by citing the behavior of the counter in the following spreadsheet example.

```
STATE: threshold_condition
   CONDITIONS: KEYBOARD " "
   ACTIONS: COUNTER spacebar INC
   CONDITIONS: COUNTER spacebar GE 7
   ACTIONS: ALARM
```

Each time you press the space bar while this program is running, the counter will increment, but no matter how many times you press the space bar the alarm will only sound once. It will sound on the seventh keystroke, the *first time* the counter is greater than or equal to 7. If the program had a decrement or set action that lowered the counter to less than 7, the alarm would sound again when the counter reached the 7 threshold.

The translator accomplishes this threshold condition by coding the *waitfor* clause as follows:

```
counter_spacebar_change && (! (counter_spacebar.old >= 7)) && (counter_spacebar.current >= 7):
```

Since *counter_spacebar.prev* was used (and then updated to "current") in the *if* statement that sent the *counter_spacebar_change* interrupt, the "old" value is required in the *waitfor* condition to insure a "transitional" or "threshold" counter condition.

(B) Sampling a Counter

Here is the translator's version of a counter sample action:

```
counter_name.last = counter_name.current;
if (counter_name.current > counter_name.maximum)
  {
    counter_name.maximum = counter_name.current;
  }
if (counter_name.current < counter_name.minimum)
  {
    counter_name.minimum = counter_name.current;
  }
counter_name.sample_count++;
{
  unsigned long temp;
  temp = (counter_name.current & 0x0000ffff) + counter_name.total_low_low;
  counter_name.total_low_low = temp;
  temp = (counter_name.current >> 16) + counter_name.total_low_high + (temp >> 16);
  counter_name.total_low_high = temp;
  counter_name.total_high += temp >> 16;
}
counter_name.current = 0;
```

In order to establish an average value for all samples, a grand total for current values at the time of each sampling must be maintained. Since an ordinary INTERVIEW current counter is 32 bits, the counter that maintains the grand total of current counts must be larger (64 bits). There is no data type this large in C, and so the "total" counter is distributed among three variables and the somewhat complicated coding involving the *temp* variable is required to add the current counter to this composite counter.

(C) Updating the Statistics Screen

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.

In order to know where on the statistics screens the values for the particular counters (and timers and accumulators) should be placed, the 68010 CPU on the CPM needs some help from the program (that is, from the MPM). This help is in the form of a "stat message" that the translator (or the programmer) codes once at the beginning of the program. The stat message is a structure that the MPM sends to the CPM. See Table 65-3. The stat message says, in effect, "Here is the address of a counter structure. When you access this structure during the running of the program in order to pull out the current, last, maximum, minimum, total, and sample-count values, display those values on the row of the tabular stats screen where the user has typed *spacebar*" (for example).

Table 65-3
Counter, Timer, and Accumulator Structures

Type	Variable	Value (hex/decimal)	Meaning
Structure Name:	**stat_msg**		Structure of a stat message. A stat message is sent once for each named counter, timer, or accumulator. Declared as type struct. Declared automatically if a softkey-entered COUNTER is used as a condition, or if softkey-entered COUNTER, TIMER, or ACCUMUL action is taken. Program stat messages assigned to structure as follows: struct stat_msg *name*. You must assign values to the elements of the structure. Reference a structure variable as follows: *name*.type.
unsigned short	op_type	0a00/2560	Register statistics objects from the MPM to the CPM. Other values and meanings for future use.
unsigned short	type	0 0100/256 0200/512	accumulator counter timer
unsigned long	object_name		The MPM (80286) address of a counter, timer, or accumulator name, converted to CPM (68010) format. To get an object_name address, enter: *name*.object_name = get_68k_phys_addr("*name_of_counter*");
unsigned long	object_address		The MPM (80286) address of a counter, timer, or accumulator structure, converted to CPM (68010) format. To get a structure address for a counter, enter: *name*.object_address = get_68k_phys_addr(&counter_*name_of_counter*);

Here is a C program that causes the current value of a counter named "key" to increment on the tabular-statistics screen each time an ASCII-keyboard key is struck.

```
{
 struct
   {
    unsigned short op_type;
    unsigned short type;
    unsigned long object_name;
    unsigned long object_address;
   } stat_msg;
 extern unsigned long get_68k_phys_addr();
```

```
    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;
      };
    struct counter_structure counter_key;
    extern fast_event keyboard_new_key;
}
    STATE: update_stat_screen
        {
          stat_msg.op_type = 2560;
          stat_msg.type = 256;
          stat_msg.object_name = get_68k_phys_addr("key");
          stat_msg.object_address = get_68k_phys_addr(&counter_key);
          send_stat_message(&stat_msg);
          waitfor
            {
              keyboard_new_key:
                {
                  counter_key.current++;
                }
            }
        }
```

The variable *stat_msg.object_name* is a pointer to the name of the counter that the user has entered on the protocol spreadsheet. The program gives this name to the CPM, and expects the CPM to locate the name among the names that the user has entered on the tabular or graphic statistics menu. The delivery to the CPM of a pointer to the stats-menu name and a pointer to the counter structure is the purpose of the stat message. The message allows the CPM to correlate a line on the statistics results screen with an actual program counter (or timer or accumulator).

NOTE TO C PROGRAMMERS: When the translator creates a counter variable it adds the prefix *counter_* to the spreadsheet name, but the programmer who is working primarily in C and is not making use of spreadsheet counters can name the counter any way he wishes, with or without the prefix. Similarly, the string that is communicated to the CPM in *stat_msg.object_name* ("key" in the example above) must agree with the name on the stats menu, but it need not bear any resemblance to the name of the counter structure.

NOTE ALSO: In most of the examples in this 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. *get_68k_phys_addr()* returns a *long*, however, and must be declared.

65.2 Timers

The translator declares the following structure for timers that are entered as softkey tokens on the Protocol Spreadsheet:

```
struct timer_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 long start_tick_value;
  unsigned short running;
  unsigned short changed;
};
```

There are no timer conditions in the software (since timeouts provide the time-triggering function), and therefore all of the variables in the structure serve as data for the CPM when it updates the stats screens. See Table 65-4. A stat message must be sent so the CPM can correlate a line on the statistics results screen with the correct program timer. The stat message is documented in the previous section on counters. The timer stat message is different only in respect that the *stat_msg.type* element should be set to 512 instead of 256.

Timer restart, continue, and stop actions are explained in this section. The clear action is simply a matter of changing the elements in the structure to zero (except for *timer_name.minimum*, which becomes the one's complement of zero).

(A) Time Ticks

Time ticks are timed increments of either of two hardware counters in the INTERVIEW. The programmer can select which of the two timing mechanisms to use for a given timer.

One tick-counter is on the FEB card and is used to time-stamp incoming data and EIA leads. The intervals between ticks is determined on the FEB Setup menu. Ticks can be enabled/disabled on the same menu. The current value of this counter is available in a variable called *11_tick_count*. See Table 65-5. The current value always reflects the number of ticks since the program entered Run mode. The number of ticks may or may not equate to the amount of time in Run mode, since ticks are also encoded in playback data and the playback rate is subject to "local conditions" such as playback speed and idle suppression.

FEB time ticks are the most precise timing mechanism in that they have a resolution to 10 microseconds. They also represent the most durable method of timekeeping, since they preserve the original data timings even during playback.

Table 65-4
Timer Structures

Type	Variable	Value (hex/decimal)	Meaning
Structure Name:	**timer_struct**		Structure of a timer. Declared as type *struct*. Declared automatically if a program timer is used. Program timers assigned to structure as follows: struct timer_struct timer_*name*. Reference a structure variable as follows: timer_*name*.current.
unsigned long	current		Current value of timer, not updated while timer is running. Values are in microseconds rounded to tick-unit on FEB Setup screen.
unsigned long	last		Value of last sample; displayed on the tabular statistics screen.
unsigned long	maximum		Maximum value of all samples; displayed on the tabular statistics screen.
unsigned long	minimum		Minimum value of all samples; displayed on the tabular statistics screen. Should be initialized as -0ul.
unsigned short	sample_count		Number of samples.
unsigned long	total_high		High four bytes of an eight-byte timer total.
unsigned short	total_low_low		Low two bytes of an eight-byte timer total.
unsigned short	total_low_high		Bytes 3 and 4 of an eight-byte timer total.
unsigned long	start_tick_value		Tick-count in microseconds when timer was started, restarted, or continued. For line-related conditions at Layer 1, this value is stored in *l1_tick_count*; for non-line conditions, use *get_wall_time_286_ticks* routine.
unsigned short	running	0	Stopped. This variable is polled and a zero stops the timer from incrementing and sets the current value to *timer_name.current* (understood as microseconds).
		-0	Running. All 1's in this variable causes the timer to increment, showing a value that equals (wall-time ticks - *timer_name.start_tick_value*) + *timer_name.current*.
unsigned short	changed	-0	For future use.

Table 65-5
Timer Variables

Type	Variable	Meaning
extern unsigned long	l1_tick_count	This variable counts ticks from the start of Run mode. Tick=sec, msec, etc., depending on FEB setup. Subtract early value from later value to create a timer. ACTIONS: *{ displayf (" %ld msecs ", (l1_tick_count - timer_name.start_tick_value));}* Add to *start_of_run_time* to determine more precise current time for time-stamping events. Line Setup configured for emulate or monitor mode.
extern unsigned long	start_of_run_date	Date when Run mode entered. Byte 1 (low byte) indicates day; byte 2 stores month; and bytes 3 and 4 indicates year. May be used to time-stamp events. See also *start_of_run_time*. Line Setup configured for emulate or monitor mode.
extern unsigned long	start_of_run_time	Time when Run mode entered. Byte 1 (low byte) indicates seconds; byte 2 stores minutes; and byte 3 indicates hours. May be used to time-stamp events. See also *start_of_run_date* and *l1_tick_count*. † Line Setup configured for emulate or monitor mode.

† In the example below, the *displayf* (or *tracef*) routine uses timer variables to time-stamp good BCCs on the DCE side. (Similar programming could determine the current date.) The tick unit selected on the FEB Setup menu is seconds. Adjust the program as needed for other tick units.

```
{
 extern unsigned long start_of_run_date, start_of_run_time, l1_tick_count;
 unsigned short seconds, hours, minutes, tick_mins, tick_secs, tick_hours;
 #define SECS(run_time) (unsigned short)(run_time & 0xff)
 #define MINS(run_time) ((unsigned short)(run_time >> 8) & 0xff)
}
      STATE: time
         CONDITIONS: DCE GOOD_BCC
         ACTIONS:
         {
           tick_secs = l1_tick_count % 60;
           tick_mins = (l1_tick_count + SECS(start_of_run_time)) / 60;
           tick_hours = (tick_mins + MINS(start_of_run_time)) / 60;
           displayf("Time: %.2d:%.2d:%.2d\n",
                  (unsigned short)(((start_of_run_time >> 16) & 0xff) + tick_hours)%24,
                  (MINS(start_of_run_time) + tick_mins)%60,
                  (SECS(start_of_run_time) + tick_secs)%60);
         }
```

The other tick-counter is on the MPM and is referred to as the wall-time clock. This clock ticks once per millisecond and drives the timers displayed on the statistics results screens—at least while they are incrementing. At the moment a timer stops incrementing, the programmer can reach in and replace the incremented value with a timer value based the FEB tick-counter instead.

The current value of this wall-time tick-counter is available to the program via the *get_wall_time_286_ticks* routine. The current value always reflects both the number of ticks and the actual elapsed time ("wall time") since the program entered Run mode.

(B) Running

While it increments on the stats screen, a timer always is driven by wall-time ticks. To start a current timer incrementing, first you must have sent a stat message to correlate the timer structure with a timer line on the stats screen. At that point the simple statement *timer_name.running = -0* will start the timer. The value of the timer at any given time while it is running will be the MPM (wall-time) ticks minus the *timer_name.start_tick_value* plus any *timer_name.current* value.

To stop a timer, change *timer_name.running* to zero. The current column of the timer will immediately display the value of *timer_name.current* (zero, unless you have done something in your program to calculate the current value of the timer). The stats display will interpret *timer_name.current* as a value in microseconds and convert it to the unit selected for that timer line.

(C) Restart

The translator has two different versions of the timer restart action, depending on what condition precipitated the action. The first version is used if the condition was data-related (or EIA-related) *and* time ticks are enabled on the FEB Setup menu. Here is this data-timer version:

```
unsigned long temp;
convert_tick_count (l1_tick_count, &temp);
timer_name.current = 0;
timer_name.start_tick_value = temp;
timer_name.running = -0;
```

The *convert_tick_count* routine converts *l1_tick_count* into microseconds and stores the result in *temp*. The value of *temp* is assigned immediately to *timer_name.start_tick_value*. When the 68010 sees that *timer_name.running* equals the one's complement of zero, it subtracts the start-tick value from the l1-tick count and displays the difference in the current column of the timer line. Since the start-tick value was derived a moment before from the l1-tick count, the difference will be zero. The current column on the stats screen should begin a timer at zero following a restart.

A slightly different version of the program is used if the condition was nondata-related or if time ticks are disabled in the FEB. The *convert_tick_count* routine is not used and the following routine is used in its place:

```
get_wall_time_286_ticks (&temp);
```

This routine returns the current value of the wall-time tick-counter, in milliseconds zero-padded to microseconds. It stores the value in *temp* and the program proceeds as above.

(D) Continue

The timer-continue action is very similar to the restart. There are just two differences. One, the action is enclosed in an if statement that verifies that *timer_name.running* equals zero—that the timer actually is stopped, in other words; and two, *timer_name.current* is not set to zero, but retains the value it received the last time the timer stopped.

(E) Stop

Here is one of the two versions of a timer stop action:

```
if (timer_name.running != 0)
  {
   unsigned long temp;
   convert_tick_count (11_tick_count, &temp);
   timer_name.current += temp - timer_name.start_tick_value;
   timer_name.running = 0;
  }
```

In this translation, the start-tick value is subtracted from the current tick count, and any pending current value (held over if the timer was continued) is added in. The result is a new *timer_name.current* value. This value is posted to the stats screen as soon as the 68010 sees *timer_name.running = 0*.

The other version of the stop action uses *get_wall_time_286_ticks* instead of *convert_tick_count*.

(F) Sample Action

The code that produces the sample action is identical to the code that sampled a counter. See Section 65.1(B). The *timer_name.sample_count* variable's not equaling zero causes minimum, maximum, and average values to be displayed.

65.3 Accumulators

Shown below is the structure of an accumulator as the translator declares it (and as the 68010 accesses it to update the statistics screens). Also refer to Table 65-6. Note that there is no current value, since an accumulator neither counts nor times. There are no "previous" and "old" values, because in its spreadsheet implementation an accumulator never is tested in a Conditions block.

```
struct accumulator_struct
{
  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 changed;
};
struct accumulator_struct accumulator_name={0,0,-0ul};
```

Here is the translator's version of an accumulate action when the object of the accumulation (selected by the user) was the maximum sampled value of a counter named *framechar*.

```
accumulator_name.last = accumulator_framechar.maximum;
if (accumulator_name.last > accumulator_name.maximum)
  {
    accumulator_name.maximum = accumulator_name.last;
  }
if (accumulator_name.last < accumulator_name.minimum)
  {
    accumulator_name.minimum = accumulator_name.last;
  }


accumulator_name.sample_count++;
  {
    unsigned long temp;
    temp = (accumulator_name.last & 0x0000ffff) + accumulator_name.total_low_low;
    accumulator_name.total_low_low = temp;
    temp = (accumulator_name.last >> 16) + accumulator_name.total_low_high + (temp >> 16);
    accumulator_name.total_low_high = temp;
    accumulator_name.total_high += temp >> 16;
  }
accumulator_name.changed = -0;
```

A stat message must be sent so the CPM can correlate a line on the statistics results screen with the correct accumulator. The stat message is documented in the previous section on counters. The accumulator stat message is different only in respect that the *stat_msg.type* element should be set to 0 instead of 256.

The *accumulator_name.sample_count* variable's not equaling zero causes minimum, maximum, and average values to be displayed.

Table 65-6
Accumulator Structures

Type	Variable	Meaning
Structure Name:	**accumulator_struct**	Structure of an accumulator. Declared as type *struct*. Declared automatically by program when the user softkey-enters an ACCUMULATE action. Specific accumulator assigned to structure as follows: struct accumulator_struct accumulator_*name*. Reference a structure variable as follows: accumulator_*name*.last.
unsigned long	last	Value of last sample; displayed on the tabular statistics screen.
unsigned long	maximum	Maximum value of all samples; displayed on the tabular statistics screen.
unsigned long	minimum	Minimum value of all samples; displayed on the tabular statistics screen. Should be initialized as ~0ul.
unsigned short	sample_count	Number of samples.
unsigned long	total_high	High four bytes of an eight-byte accumulator total.
unsigned short	total_low_low	Low two bytes of an eight-byte accumulator total.
unsigned short	total_low_high	Bytes 3 and 4 of an eight-byte accumulator total.
unsigned short	changed	For future use.

65.4 Routines

get_68k_phys_addr

Synopsis

```
extern unsigned long get_68k_phys_addr(variable_ptr);
unsigned char * variable_ptr;
```

Description

This routine converts the address of a specified variable in the 80286 processors (MPM boards) to 68010 (CPM) format. This routine must be declared.

Inputs

The only parameter is the address to be converted.

Returns

The *get_68k_phys_addr* routine returns the converted address.

Example

See *send_stat_message* routine.

send_stat_message

Synopsis

```
extern void send_stat_message(struct_stat_msg_ptr);
struct stat_msg
  {
   unsigned short op_type;
   unsigned short type;
   unsigned long object_name;
   unsigned long object_address;
  );
struct stat_msg * struct_stat_msg_ptr;
```

Description

The *send_stat_message* routine sends the stat message structure to the 68010 CPU (CPM board). The current use of this routine sends the addresses of program counters, timers, and accumulators in the 80286 processors (MPM boards) to the CPM board where the tabular and graphic statistics displays are located.

The routine is called only one time in a program for each named counter, timer, or accumulator. Entering COUNTER as a condition or action (or TIMER or ACCUMUL as actions) via softkey on the Protocol Spreadsheet automatically declares the counter named and sends the stat message.

Inputs

The only parameter is a pointer to the structure of the stat message. For an explanation of the elements of the stat message, see Table 65-3.

Example

You plan on incrementing a counter named "dte_info" when a DTE Info frame is detected.

```
{
 struct
  {
   unsigned short op_type;
   unsigned short type;
   unsigned long object_name;
   unsigned long object_address;
  } stat_msg;
```

```
  struct counter_structure
  {
   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;
  };
 struct counter_structure counter_dte_info = {0, 0, 0, -0ul};
 extern unsigned long get_68k_phys_addr();
}
LAYER: 2
      STATE:send_stat_message
         CONDITIONS: ENTER_STATE
         ACTIONS:
         {
          stat_msg.op_type = 2560;
          stat_msg.type = 256;
          stat_msg.object_name = get_68k_phys_addr("dte_info");
          stat_msg.object_address = get_68k_phys_addr(&counter_dte_info);
          send_stat_message(&stat_msg);
         }
         NEXT_STATE: count_info
      STATE: count_info
         CONDITIONS: DTE INFO
         ACTIONS:
         {
          counter_dte_info.current++;
         }
```

get_wall_time_ticks

Synopsis

```
extern void get_wall_time_ticks(ticks_68k_format_ptr);
unsigned long * ticks_68k_format_ptr;
```

Description

The *get_wall_time_ticks* routine gets the number of wall-time ticks (in CPM storage format) from the time RUN was hit. The wall clock gives millisecond resolution rounded to microseconds.

Inputs

The only input is a pointer to the location where the returned time-tick value will be stored.

Example

```
{
 unsigned long ticks;
}
LAYER: 2
     STATE: get_ticks
        CONDITIONS: KEYBOARD " "
        ACTIONS:
        {
         get_wall_time_ticks(&ticks);
        }
```

get_wall_time_286_ticks

Synopsis

```
extern void get_wall_time_286_ticks(ticks_286_format_ptr);
unsigned long * ticks_286_format_ptr;
```

Description

The *get_wall_time_286_ticks* routine gets the number of wall-time ticks (in MPM storage format) from the time RVR was hit. The wall clock gives millisecond readings rounded to microseconds. Use this routine prior to setting the *start_tick_value* in a timer action when **Time Ticks:** OFF has been selected on the Front-End Buffer Setup screen. Also use this routine to derive the *start_tick_value* if the condition is not line-related, e.g., KEYBOARD, even when time ticks are enabled on the FEB Setup menu.

Inputs

The only input is a pointer to the location where the returned time-tick value will be stored.

Example

```
{
 unsigned long ticks_286;
}
LAYER: 3
     STATE: get_ticks
        CONDITIONS: KEYBOARD " "
        ACTIONS:
        {
         get_wall_time_286_ticks(&ticks_286);
         displayf ("%lu", ticks_286);
        }
```

convert_tick_count

Synopsis

```
extern void convert_tick_count(mpm_format_ticks, converted_ticks_ptr);
unsigned long mpm_format_ticks;
unsigned long * converted_ticks_ptr;
```

Description

The *convert_tick_count* routine converts a designated tick count into microseconds.

Use this routine to derive the *start_tick_value* for a timer action if ticks are enabled on the FEB Setup menu and the condition is line-related, e.g., RCV INFO.

Inputs

The first parameter is a designated tick count as long as it is in MPM storage format. It may be any of the layer tick counts. The unit of the *l1_tick_count* (and other layers' tick counts) value is determined on the Front End Buffer menu.

The second parameter is a pointer to the location where the returned tick count converted to microseconds will be stored.

Example

```
{
 extern unsigned long l1_tick_count;
 unsigned long converted_ticks;
}
LAYER: 1
     STATE: convert_ticks
        CONDITIONS: RECEIVE GOOD_BCC
        ACTIONS:
        {
         convert_tick_count(l1_tick_count, &converted_ticks);
         displayf ("%lu", converted_ticks);
        }
```

66 OSI

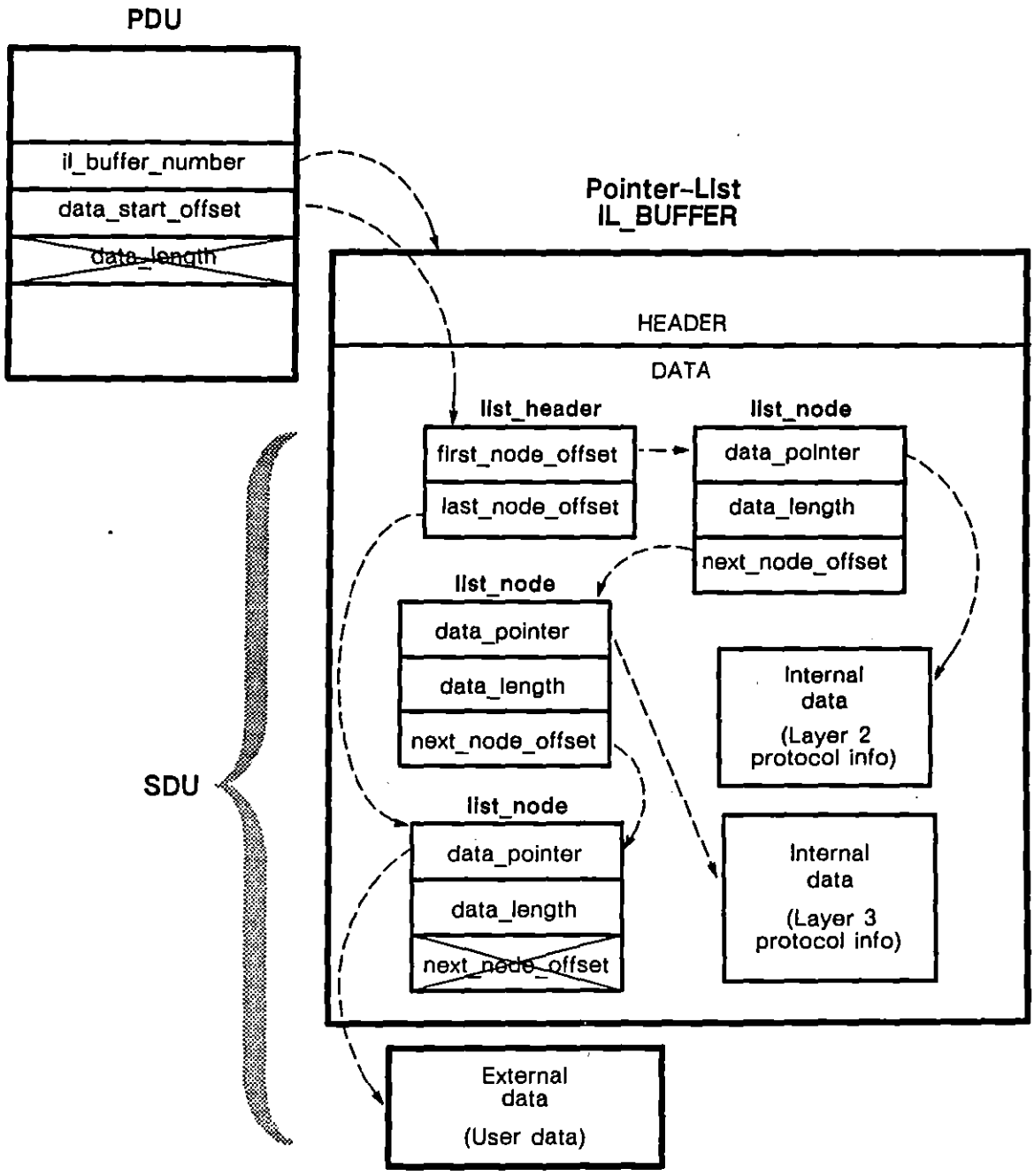

Figure 66-1 Primitive Data Unit and sample Pointer-List Buffer being passed *down* the layers.

66 OSI

The most convenient tools for handling protocol headers while data is moving down and up the layers in the INTERVIEW are the spreadsheet SEND and GIVE_DATA actions in the various protocol packages. For instances when a protocol package is not loaded, such as when you are developing a new protocol or simply using a protocol that is not yet an option on the Layer Setup screen, OSI structures, variables, and routines in C become essential tools also.

66.1 Structures

The programmer may access the information in primitive data units conveniently by using a C structure as a multibyte pointer that is superimposed on data in the PDU's. Before using a structure-pointer, it is necessary to understand the contents of IL buffers and primitive data units. All structures referenced may be found in Table 66-1.

(A) Interlayer Message Buffers

1. *Configuring the number/size of IL buffers.* By default, there are a maximum sixteen IL buffers in use at a given time. Each buffer's size is 4,096 bytes. You may change the number and size of the interlayer (IL) buffers. The IL BUFS softkey on the Protocol Spreadsheet presents seven number/size combinations that allocate 64 Kbytes of RAM to IL buffers. See Section 27. In addition to these softkey selections, there are two C preprocessor directives the programmer may use to reconfigure the number and/or size of IL buffers:

   (a) *#pragma il_buffers* sets the number of IL buffers that will be available at a given time. Following the directive, enter a space and then a decimal integer within the range 4 through 255. In the following example, the number of buffers is set to 25:

   #pragma il_buffers 25

   The specified number of buffers will override the number selection on the Interlayer Buffers menu. The buffer size indicated on the Interlayer Buffers menu will remain unchanged, however, unless altered via the *#pragma il_buffer_size* directive.

(b) *#pragma il_buffer_size* sets the size of IL buffers. Following the directive, enter a space and then a decimal integer within the range 33 through 65535. These values include the 32-byte buffer header. (See Section 2. below.) In the following example, the size of buffers is set to 8 Kbytes:

#pragma il_buffer_size 8192

The specified buffer size will override the size selection on the Interlayer Buffers menu. The number of buffers indicated on the Interlayer Buffers menu will remain unchanged, however, unless altered via the *#pragma il_buffers* directive.

Be careful when you are passing messages down from higher layers that you do not make the buffer size too small. Even small messages require a buffer large enough to accommodate the overhead of linked lists.

These two directives provide the programmer with more flexibility in configuring IL buffers than the Protocol Spreadsheet softkeys. With the *#pragmas*, the available RAM for IL buffers may exceed the 64-Kbyte threshold of the IL BUFS selections.

The memory required for IL buffers is the product of the number and size of the buffers (number * size). If this amount exceeds available memory, your program will not compile and the message *"Error 219: Out of memory during compilation — program too big"* will be displayed. Available memory for IL buffers varies depending on the complexity of your program.

2. *IL buffer components.* IL buffers may be one of two kinds: data-character or pointer-list. In buffers being passed up the layers, data-character buffers (Figure 66-2) are always used. In buffers going down the layers, pointer-list buffers (Figure 66-1) are primarily used. The difference is that pointer-list buffers contain list-nodes which provide information about the location of data (or "lists") inserted or referenced in the buffer, while data-character buffers do not.

   (a) *Header.* Each IL buffer contains a header that stores useful information such as the status of the maintain bits that prevent the buffer from being returned to the general pool; the position of the buffered data in the INTERVIEW's display buffer; and the tick count (time) when the data was buffered from the line. (See *il_buffer* structure.)

   (b) *Service Data Unit.* The IL buffer also contains the data itself. This data component, the service data unit (or "SDU"), is added to as the buffer is passed down the layers, and subtracted from as a buffer travels up the layers. *A data-character IL buffer includes all the data that was present when the data was first buffered*, and the contents of this buffer do not change as the buffer is passed up the layers. What changes is the service data unit, derived from the data-start offset in the PDU.

The first part of the SDU in a pointer-list buffer is a *list-header node* (structure *il_list_header*) which contains information about the location of the first and last text nodes. As a buffer is passed down from Layer 3 to Layer 2 in X.25 (see Figure 66-1), a new text node containing a Layer 3 protocol header is inserted in buffer. Since the Layer 3 data will precede user data, the list node for the protocol information is referenced ahead of any other list nodes, changing the first-node reference in the list header. (If text is appended to the end of existing data, the list node referenced as last will change.)

The SDU in a pointer-list buffer also includes *list nodes* (structure *il_list_node*) which give a pointer to data, the length of the data pointed to, and the offset from the start of the buffer to the *next* list node.

Finally, the service data unit in all buffers includes *data*, whether copied into the buffer (usually protocol information) or located in memory outside of the buffer (usually user data).

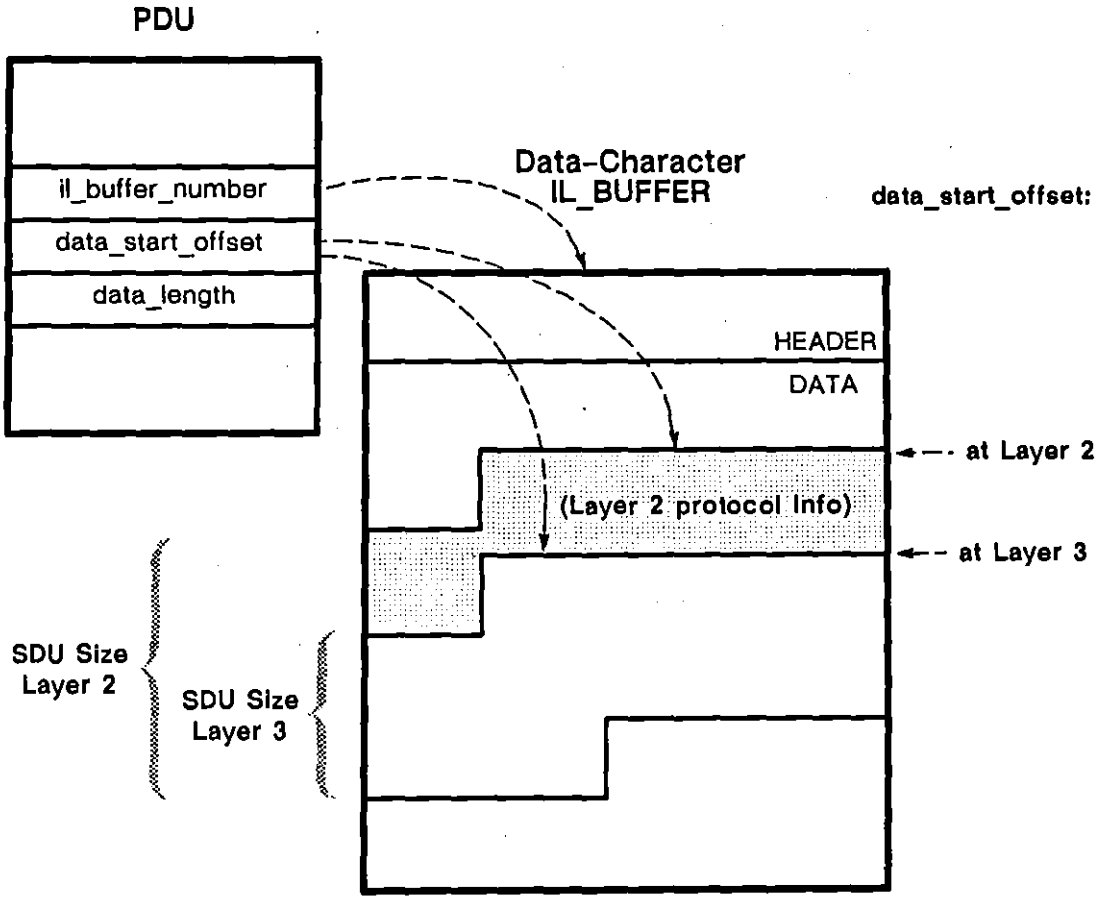

Figure 66-2 Primitive Data Unit and sample Data-Character Buffer being passed *up* the layers.

(B) Primitive Data Units

Like interlayer message buffers, PDU's have a format that is dependent on which direction the primitive is being passed. Refer again to Figure 66-1 and Figure 66-2.

1. *IL buffer number.* The buffer number to be passed with the primitive is always stored in the primitive. This buffer number is actually an 80286-processor segment number.

2. *Data-start offset.* The offset from the beginning of the buffer to the beginning of the service data unit for a given layer is different for the two types of buffers. In a pointer-list buffer going down the layers, the data-start offset will indicate the offset from the beginning of the buffer to the list-header node. This offset will vary if different linked lists have been started at different layers. Each list will have its own list header. In a data-character buffer going up the layers, the data-start offset will change from layer to layer. For example, a buffer containing X.25 data that is being passed from Layer 2 to Layer 3 will have an offset at Layer 3 two bytes beyond the offset at Layer 2.

3. *Data length.* The size of the SDU in a data-character buffer also varies from layer to layer. In the example just given, the SDU will be smaller by two bytes at Layer 3 than it was at Layer 2. In pointer-list buffers, the length of all data is unknown at any given layer.

(C) Accessing Information in Structures

There are two stages that are preliminary to accessing the information in these structures. The first step is to convert the 80286-processor segment number into a 32-bit address. The second stage is to place a pointer, in the shape of an IL buffer structure, at that address. Let's use an IL buffer as an example.

1. *Converting a segment number.* The IL-buffer segment number is returned any time you access one of the external, protocol-independent *il_buffer* variables listed in Table 66-1. These variables have names like *m_lo_dl_il_buff* and *up_n_il_buff*.

   To make a pointer to an IL buffer, (1) shift the 80286 segment number to the left sixteen bits, since a full address in the 80286 is 32 bits long; (2) cast it as a *long*, so that the segment number is in the high 16 bits and the offset to a buffer for that segment is zero (the low 16 bits); and (3) cast it as a pointer. The following expression will take care of all three requirements:

   ```
   (void *) ((long) m_lo_dl_il_buff <<16);
   ```

   Now you have a pointer to the first memory location of the most recent monitor-mode IL buffer passed up from Layer 2 to Layer 3. An upward-moving IL buffer was illustrated in Figure 66-2. The precise structure of both the IL buffer is given in the following declaration.

```
{
 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];
   };
}
```

2. *Create a structure-pointer at a given address.* First, declare the structure of *il_buffer*, as indicated above. Then declare *il_buffer_pointer* as a structure-pointer, as follows:

```
struct il_buffer * il_buffer_pointer;
```

Converting the segment number and assigning it to *il_buffer_pointer* may be accomplished with this one statement:

```
il_buffer_pointer = (void *) ((long) m_lo_dl_il_buff <<16);
```

Now a structure has been created around the most recent upward-moving IL buffer at Layer 3. 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 *il_buffer_pointer->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 *il_buffer_pointer->data[0]*. Here is a program that displays on the prompt line the fifth data element, the packet-type byte, in every IL buffer that is monitored at Layer 3.

```
{
  extern event m_lo_dl_prmtv;
  extern volatile unsigned short m_lo_dl_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 * il_buffer_pointer;
}
LAYER: 3
     STATE: monitor_ll_buffers
        CONDITIONS:
        {
         m_lo_dl_prmtv
        }
        ACTIONS:
        {
          il_buffer_pointer = (void *) ((long) m_lo_dl_il_buff <<16);
          pos_cursor (0,0);
         displayf ("%02x ", il_buffer_pointer->data[4]);
        }
```

If you run this program, be sure to load in the Layer 2 and Layer 3 personality packages for X.25. These packages will take care of delivery of the monitor primitives to Layer 3.

Table 66-1
OSI Structures

Type	Variable	Value (hex/decimal)	Meaning
Structure Name: pdu			Structure of an OSI primitive data unit (PDU). Declared as type *struct*. Use this structure as follows. Declare the entire structure. Make a pointer to a PDU by shifting *m*_lo_*dl*_pdu_seg (or up_*n*_pdu_seg) 16 bits to the left. Then convert this pointer to a pointer to a PDU structure: struct pdu * pdu_pointer pdu_pointer =(void *)((long)m_lo_dl_pdu_seg << 16). Reference a structure-pointer variable as follows: pdu_pointer->primitive_code.
unsigned char	primitive_code		Codes for OSI variables are listed in Table 66-2 through Table 66-8. For Layer 3 primitive codes, for example, refer to Table 66-4. The value of this variable is also stored in external variable *m*_lo_*dl*_prmtv_code (or up_*n*_prmtv_code).
unsigned char	path	*0-8*	Path number, both directions. The value of this variable is also stored in external variable *m*_lo_*dl*_prmtv_path (or up_*n*_prmtv_path).
unsigned long	parameter		For future use. At present, under user control.
unsigned short	relay_baton		Maintain bit passed with an interlayer-message buffer, both directions. Zero in this variable identifies maintain bit.
unsigned short	ll_buffer_number		Segment number of the interlayer-message buffer, both directions. The value of this variable is also stored in external variable *m*_lo_*dl*_ll_buff (or up_*n*_ll_buff).
unsigned char	buffer_contents	0	Contains data-character buffer type. Must be used for buffer being passed up.
		1	Contains pointer-list buffer type. May be used for buffers being passed up, but is currently used primarily for buffers being passed down.
unsigned short	data_start_offset		Offset from the beginning of the buffer to the header node in the SDU of an interlayer-message buffer in an OSI primitive being sent down from a layer above. In a primitive being sent up from a layer below, it is the offset to the SDU. Varies according to the layer at which the buffer is located. For example, in a buffer passed up to Layer 3 from Layer 2, the offset would be to the beginning of the Layer 3 header, bypassing Layer 2 header information. The value of this variable is also stored in external variable *m*_lo_*dl*_sdu_*offset* (or up_*n*_sdu).
unsigned short	data_length		Length of the service data unit, including headers and user data. Only for primitives sent up from layer below. Varies with the layer where the buffer is located. For example, at Layer 3, length would exclude Layer 2 header (or trailer) information. The value of this variable is also stored in external variable m_lo_*dl*_sdu_size.

Table 66-1 (continued)

Type	Variable	Value (hex/decimal)	Meaning
Structure Name:	**il_buffer**		Structure of an interlayer-message buffer, both directions. Declared as type *struct*. Use this structure as follows. Declare the entire structure. Make a pointer to an il_buffer by shifting *m_lo_dl_*il_buff (or up_*n*_il_buff) 16 bits to the left: il_buffer_pointer = (void *)((long)(lo_dl_il_buff << 16). Then convert this pointer to a pointer to an il_buffer structure: struct il_buffer * il_buffer_pointer. Reference a structure-pointer variable as follows: il_buffer_pointer->tick_count_low.
unsigned short	lock	0	Internal variable which prevents structure from being updated by more than one program at the same time.
unsigned short	maintain_bits		Two-byte variable which provides the status of the maintain bits. A bit with a value of 1 is in use.
unsigned short	buffer_size	1000/4096 *21-ffff/33-65535*	default value Specific value depends on buffer size set via IL_BUFFERS programming block or #pragma il_buffer_size
unsigned short	transmit_tag		Bits 1-3 define bcc indication:
		0	no bcc
		1	good bcc
		2	bad bcc
		3	abort
		4	half bad bcc (DDCMP)
			Bits 4-8 for future use.
unsigned short	receive_tag		Bits 1-3 define bcc indication:
		0	no bcc
		1	good bcc
		2	bad bcc
		3	abort
		4	half bad bcc (DDCMP)
			Bit 4 identifies side of the line:
		0	td
		1	rd
			Bit 5—message buffer overflow:
		0	frame fits in buffer
		1	frame too large for the buffer
			Bits 6-8 for future use.
unsigned long	char_buff_frame_start		Location in the character buffer of the start of the buffered data.
unsigned long	char_buff_frame_end		Location in the character buffer of the end of the buffered data.

(il_buffer structure continued on next page)

Table 66-1 (continued)

Type	Variable	Value (hex/decimal)	Meaning
il_buffer (continued)			
unsigned short	tick_count_high		Value of internal variable that counts the number of times *l1_tick_count* has reached its maximum value. Together, the three *ll_buffer* tick-count variables preserve at each layer the original time when the end of the data (BCC) was clocked into the buffer.
unsigned short	tick_count_mid		16 high-order bits of 32-bit *l1_tick_count*.
unsigned short	tick_count_low		16 low-order bits of 32-bit *l1_tick_count*.
unsigned short	available_space_offset		Offset to the next available space in the interlayer-message buffer.
unsigned short	bytes_remaining		Available number of bytes remaining in the buffer.
unsigned long	bcc_indicator	0	reserved
unsigned char	data [4064]		Contains all data including each layer's header information, as well as the first of two block check characters. Does not vary from layer to layer. Default size is 4064, but may range from 33-65535 (hex 21-ffff) depending on the buffer size set via IL_BUFFERS programming block or #pragma ll_buffer_size.
Structure Name: ll_list_header			Structure of the header node in an interlayer-message buffer. Only for primitives sent down from the layer above. Declared as type *struct*. Use this structure as follows. Declare the entire structure. Make a pointer to an ll_list_header by shifting up_*n*_ll_buff (or *m*_lo_*dl*_ll_buff) 16 bits to the left and adding the data_start_offset from the PDU structure (also stored as external variable up_*n*_sdu or *m*_lo_*dl*_sdu_*offset*): ll_list_header_pointer = (void *)(((long)up_n_ll_buff) << 16) + up_n_sdu). Then convert this pointer into a pointer to an ll_list_header structure: struct ll_list_header * ll_list_header_pointer. Reference a structure-pointer variable as follows: ll_list_header_pointer->last_node_offset.
unsigned short	first_node_offset		Offset from the beginning of the buffer to the first text node in the buffer. Varies according to the layer at which the buffer is located. At Layer 2, the offset would be to different starting node than at Layer 3.
unsigned short	last_node_offset		Offset to the location of the last text node in the buffer, from the beginning of the buffer.
unsigned long	reserved		reserved

Table 66-1 (continued)

Type	Variable	Value (hex/decimal)	Meaning
Structure Name:	**ll_list_node**		Structure of text nodes in an interlayer-message buffer. Only for primitives sent down from the layer above. Declared as type *struct*. Use this structure as follows. Declare the entire structure. Make a pointer to an ll_list_node by shifting up_*n*_ll_buff (or *m_lo_dl*_ll_buff) 16 bits to the left and adding the first_node_offset (or last_node_offset) from the ll_list_header structure: ll_list_node_pointer = (void *)(((long)up_n_ll_buff << 16) + ll_list_header_pointer->first_node_offset). Point to the next node as follows: next_node_pointer = (ll_list_node_pointer + ll_list_node_pointer->next_node_offset).
unsigned char *	data_pointer		Pointer to the data in a text node.
unsigned short	data_length		Length of the data in a text node.
unsigned short	next_node_offset		Offset to the location of the next text node in the buffer, from the beginning of the buffer. Generally, there is a text node for each layer's header information and one for the user data. A buffer that started at Layer 3 would have two text nodes, one for Layer 3 header information and one for user data (if any). At Layer 2, the buffer would acquire an additional text node.

66.2 Variables

OSI variables are layer-specific. The information stored in the OSI variables may be obtained by using the structure-pointer to IL buffers and primitives. But rather than requiring the user to repeat this process at each layer as a buffer moves through the layers, monitor and emulate variables have been made available at Layers 2–7 to store layer-specific, as well as general, information: the interlayer-buffer number, the offset to the service data unit, the path number, the size of the SDU, the segment number of the PDU, etc. There are also event variables which indicate that a primitive has been received at a given layer. Table 66-2 through Table 66-8 give the current OSI variables and their meanings.

The exchange of connect primitives shown primarily in Figure 33-4 is demonstrated in Figure 66-3 using C variables and routines. The SEND actions insert data in a buffer and send the buffer in a DATA REQ primitive. See Section 66.3 for an explanation of the *_insert_il_buff_list_cnt* and *send* primitive routines. The conditions use event variables to detect primitives and non-event variables to identify specific primitive types.

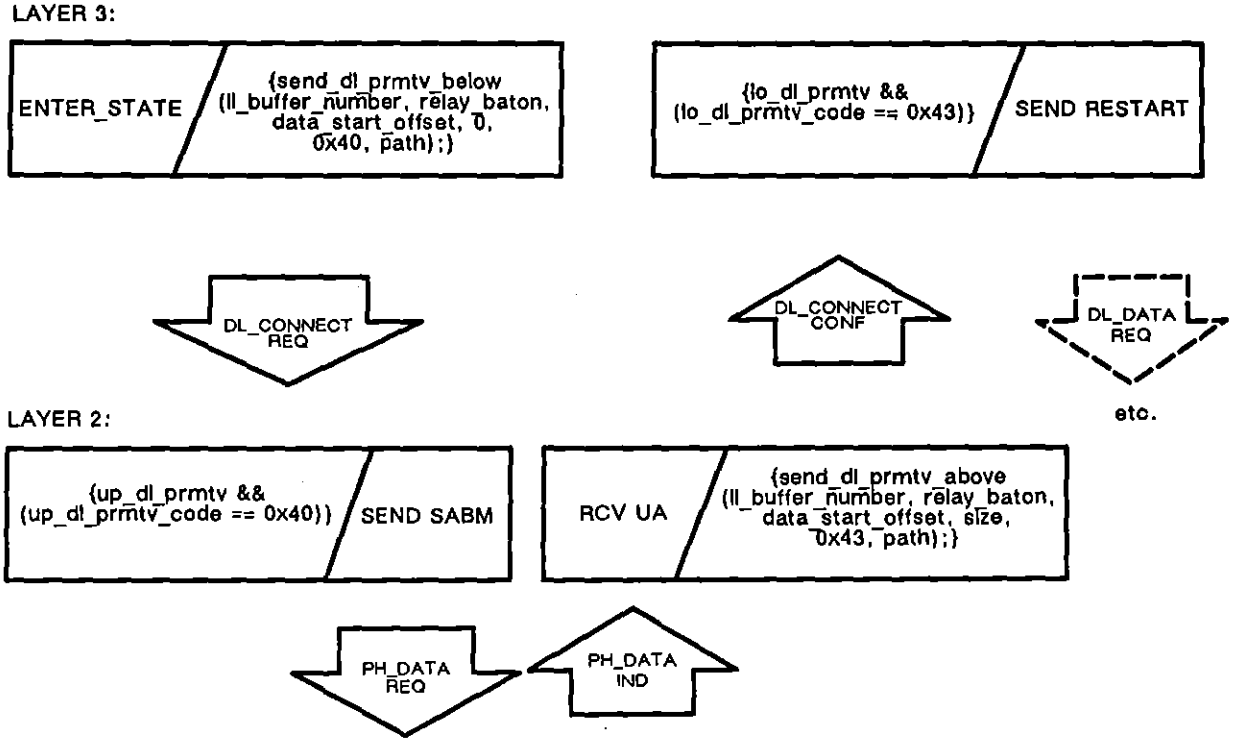

Figure 66-3 Layer 3 uses connect primitives to be sure that the Layer 2 entity below has established a link.

Table 66-2
Layer 1 OSI Variables

Type	Variable	Value (hex/decimal)	Meaning
extern volatile unsigned char	ph_prmtv_type	20/32	ph activate req
		21/33	ph activate ind
		22/34	ph activate resp
		23/35	ph activate conf
		24/36	ph data req
		25/37	ph data ind
		2a/42	ph reset req
		2b/43	ph reset ind
		2c/44	ph reset resp
		2d/45	ph reset conf
		2e/46	ph deactivate req
		2f/47	ph deactivate ind
		30/48	ph debug req
		31/49	ph debug ind
		33/51	ph error report ind
		34/52	ph xmit req
		35/53	ph set idle req
		38/56	ph mgt facility req
		39/57	ph mgt facility ind
			OSI primitive code for primitives moving between Layers 1 and 2. Line Setup configured for emulate mode only.

Table 66-3
Layer 2 OSI Variables

Type	Variable	Value (hex/decimal)	Meaning
extern event	lo_ph_prmtv		True when an OSI primitive is received at Layer 2 from Layer 1. Line Setup configured for emulate mode only.
extern event	m_lo_ph_prmtv		True when an OSI primitive is received at Layer 2 from Layer 1. Line Setup configured for emulate or monitor mode.
extern event	up_dl_prmtv		True when an OSI primitive is received at Layer 2 from Layer 3. Line Setup configured for emulate mode only.
extern volatile unsigned short	lo_ph_pdu_seg		OSI primitive data unit (PDU) iAPX-286 segment number received at Layer 2 from Layer 1. This segment number can be converted to a pointer by shifting it left 16 bits. Line Setup configured for emulate mode only.
extern volatile unsigned short	m_lo_ph_pdu_seg		OSI primitive data unit (PDU) iAPX-286 segment number received at Layer 2 from Layer 1. This segment number can be converted to a pointer by shifting it left 16 bits. Line Setup configured for emulate or monitor mode.
extern volatile const unsigned char	lo_ph_prmtv_code	21/33 23/35 25/37 2b/43 2d/45 2f/47 31/49 33/51 39/57	ph activate ind ph activate conf ph data ind ph reset ind ph reset conf ph deactivate ind ph debug ind ph error report ind ph mgt facility ind OSI primitive code received at Layer 2 in a PDU from Layer 1. Line Setup configured for emulate mode only.
extern volatile const unsigned char	m_lo_ph_prmtv_code	24/36 25/37	td ph data ind rd ph data ind OSI primitive code received at Layer 2 in a PDU from Layer 1. Line Setup configured for emulate or monitor mode.

Table 66-3 (continued)

Type	Variable	Value (hex/decimal)	Meaning
extern volatile const unsigned char	lo_ph_prmtv_path	*0-8*	Path number received at Layer 2 in a PDU from Layer 1. Line Setup configured for emulate mode only.
extern volatile const unsigned char	m_lo_ph_prmtv_path	*0-8*	Path number received at Layer 2 in a PDU from Layer 1. Line Setup configured for emulate or monitor mode.
extern volatile unsigned short	lo_ph_il_buff		Interlayer-buffer number (an iAPX-286 segment number) received at Layer 2 in a PDU from Layer 1. This segment number can be converted to a pointer by shifting it left 16 bits. Line Setup configured for emulate mode only.
extern volatile unsigned short	m_lo_ph_il_buff		Interlayer-buffer number (an iAPX-286 segment number) received at Layer 2 in a PDU from Layer 1. This segment number can be converted to a pointer by shifting it left 16 bits. Line Setup configured for emulate or monitor mode.
extern volatile unsigned short	lo_ph_sdu		In OSI primitive received at Layer 2 from Layer 1, the offset to where the service data unit begins. Line Setup configured for emulate mode only.
extern volatile unsigned short	m_lo_ph_sdu_offset		In OSI primitive received at Layer 2 from Layer 1, the offset to where the service data unit begins. Line Setup configured for emulate or monitor mode.
extern volatile unsigned short	m_lo_ph_sdu_size		Size of the service data unit in an interlayer-message buffer, displayed as SIZE on the Layer 2 trace screen. Received at Layer 2 from Layer 1. Same as *data_length* in a PDU. Line Setup configured for emulate or monitor mode.
extern volatile unsigned short	up_di_pdu_seg		OSI primitive data unit (PDU) iAPX-286 segment number received at Layer 2 from Layer 3. This segment number can be converted to a pointer by shifting it left 16 bits. Line Setup configured for emulate mode only.

Table 66-3 (continued)

Type	Variable	Value (hex/decimal)	Meaning
extern volatile const unsigned char	up_dl_prmtv_code	40/64 42/66 44/68 48/72 4a/74 4c/76 4e/78 50/80 52/82 58/88	dl conn req dl conn resp dl data req dl expd data req dl reset req dl reset resp dl disconn req dl debug req dl unit data req dl mgt facility req OSI primitive code received at Layer 2 in a PDU from Layer 3. Line Setup configured for emulate mode only.
extern volatile const unsigned char	up_dl_prmtv_path	*0-8*	Path number received at Layer 2 in a PDU from Layer 3. Line Setup configured for emulate mode only.
extern volatile unsigned short	up_dl_il_buff		Interlayer-buffer number (an iAPX-286 segment number) received at Layer 2 in a PDU from Layer 3. This segment number can be converted to a pointer by shifting it left 16 bits. Line Setup configured for emulate mode only.
extern volatile unsigned short	up_dl_sdu		Offset to the start (header node) of the service data unit in an interlayer-message buffer. Received at Layer 2 from Layer 3. Same as *data_start_offset* in a PDU. Line Setup configured for emulate mode only.
extern unsigned long	l2_tick_count		32-bit *l1_tick_count* stored in header of most recent IL buffer passed up to Layer 2. Preserves at each layer the original time when the end of the data (BCC) was clocked into the buffer. Line Setup configured for emulate or monitor mode.

Table 66-4
Layer 3 OSI Variables

Type	Variable	Value (hex/decimal)	Meaning
extern event	lo_dl_prmtv		True when an OSI primitive is received at Layer 3 from Layer 2. Line Setup configured for emulate mode only.
extern event	m_lo_dl_prmtv		True when an OSI primitive is received at Layer 3 from Layer 2. Line Setup configured for emulate or monitor mode.
extern event	up_n_prmtv		True when an OSI primitive is received at Layer 3 from Layer 4. Line Setup configured for emulate mode only.
extern volatile unsigned short	lo_dl_pdu_seg		OSI primitive data unit (PDU) IAPX-286 segment number received at Layer 3 from Layer 2. This segment number can be converted to a pointer by shifting it left 16 bits. Line Setup configured for emulate mode only.
extern volatile unsigned short	m_lo_dl_pdu_seg		OSI primitive data unit (PDU) IAPX-286 segment number received at Layer 3 from Layer 2. This segment number can be converted to a pointer by shifting it left 16 bits. Line Setup configured for emulate or monitor mode.
extern volatile const unsigned char	lo_dl_prmtv_code	41/65 43/67 45/69 49/73 4b/75 4d/77 4f/79 51/81 53/83 55/85 59/89	dl conn ind dl conn conf dl data ind dl expd data ind dl reset ind dl reset conf dl disconn ind dl debug ind dl unit data ind dl error report ind dl mgt facility ind
			OSI primitive code received at Layer 3 in a PDU from Layer 2. Line Setup configured for emulate mode only.

Table 66-4 (continued)

Type	Variable	Value (hex/decimal)	Meaning
extern volatile const unsigned char	m_lo_dl_prmtv_code	44/68 45/69 48/72 49/73 54/84 55/85	td dl data ind rd dl data ind td dl expd data ind rd dl expd data ind td dl unit data ind rd dl unit data ind OSI primitive code received at Layer 3 in a PDU from Layer 2. Line Setup configured for emulate or monitor mode.
extern volatile const unsigned char	lo_dl_prmtv_path	*0-8*	Path number received at Layer 3 in a PDU from Layer 2. Line Setup configured for emulate mode only.
extern volatile const unsigned char	m_lo_dl_prmtv_path	*0-8*	Path number received at Layer 3 in a PDU from Layer 2. Line Setup configured for emulate or monitor mode.
extern volatile unsigned short	lo_dl_il_buff		Interlayer-buffer number (an iAPX-286 segment number) received at Layer 3 in a PDU from Layer 2. This segment number can be converted to a pointer by shifting it left 16 bits. Line Setup configured for emulate mode only.
extern volatile unsigned short	m_lo_dl_il_buff		Interlayer-buffer number (an iAPX-286 segment number) received at Layer 3 in a PDU from Layer 2. This segment number can be converted to a pointer by shifting it left 16 bits. Line Setup configured for emulate or monitor mode.
extern volatile unsigned short	lo_dl_sdu		In OSI primitive received at Layer 3 from Layer 2, the offset to where the service data unit begins. Line Setup configured for emulate mode only.
extern volatile unsigned short	m_lo_dl_sdu_offset		In OSI primitive received at Layer 3 from Layer 2, the offset to where the service data unit begins. Line Setup configured for emulate or monitor mode.
extern volatile unsigned short	m_lo_dl_sdu_size		Size of the service data unit in an interlayer-message buffer, displayed as SIZE on the Layer 3 trace screen. Received at Layer 3 from Layer 2. Same as *data_length* in a PDU. Line Setup configured for emulate or monitor mode.

Table 66-4 (continued)

Type	Variable	Value (hex/decimal)	Meaning
extern volatile unsigned short	up_n_pdu_seg		OSI primitive data unit (PDU) IAPX-286 segment number received at Layer 3 from Layer 4. This segment number can be converted to a pointer by shifting it left 16 bits. Line Setup configured for emulate mode only.
extern volatile const unsigned char	up_n_prmtv_code	60/96 62/98 64/100 66/102 68/104 6a/106 6c/108 6e/110 70/112 72/114 74/116 76/118 78/120	n conn req n conn resp n data req n data ack req n expd data req n reset req n reset resp n disconn req n debug req n unit data req n qual data req n qual data ack req n mgt facility req
			OSI primitive code received at Layer 3 in a PDU from Layer 4. Line Setup configured for emulate mode only.
extern volatile const unsigned char	up_n_prmtv_path	*0-8*	Path number received at Layer 3 in a PDU from Layer 4. Line Setup configured for emulate mode only.
extern volatile unsigned short	up_n_il_buff		Interlayer-buffer number (an IAPX-286 segment number) received at Layer 3 in a PDU from Layer 4. This segment number can be converted to a pointer by shifting it left 16 bits. Line Setup configured for emulate mode only.
extern volatile unsigned short	up_n_sdu		Offset to the start (header node) of the service data unit in an interlayer-message buffer. Received at Layer 3 from Layer 4. Same as *data_start_offset* in a PDU. Line Setup configured for emulate mode only.
extern unsigned long	l3_tick_count		32-bit *l1_tick_count* stored in header of most recent IL buffer passed up to Layer 3. Preserves at each layer the original time when the end of the data (BCC) was clocked into the buffer. Line Setup configured for emulate or monitor mode.

Table 66-5
Layer 4 OSI Variables

Type	Variable	Value (hex/decimal)	Meaning
extern event	lo_n_prmtv		True when an OSI primitive is received at Layer 4 from Layer 3. Line Setup configured for emulate mode only.
extern event	m_lo_n_prmtv		True when an OSI primitive is received at Layer 4 from Layer 3. Line Setup configured for emulate or monitor mode.
extern event	up_t_prmtv		True when an OSI primitive is received at Layer 4 from Layer 5. Line Setup configured for emulate mode only.
extern volatile unsigned short	lo_n_pdu_seg		OSI primitive data unit (PDU) IAPX-286 segment number received at Layer 4 from Layer 3. This segment number can be converted to a pointer by shifting it left 16 bits. Line Setup configured for emulate mode only.
extern volatile unsigned short	m_lo_n_pdu_seg		OSI primitive data unit (PDU) IAPX-286 segment number received at Layer 4 from Layer 3. This segment number can be converted to a pointer by shifting it left 16 bits. Line Setup configured for emulate or monitor mode.
extern volatile const unsigned char	lo_n_prmtv_code	61/97 63/99 65/101 67/103 69/105 6b/107 6d/109 6f/111 71/113 73/115 75/117 77/119 79/121 7a/122	n conn ind n conn conf n data ind n data ack ind n expd data ind n reset ind n reset conf n disconn ind n debug ind n unit data ind n qual data ind n qual data ack ind n mgt facility ind n error report ind OSI primitive code received at Layer 4 in a PDU from Layer 3. Line Setup configured for emulate mode only.

Table 66-5 (continued)

Type	Variable	Value (hex/decimal)	Meaning
extern volatile const unsigned char	m_lo_n_prmtv_code	64/100 65/101 68/102 69/103 74/116 75/117	td n data ind rd n data ind td n expd data ind rd n expd data ind td n unit data ind rd n unit data ind
			OSI primitive code received at Layer 4 in a PDU from Layer 3. Line Setup configured for emulate or monitor mode.
extern volatile const unsigned char	lo_n_prmtv_path	*0-8*	Path number received at Layer 4 in a PDU from Layer 3. Line Setup configured for emulate mode only.
extern volatile const unsigned char	m_lo_n_prmtv_path	*0-8*	Path number received at Layer 4 in a PDU from Layer 3. Line Setup configured for emulate or monitor mode.
extern volatile unsigned short	lo_n_il_buff		Interlayer-buffer number (an iAPX-286 segment number) received at Layer 4 in a PDU from Layer 3. This segment number can be converted to a pointer by shifting it left 16 bits. Line Setup configured for emulate mode only.
extern volatile unsigned short	m_lo_n_il_buff		Interlayer-buffer number (an iAPX-286 segment number) received at Layer 4 in a PDU from Layer 3. This segment number can be converted to a pointer by shifting it left 16 bits. Line Setup configured for emulate or monitor mode.
extern volatile unsigned short	lo_n_sdu		In OSI primitive received at Layer 4 from Layer 3, the offset to where the service data unit begins. Line Setup configured for emulate mode only.
extern volatile unsigned short	m_lo_n_sdu_offset		In OSI primitive received at Layer 4 from Layer 3, the offset to where the service data unit begins. Line Setup configured for emulate or monitor mode.
extern volatile unsigned short	m_lo_n_sdu_size		Size of the service data unit in an interlayer-message buffer. Received at Layer 4 from Layer 3. Same as *data_length* in a PDU. Line Setup configured for emulate or monitor mode.

Table 66-5 (continued)

Type	Variable	Value (hex/decimal)	Meaning
extern volatile unsigned short	up_t_pdu_seg		OSI primitive data unit (PDU) iAPX-286 segment number received at Layer 4 from Layer 5. This segment number can be converted to a pointer by shifting it left 16 bits. Line Setup configured for emulate mode only.
extern volatile const unsigned char	up_t_prmtv_code	80/128 82/130 84/132 88/136 8e/142 90/144 92/146 98/152	t conn req t conn resp t data req t expd data req t disconn req t debug req t unit data req t mgt facility req OSI primitive code received at Layer 4 in a PDU from Layer 5. Line Setup configured for emulate mode only.
extern volatile const unsigned char	up_t_prmtv_path	*0-8*	Path number received at Layer 4 in a PDU from Layer 5. Line Setup configured for emulate mode only.
extern volatile unsigned short	up_t_il_buff		Interlayer-buffer number (an iAPX-286 segment number) received at Layer 4 in a PDU from Layer 5. This segment number can be converted to a pointer by shifting it left 16 bits. Line Setup configured for emulate mode only.
extern volatile unsigned short	up_t_sdu		Offset to the start (header node) of the service data unit in an interlayer-message buffer. Received at Layer 4 from Layer 5. Same as *data_start_offset* in a PDU. Line Setup configured for emulate mode only.
extern unsigned long	l4_tick_count		32-bit *l1_tick_count* stored in header of most recent IL buffer passed up to Layer 4. Preserves at each layer the original time when the end of the data (BCC) was clocked into the buffer. Line Setup configured for emulate or monitor mode.

Table 66-6
Layer 5 OSI Variables

Type	Variable	Value (hex/decimal)	Meaning
extern event	lo_t_prmtv		True when an OSI primitive is received at Layer 5 from Layer 4. Line Setup configured for emulate mode only.
extern event	m_lo_t_prmtv		True when an OSI primitive is received at Layer 5 from Layer 4. Line Setup configured for emulate or monitor mode.
extern event	up_s_prmtv		True when an OSI primitive is received at Layer 5 from Layer 6. Line Setup configured for emulate mode only.
extern volatile unsigned short	lo_t_pdu_seg		OSI primitive data unit (PDU) IAPX-286 segment number received at Layer 5 from Layer 4. This segment number can be converted to a pointer by shifting it left 16 bits. Line Setup configured for emulate mode only.
extern volatile unsigned short	m_lo_t_pdu_seg		OSI primitive data unit (PDU) IAPX-286 segment number received at Layer 5 from Layer 4. This segment number can be converted to a pointer by shifting it left 16 bits. Line Setup configured for emulate or monitor mode.
extern volatile const unsigned char	lo_t_prmtv_code	81/129 83/131 85/133 89/137 8f/143 91/145 93/147 95/149 99/153	t conn ind t conn conf t data ind t expd data ind t disconn ind t debug ind t unit data ind t error report ind t mgt facility ind OSI primitive code received at Layer 5 in a PDU from Layer 4. Line Setup configured for emulate mode only.
extern volatile const unsigned char	m_lo_t_prmtv_code	84/132 85/133 88/136 89/137 94/148 95/149	td t data ind rd t data ind td t expd data ind rd t expd data ind td t unit data ind rd t unit data ind OSI primitive code received at Layer 5 in a PDU from Layer 4. Line Setup configured for emulate or monitor mode.

Table 66-6 (continued)

Type	Variable	Value (hex/decimal)	Meaning
extern volatile const unsigned char	lo_t_prmtv_path	*0-8*	Path number received at Layer 5 in a PDU from Layer 4. Line Setup configured for emulate mode only.
extern volatile const unsigned char	m_lo_t_prmtv_path	*0-8*	Path number received at Layer 5 in a PDU from Layer 4. Line Setup configured for emulate or monitor mode.
extern volatile unsigned short	lo_t_il_buff		Interlayer-buffer number (an iAPX-286 segment number) received at Layer 5 in a PDU from Layer 4. This segment number can be converted to a pointer by shifting it left 16 bits. Line Setup configured for emulate mode only.
extern volatile unsigned short	m_lo_t_il_buff		Interlayer-buffer number (an iAPX-286 segment number) received at Layer 5 in a PDU from Layer 4. This segment number can be converted to a pointer by shifting it left 16 bits. Line Setup configured for emulate or monitor mode.
extern volatile unsigned short	lo_t_sdu		In OSI primitive received at Layer 5 from Layer 4, the offset to where the service data unit begins. Line Setup configured for emulate mode only.
extern volatile unsigned short	m_lo_t_sdu_offset		In OSI primitive received at Layer 5 from Layer 4, the offset to where the service data unit begins. Line Setup configured for emulate or monitor mode.
extern volatile unsigned short	m_lo_t_sdu_size		Size of the service data unit in an interlayer-message buffer. Received at Layer 5 from Layer 4. Same as *data_length* in a PDU. Line Setup configured for emulate or monitor mode.
extern volatile unsigned short	up_s_pdu_seg		OSI primitive data unit (PDU) iAPX-286 segment number received at Layer 5 from Layer 6. This segment number can be converted to a pointer by shifting it left 16 bits. Line Setup configured for emulate mode only.

Table 66-6 (continued)

Type	Variable	Value (hex/decimal)	Meaning
extern volatile const unsigned char	up_s_prmtv_code	a0/160 a2/162 a4/164 a8/168 ac/172 ae/174 b0/176 b2/178 b8/184	s conn req s conn resp s data req s expd data req s release req s release resp s debug req s unit data req s mgt facility req OSI primitive code received at Layer 5 in a PDU from Layer 6. Line Setup configured for emulate mode only.
extern volatile const unsigned char	up_s_prmtv_path	*0-8*	Path number received at Layer 5 in a PDU from Layer 6. Line Setup configured for emulate mode only.
extern volatile unsigned short	up_s_il_buff		Interlayer-buffer number (an IAPX-286 segment number) received at Layer 5 in a PDU from Layer 6. This segment number can be converted to a pointer by shifting it left 16 bits. Line Setup configured for emulate mode only.
extern volatile unsigned short	up_s_sdu		Offset to the start (header node) of the service data unit in an interlayer-message buffer. Received at Layer 5 from Layer 6. Same as *data_start_offset* in a PDU. Line Setup configured for emulate mode only.
extern unsigned long	l5_tick_count		32-bit *l1_tick_count* stored in header of most recent IL buffer passed up to Layer 5. Preserves at each layer the original time when the end of the data (BCC) was clocked into the buffer. Line Setup configured for emulate or monitor mode.

Table 66-7
Layer 6 OSI Variables

Type	Variable	Value (hex/decimal)	Meaning
extern event	lo_s_prmtv		True when an OSI primitive is received at Layer 6 from Layer 5. Line Setup configured for emulate mode only.
extern event	m_lo_s_prmtv		True when an OSI primitive is received at Layer 6 from Layer 5. Line Setup configured for emulate or monitor mode.
extern event	up_p_prmtv		True when an OSI primitive is received at Layer 6 from Layer 7. Line Setup configured for emulate mode only.
extern volatile unsigned short	lo_s_pdu_seg		OSI primitive data unit (PDU) iAPX-286 segment number received at Layer 6 from Layer 5. This segment number can be converted to a pointer by shifting it left 16 bits. Line Setup configured for emulate mode only.
extern volatile unsigned short	m_lo_s_pdu_seg		OSI primitive data unit (PDU) iAPX-286 segment number received at Layer 6 from Layer 5. This segment number can be converted to a pointer by shifting it left 16 bits. Line Setup configured for emulate or monitor mode.
extern volatile const unsigned char	lo_s_prmtv_code	a1/161 a3/163 a5/165 a9/169 ad/173 af/175 b1/177 b3/179 b5/181 b9/185	s conn ind s conn conf s data ind s expd data ind s release ind s release conf s debug ind s unit data ind s error report ind s mgt facility ind OSI primitive code received at Layer 6 in a PDU from Layer 5. Line Setup configured for emulate mode only.
extern volatile const unsigned char	m_lo_s_prmtv_code	a4/164 a5/165 a8/168 a9/169 b4/180 b5/181	td s data ind rd s data ind td s expd data ind rd s expd data ind td s unit data ind rd s unit data ind OSI primitive code received at Layer 6 in a PDU from Layer 5. Line Setup configured for emulate or monitor mode.

Table 66-7 (continued)

Type	Variable	Value (hex/decimal)	Meaning
extern volatile const unsigned char	lo_s_prmtv_path	*0-8*	Path number received at Layer 6 in a PDU from Layer 5. Line Setup configured for emulate mode only.
extern volatile const unsigned char	m_lo_s_prmtv_path	*0-8*	Path number received at Layer 6 in a PDU from Layer 5. Line Setup configured for emulate or monitor mode.
extern volatile unsigned short	lo_s_il_buff		Interlayer-buffer number (an iAPX-286 segment number) received at Layer 6 in a PDU from Layer 5. This segment number can be converted to a pointer by shifting it left 16 bits. Line Setup configured for emulate mode only.
extern volatile unsigned short	m_lo_s_il_buff		Interlayer-buffer number (an iAPX-286 segment number) received at Layer 6 in a PDU from Layer 5. This segment number can be converted to a pointer by shifting it left 16 bits. Line Setup configured for emulate or monitor mode.
extern volatile unsigned short	lo_s_sdu		In OSI primitive received at Layer 6 from Layer 5, the offset to where the service data unit begins. Line Setup configured for emulate mode only.
extern volatile unsigned short	m_lo_s_sdu_offset		In OSI primitive received at Layer 6 from Layer 5, the offset to where the service data unit begins. Line Setup configured for emulate or monitor mode.
extern volatile unsigned short	m_lo_s_sdu_size		Size of the service data unit in an interlayer-message buffer. Received at Layer 6 from Layer 5. Same as *data_length* in a PDU. Line Setup configured for emulate or monitor mode.
extern volatile unsigned short	up_p_pdu_seg		OSI primitive data unit (PDU) iAPX-286 segment number received at Layer 6 from Layer 7. This segment number can be converted to a pointer by shifting it left 16 bits. Line Setup configured for emulate mode only.

Table 66-7 (continued)

Type	Variable	Value (hex/decimal)	Meaning
extern volatile const unsigned char	up_p_prmtv_code	c0/192 c2/194 c4/196 c8/200 cc/204 ce/206 d0/208 d2/210 d8/216	p conn req p conn resp p data req p expd data req p release req p release resp p debug req p unit data req p mgt facility req OSI primitive code received at Layer 6 from Layer 7 in a PDU. Line Setup configured for emulate mode only.
extern volatile const unsigned char	up_p_prmtv_path	*0-8*	Path number received at Layer 6 from Layer 7 in a PDU. Line Setup configured for emulate mode only.
extern volatile unsigned short	up_p_il_buff		Interlayer-buffer number (an IAPX-286 segment number) received at Layer 6 from Layer 7 in a PDU. This segment number can be converted to a pointer by shifting it left 16 bits. Line Setup configured for emulate mode only.
extern volatile unsigned short	up_p_sdu		Offset to the start (header node) of the service data unit in an interlayer-message buffer. Received at Layer 6 from Layer 7. Same as *data_start_offset* in a PDU. Line Setup configured for emulate mode only.
extern unsigned long	l6_tick_count		32-bit *l1_tick_count* stored in header of most recent IL buffer passed up to Layer 6. Preserves at each layer the original time when the end of the data (BCC) was clocked into the buffer. Line Setup configured for emulate or monitor mode.

Table 66-8
Layer 7 OSI Variables

Type	Variable	Value (hex/decimal)	Meaning
extern event	lo_p_prmtv		True when an OSI primitive is received at Layer 7 from Layer 6. Line Setup configured for emulate mode only.
extern event	m_lo_p_prmtv		True when an OSI primitive is received at Layer 7 from Layer 6. Line Setup configured for emulate or monitor mode.
extern volatile unsigned short	lo_p_pdu_seg		OSI primitive data unit (PDU) iAPX-286 segment number received at Layer 7 from Layer 6. This segment number can be converted to a pointer by shifting it left 16 bits. Line Setup configured for emulate mode only.
extern volatile unsigned short	m_lo_p_pdu_seg		OSI primitive data unit (PDU) iAPX-286 segment number received at Layer 7 from Layer 6. This segment number can be converted to a pointer by shifting it left 16 bits. Line Setup configured for emulate or monitor mode.
extern volatile const unsigned char	lo_p_prmtv_code	c1/193 c3/195 c5/197 c9/201 cd/205 cf/207 d1/209 d3/211 d5/213 d9/217	p conn ind p conn conf p data ind p expd data ind p release ind p release conf p debug ind p unit data ind p error report ind p mgt facility ind OSI primitive code received at Layer 7 in a PDU from Layer 6. Line Setup configured for emulate mode only.
extern volatile const unsigned char	m_lo_p_prmtv_code	c4/196 c5/197 c8/200 c9/201 d4/212 d5/213	td p data ind rd p data ind td p expd data ind rd p expd data ind td p unit data ind rd p unit data ind OSI primitive code received at Layer 7 in a PDU from Layer 6. Line Setup configured for emulate or monitor mode.
extern volatile const unsigned char	lo_p_prmtv_path	*0-8*	Path number received at Layer 7 in a PDU from Layer 6. Line Setup configured for emulate mode only.

Table 66-8 (continued)

Type	Variable	Value (hex/decimal)	Meaning
extern volatile const unsigned char	m_lo_p_prmtv_path	*0-8*	Path number received at Layer 7 in a PDU from Layer 6. Line Setup configured for emulate or monitor mode.
extern volatile unsigned short	lo_p_il_buff		Interlayer-buffer number (an IAPX-286 segment number) received at Layer 7 in a PDU from Layer 6. This segment number can be converted to a pointer by shifting it left 16 bits. Line Setup configured for emulate mode only.
extern volatile unsigned short	m_lo_p_il_buff		Interlayer-buffer number (an IAPX-286 segment number) received at Layer 7 in a PDU from Layer 6. This segment number can be converted to a pointer by shifting it left 16 bits. Line Setup configured for emulate or monitor mode.
extern volatile unsigned short	lo_p_sdu		In OSI primitive received at Layer 7 from Layer 6, the offset to where the service data unit begins. Line Setup configured for emulate mode only.
extern volatile unsigned short	m_lo_p_sdu_offset		In OSI primitive received at Layer 7 from Layer 6, the offset to where the service data unit begins. Line Setup configured for emulate or monitor mode.
extern volatile unsigned short	m_lo_p_sdu_size		Size of the service data unit in an interlayer-message buffer. Received at Layer 7 from Layer 6. Same as *data_length* in a PDU. Line Setup configured for emulate or monitor mode.
extern unsigned long	l7_tick_count		32-bit *l1_tick_count* stored in header of most recent IL buffer passed up to Layer 7. Preserves at each layer the original time when the end of the data (BCC) was clocked into the buffer. Line Setup configured for emulate or monitor mode.

66.3 Routines

OSI routines available at each layer make sending primitives to a layer above or below possible (see Figure 66-3). The routine name and its arguments provide the same information as the softkey selections on the Protocol Spreadsheet. (In the early phases of compiling the program, the C translator uses the routines to convert the spreadsheet softkey-token primitives into C.) All routines are protocol-independent.

(A) Layer-Independent OSI Routines

The following interlayer buffer service routines operate at any layer, regardless of protocol (or in the absence of a protocol package).

_get_il_msg_buff

Synopsis

```
extern void _get_il_msg_buff(buffer_number_ptr, maintain_bit_ptr);
unsigned short * buffer_number_ptr;
unsigned short * maintain_bit_ptr;
```

Description

The *_get_il_msg_buff* routine gets a free interlayer message buffer from the pool and returns the buffer number to the caller for use in subsequent calls to other interlayer buffer services. It also returns a maintain bit for use in the freeing operation.

Inputs

The first parameter is a pointer to the location where the buffer number is to be stored. The buffer number that is returned is actually an iAPX-286 segment number which can be converted to a pointer by shifting it 16 bits to the left. If there is no free buffer available, the routine will wait for one to become available.

The second parameter is a pointer to the location where the maintain bit will be stored. Since it must be used in the freeing operation, the maintain bit value should not be modified. The zero bit in this variable indicates your maintain bit.

Example

The variables in which the returned buffer number and maintain bit will be stored must be declared. When calling the routine, reference the addresses of these variables.

```
{
 unsigned short il_buffer_number;
 unsigned short relay_baton;
}
LAYER: 4
     STATE: get_a_buffer
        CONDITIONS: KEYBOARD " "
        ACTIONS:
        {
         _get_il_msg_buff(&il_buffer_number, &relay_baton);
        }
```

The routine will get a buffer number and store it in variable *il_buffer_number*. It will also return a maintain bit and store it in variable *relay_baton*.

_start_il_buff_list

Synopsis

```
extern void _start_il_buff_list(il_buffer_number, start_offset_ptr);
unsigned short il_buffer_number;
unsigned short * start_offset_ptr;
```

Description

The *_start_il_buff_list* routine starts a linked list of text inside an interlayer message buffer. The list is made up of a header node and text nodes. The header node contains offsets to the first and last text nodes. Each text node contains a pointer to the actual text, the length of the text, and the offset to the next text node. This routine actually creates the header node inside the interlayer message buffer and initializes the first and last text node offsets to zero, indicating an empty list. It will return the offset to the list header node for use in subsequent list service calls.

Inputs

The first parameter is the interlayer message buffer number that will contain the list.

The second parameter is a pointer to the location where the offset to the list header will be stored. The returned offset will be zero if there is insufficient room in the buffer for the header node and one text node. Otherwise, it is the offset from the beginning of the message buffer to the start of the header node.

To convert the offset into a pointer, shift the buffer number 16 bits to the left and add the offset:

```
(void *)(((long)il_buffer_number << 16) + data_start_offset);
```

Example

Get a buffer and start a linked list. The variable in which the returned offset will be stored must be declared. When calling the routine, reference the address of this variable.

```
{
 unsigned short il_buffer_number;
 unsigned short relay_baton;
 unsigned short data_start_offset;
}
```

```
STATE: start_a_list
   CONDITIONS: KEYBOARD " "
   ACTIONS:
   {
    _get_il_msg_buff(&il_buffer_number, &relay_baton);
    _start_il_buff_list(il_buffer_number, &data_start_offset);
/* See _insert_il_buff_list_cnt routine on how information is inserted in the buffer. */
   }
```

The routine will get the offset to the header node and store it in variable *data_start_offset*.

_dup_il_buff_list_start

Synopsis

```
extern unsigned short _dup_il_buff_list_start(il_buffer_number, start_offset,
   new_start_offset_ptr);
unsigned short il_buffer_number;
unsigned short start_offset;
unsigned short * new_start_offset_ptr;
```

Description

This routine duplicates the header node of a pointer list. In order for a layer to retain the ability to resend a buffer—that is, to reference again the same list header with the same first-node offset—it must keep its own linked list safe from data inserted at a layer below. The *_dup_il_buff_list_start* routine allows the *lower* layer to start its own list.

If the lower layer will *insert* data into the buffer, it need duplicate only the list header ("*list_start*"), not the entire list. If the layer will append data to the end of the buffer, it must duplicate the complete linked list via the *_dup_il_buff_list* routine.

Inputs

The first parameter is the interlayer message buffer number in which the header node will be duplicated.

The second parameter is the offset to the header node to be duplicated.

The third parameter is a pointer to the location where the offset to the new header node will be stored.

Returns

This routine returns zero if there is not enough room in the buffer for the duplicated header node and at least one list node.

Example

Duplicate the header node of a buffer passed down from Layer 3.

```
{
 extern volatile unsigned short up_dl_il_buff;
 extern volatile unsigned short up_dl_sdu;
 unsigned short l2_data_start_offset;
}
LAYER: 3
      STATE: message
         CONDITIONS: KEYBOARD " "
         ACTIONS: DL_DATA REQ "ﾚﾋﾏﾝ(FOX)"
LAYER: 2
      STATE: duplicate_header
         CONDITIONS: DL_DATA REQ
         ACTIONS:
         {
          _dup_il_buff_list_start(up_dl_il_buff, up_dl_sdu, &l2_data_start_offset);
/* See _insert_il_buff_list_cnt routine on how information is inserted in the buffer. */
         }
```

_dup_il_buff_list

Synopsis

```
extern unsigned short _dup_il_buff_list(il_buffer_number, start_offset, new_start_offset_ptr);
unsigned short il_buffer_number;
unsigned short start_offset;
unsigned short * new_start_offset_ptr;
```

Description

This routine duplicates an entire pointer list. In order for a layer to be able to retain the ability to resend a buffer—that is, to reference again the same list header with the same first- and last-node offsets—it must keep its own linked list safe from data inserted and appended at a layer below. The *_dup_il_buff_list* routine allows the *lower* layer to have its own list.

If the lower layer will append data to the buffer, it should duplicate the entire linked list. If the layer will only insert data into the buffer, it need only duplicate the header node via the *_dup_il_buff_list_start* routine.

Inputs

The first parameter is the interlayer message buffer number in which the list will be duplicated.

The second parameter is the offset to the header node of the list to be duplicated.

The third parameter is a pointer to the location where the offset to the header node for the new list will be stored.

Returns

This routine returns zero if the duplication is successful. If there is not enough room in the buffer to duplicate the list, one is returned.

Example

Duplicate the entire pointer list of a buffer passed down from Layer 3.

```
{
 extern volatile unsigned short up_dl_il_buff;
 extern volatile unsigned short up_dl_sdu;
 unsigned short i2_data_start_offset;
}
LAYER: 3
     STATE: message
        CONDITIONS: KEYBOARD " "
        ACTIONS: DL_DATA REQ "ﾘﾋﾏﾝ((FOX))"
LAYER: 2
     STATE: duplicate_list
        CONDITIONS: DL_DATA REQ
        ACTIONS:
        {
         _dup_il_buff_list(up_dl_il_buff, up_dl_sdu, &i2_data_start_offset);
/* See _append_il_buff_list_cnt routine on how information is appended to the buffer. */
        }
```

_open_space_in_il_buff

Synopsis

```
extern void _open_space_in_il_buff(il_buffer_number, length, space_offset_ptr);
unsigned short il_buffer_number;
unsigned short length;
unsigned short * space_offset_ptr;
```

Description

The *_open_space_in_il_buff* routine opens up the requested amount of space in the specified interlayer message buffer. It returns an offset from the beginning of the buffer to the start of the open space.

Inputs

The first parameter is the interlayer message buffer number in which space is to be made.

The second parameter is the amount of space (number of bytes) requested.

The third parameter is a pointer to the location where the returned offset will be stored. The returned offset will be zero if there is insufficient room in the buffer.

To convert the offset into a pointer, shift the buffer number 16 bits to the left and add the offset:

```
(void *)(((long)il_buffer_number << 16) + available_space_offset);
```

Example

Always open space in the buffer if you are going to copy data (usually header information) into the buffer. If you are not going to copy data into the buffer, but reference its location in memory outside the buffer (usually user data), you do not need to open space.

The variable in which the returned offset will be stored must be declared. When calling the routine, reference the address of this variable. The length may be entered as a numeric value, in which case a length variable need not be declared.

For example, a buffer at Layer 3 will have three X.25-header bytes inserted. The call for space to hold the header would look like this:

```
{
 unsigned short il_buffer_number;
 unsigned short relay_baton;
 unsigned short data_start_offset;
 unsigned short available_space_offset;
}
      STATE: get_space
         CONDITIONS: KEYBOARD " "
         ACTIONS:
         {
          _get_il_msg_buff(&il_buffer_number, &relay_baton);
          _start_il_buff_list(il_buffer_number, &data_start_offset);
          _open_space_in_il_buff(il_buffer_number, 3, &available_space_offset);
/* See _insert_il_buff_list_cnt routine on how information is inserted in the buffer. */
         }
```

The routine will get the offset to the next available space in the buffer and store it in variable *available_space_offset*.

Once space has been opened, the buffer-number and available-space variables can be converted into an open-space pointer. With this pointer, data can be copied into the space. The pointer can then be referenced in an *_insert_il_buff_list_cnt* routine, so that the opened space becomes threaded onto the linked list in the IL buffer. See the programming example under *_insert_il_buff_list_cnt*.

_free_il_msg_buff

Synopsis

```
extern void _free_il_msg_buff(il_buffer_number, relay_baton);
unsigned short il_buffer_number;
unsigned short relay_baton;
```

Description

The *_free_il_msg_buff* routine returns an interlayer message buffer to the pool of free buffers. Before actually returning the buffer to the pool, this routine verifies that all maintain bits have been reset, assuring that all users have freed this buffer.

Inputs

The first parameter is the interlayer-buffer number to be freed.

The second parameter is the maintain bit associated with the buffer user to be freed.

Example

See *_set_maint_buff_bit* routine.

_set_maint_buff_bit

Synopsis

```
extern void _set_maint_buff_bit(il_buffer_number, new_bit_ptr);
unsigned short il_buffer_number;
unsigned short * new_bit_ptr;
```

Description

The *_set_maint_buff_bit* routine sets a new maintain bit for a given interlayer message buffer. It returns that bit to the caller to be used in the freeing operation.

The maintain bit allocated in the *_get_il_msg_buff* routine should be considered valid only for the layer at which it was obtained. Once you pass a buffer, the maintain bit will hold the buffer at the next layer only until action on it has been processed. (In Spreadsheet terms, the buffer will be held until the ACTIONS block has been processed in response to the first CONDITIONS block identifying the buffer. In any other CONDITIONS block referring to the buffer, the buffer will not be found unless an additional maintain bit was set.) The maintain bit

eventually will be freed automatically whether or not any action is taken on it at the next layer. To hold a buffer at a particular layer, or to continue passing the buffer (in either direction), a new maintain bit must be set. The same maintain bit cannot be used continuously, since it will be freed after the *first* process on it (an ACTION to send, for example).

If you wish to keep a buffer available for your use while also sending it to another layer, set two maintain bits. One will be used to pass the buffer; the other will "maintain" the buffer for other processes. The latter will have to be freed via the *_free_il_msg_buff* routine.

Inputs

The first parameter is the interlayer-buffer number in which the new bit will be set.

The second parameter is a pointer to the location where the returned maintain bit will be stored. There are sixteen maintain bits reserved for each interlayer buffer. Each bit is identified by a two-byte variable with a single zero. The first maintain bit allocated is the least significant, so the value returned is hexadecimal FFFE (binary 11111111 11111110). The last maintain bit allocated is 7FFF (01111111 11111111). If all the maintain bits are already in use, FFFF will be returned.

The maintain bit value should not be modified. It must be used in the freeing operation to make sure the buffer is returned to the free buffer pool.

Example

The variable in which the returned maintain bit will be stored must be declared. When calling the routine, reference the address of this variable. For example, you receive a buffer at Layer 2 from Layer 3 (*up_dl_il_buff*) and insert information into it. Before passing the buffer to Layer 1, set two maintain bits. The one stored in variable *maintain_bit* will hold the buffer for the purpose of repeated resends of the frame, if necessary, and will have to be freed via the *_free_il_msg_buff* routine. When you pass the buffer down, use the bit in variable *l2_relay_baton*. When you resend the frame, set a new *resend_baton* bit and pass that down, still holding *maintain_bit* in reserve for subsequent resends.

```
{
unsigned short l2_relay_baton;
unsigned short resend_baton;
unsigned short maintain_bit;
extern volatile unsigned short up_dl_il_buff;
extern volatile unsigned short up_dl_sdu;
unsigned short l2_data_start_offset;
unsigned short available_space_offset;
static unsigned char l2_data[2] = {0x01, 0x00};
int i;
unsigned char * ptr_l2;
```

```
#define make_ptr(number,offset) ((void *)(((long)number << 16) + offset))
}
LAYER: 3
     STATE: send_fox_message
        CONDITIONS: KEYBOARD " "
        ACTIONS: DL_DATA REQ "ﾚﾗﾝ《FOX》"
LAYER: 2
     STATE: send_a_buffer
        CONDITIONS: DL_DATA REQ
        ACTIONS:
        {
/* See _insert_il_buff_list_cnt routine for an explanation of how information is inserted in the
buffer. */
         _dup_il_buff_list_start (up_dl_il_buff, up_dl_sdu, &l2_data_start_offset);
         _open_space_in_il_buff(up_dl_il_buff, 2, &available_space_offset);
         ptr_l2 = make_ptr(up_dl_il_buff, available_space_offset);
         for(i = 0; i < 2; i++)
            {
             *ptr_l2 = data_l2[i];
             ptr_l2++;
            }
         ptr_l2 -=2;
         _insert_il_buff_list_cnt(up_dl_il_buff, l2_data_start_offset, ptr_l2, 2);
         _set_maint_buff_bit(up_dl_il_buff, &maintain_bit);
         _set_maint_buff_bit(up_dl_il_buff, &l2_relay_baton);
         send_ph_prmtv_below(up_dl_il_buff, l2_relay_baton, l2_data_start_offset, 0, 0x24, 0);
        }
LAYER: 1
     STATE: resend_buffer
        CONDITIONS: RECEIVE STRING "F0ｺ《XXXX1001》"
        ACTIONS:
        {
         _set_maint_buff_bit(up_dl_il_buff, &resend_baton);
         l1_il_transmit(up_dl_il_buff, resend_baton, l2_data_start_offset, 1);
/* See Section 62, Monitor/Transmit Line Data, for an explanation of the l1_il_transmit
routine. */
        }
        CONDITIONS: RECEIVE STRING "F日《XXXX0001》"
        ACTIONS:
        {
         _free_il_msg_buff(up_dl_il_buff, maintain_bit);
/* See _free_il_msg_buff for an explanation of this routine. */
        }
```

_insert_il_buff_list_cnt

Synopsis

```
extern unsigned short _insert_il_buff_list_cnt(il_buffer_number, data_start_offset, text_ptr,
   text_length);
unsigned short il_buffer_number;
unsigned short data_start_offset;
unsigned char * text_ptr;
unsigned short text_length;
```

Description

The *_insert_il_buff_list_cnt* routine inserts a text node at the beginning of a linked list of text inside of an interlayer message buffer. It will set the text pointer and byte-count in the text node to the values specified.

Inputs

The first parameter is the interlayer-buffer number in which the linked list will be inserted.

The second parameter is the offset to the header node for the linked list, from the beginning of the buffer.

The third parameter is a pointer to a text.

The fourth parameter is the length of the text.

Returns

If the insert is successful, a value of 0 is returned; if it is not successful, a value of 1 is returned. If you want to check the returned value, do so at the time the routine is called, as in the following example at Layers 2 and 3.

Example

If text is to be copied into the buffer, a pointer to the text must be declared. If not, when calling the *_insert_il_buff_list_cnt* routine, reference the address of the text. The length of the text may be entered as an integer, in which case a *length* variable need not be declared.

Always open space in the buffer if you are going to copy data (usually header information) into the buffer. If you are not going to copy data into the buffer, but reference its location in memory outside the buffer (usually user data), you do not need to open space.

In the following spreadsheet example, an interlayer-buffer number is obtained at Layer 5, a header node is created in the buffer, and the address of a fox message text (located in memory outside of the buffer) is inserted into a text node in the buffer.

```
{
unsigned short il_buffer_number;
unsigned short relay_baton;
unsigned short l4_relay_baton
unsigned short l3_relay_baton;
unsigned short l2_relay_baton;
unsigned short data_start_offset;
unsigned short l2_data_start_offset;
```

```
unsigned short available_space_offset;
static unsigned char data[] = "((FOX))";
static unsigned char l3_data[3] = {0x10, 0x04, 0x00};
static unsigned char l2_data[2] = {0x01, 0x00};
int i;
int length;
extern volatile unsigned short up_t_il_buff;
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;
extern volatile unsigned short up_t_sdu;
unsigned char * ptr_l3, * ptr_l2;

/* Whenever make_ptr is encountered, the first parameter will be shifted 16 bits to the left.
The second parameter will be added, and the result cast into a pointer. */

#define make_ptr(number,offset) ((void *)(((long)number << 16) + offset))
}
LAYER: 5
    STATE: begin_message
        CONDITIONS: KEYBOARD " "
        ACTIONS:
        {
         _get_il_msg_buff(&il_buffer_number, &relay_baton);
         _start_il_buff_list(il_buffer_number, &data_start_offset);

/* Do not include the terminating null character in the length determination of a string. */

         length = sizeof(data) - 1;

/* The address of data outside of the buffer is given for insertion. The data itself is not copied
into the buffer. The buffer is then passed down to Layer 4 (see send_t_prmtv_below for an
explanation of this routine). */

         _insert_il_buff_list_cnt(il_buffer_number, data_start_offset, &data[0], length);
         send_t_prmtv_below(il_buffer_number, relay_baton, data_start_offset, 0, 0x84, 0);
        }
```

At Layer 4 a new maintain bit is set to use in passing the buffer to Layer 3. Since no data is inserted, the same *data_start_offset* is used (in the form of the variable *up_t_sdu*). The buffer is then passed down to Layer 3 (see *send_n_prmtv_below* for an explanation of this routine).

```
LAYER: 4
    STATE: pass
        CONDITIONS: T_DATA_REQ
        ACTIONS:
        {
         _set_maint_buff_bit(up_t_il_buff, &l4_relay_baton);
         send_n_prmtv_below(up_t_il_buff, l4_relay_baton, up_t_sdu, 0, 0x64, 0);
        }
```

At Layer 3, space is opened for an X.25 packet header. A pointer to the opened space is created and the data is inserted into the linked list passed down from Layer 4.

```
LAYER: 3
     STATE: Insert_and_send
        CONDITIONS: N_DATA_REQ
        ACTIONS:
        {
         _open_space_in_il_buff(up_n_il_buff, 3, &available_space_offset);
         ptr_l3 = make_ptr(up_n_il_buff, available_space_offset);
         for(i = 0; i < 3; i++)
            {
             *ptr_l3 = l3_data[i];
             ptr_l3++;
            }
/* The location of the data in the buffer is referenced in the insert routine, so the pointer must
be moved back to the beginning of the opened space. The offset to the Layer 3 header node is
given in the insert routine. If the insertion is not successful, an alarm will sound and a message
will be displayed on the prompt line of the screen. */
         ptr_l3 -=3;
         if(_insert_il_buff_list_cnt(up_n_il_buff, up_n_sdu, ptr_l3, 3) != 0)
            {
             sound_alarm();
             display_prompt("Insert failed at Layer 3.");
            }
/* A new maintain bit is set for passing the buffer. The buffer is then passed down to Layer 2
(see send_dl_prmtv_below for an explanation of this routine). */
         _set_maint_buff_bit(up_n_il_buff, &l3_relay_baton);
         send_dl_prmtv_below(up_n_il_buff, l3_relay_baton, up_n_sdu, 0, 0x44, 0);
        }
```

At Layer 2, a new linked list is started. The Layer 2 header could be inserted into the linked list passed down from Layer 3; but if Layer 3 wants to retain the ability to resend a buffer—that is, to reference again the same list header with the same first-node offset—it must keep its own linked list safe from data inserted at Layer 2.

```
LAYER: 2
     STATE: Insert_more
        CONDITIONS: DL_DATA_REQ
        ACTIONS:
        {
/* The _dup_il_buff_list_start routine allows Layer 2 to start its own list. Part of this routine
copies the Layer 3 header into the Layer 2 header node. */
         _dup_il_buff_list_start(up_dl_il_buff, up_dl_sdu, &l2_data_start_offset);
/* Space is opened in the buffer. A pointer to this location is created and the data is copied
into the buffer. */
         _open_space_in_il_buff(up_dl_il_buff, 2, &available_space_offset);
         ptr_l2 = make_ptr(up_dl_il_buff, available_space_offset);
         for(i = 0; i < 2; i++)
            {
             *ptr_l2 = l2_data[i];
             ptr_l2++;
            }
/* The location of the data in the buffer is referenced in the insert routine, so the pointer must
be moved back to the beginning of the opened space. The offset to the Layer 2 header node is
given in the insert routine. If the insertion is not successful, an alarm will sound and a message
will be displayed on the prompt line of the screen. */
```

```
        ptr_l2 -=2;
        if(_insert_il_buff_list_cnt(up_dl_il_buff, l2_data_start_offset, ptr_l2, 2) != 0)
           {
            sound_alarm();
            pos_cursor(0,30);
            displays("Insert failed at Layer 2.");
           }
/* A new maintain bit is set for passing the buffer. The buffer is then passed down to Layer 1
(see send_ph_prmtv_below for an explanation of this routine). */
        _set_maint_buff_bit(up_dl_il_buff, &l2_relay_baton);
        send_ph_prmtv_below(up_dl_il_buff, l2_relay_baton, l2_data_start_offset, 0, 0x24, 0);
      }
```

The following text will be sent out onto the line and displayed as line data:

THE QUICK BROWN FOX JUMPS OVER THE LAZY DOG 0123456789

_append_il_buff_list_cnt

Synopsis

```
extern unsigned short _append_il_buff_list_cnt(il_buffer_number, data_start_offset, text_ptr,
   text_length);
unsigned short il_buffer_number;
unsigned short data_start_offset;
unsigned char * text_ptr;
unsigned short text_length;
```

Description

The *_append_il_buff_list_cnt* routine appends a text node at the end of a linked list of text inside of an interlayer message buffer. It will set the text pointer and count in the text node to the information provided.

Inputs

See *_insert_il_buff_list_cnt* routine.

Returns

See *_insert_il_buff_list_cnt* routine.

Example

Two modifications to the program shown for the *_insert_il_buff_list_cnt* routine are all that is required to make the program work for appending data. The changes primarily involve Layer 2 in the example, so we will replicate only that portion of the program below. Substitute *_append_il_buff_list_cnt* for every occurrence *_insert_il_buff_list_cnt*. When data is to be appended in a buffer, you should duplicate the entire linked list received from the layer above, not just the header node. So also substitute *_dup_il_buff_list* for *_dup_il_buff_list_start*.

```
LAYER: 2
    STATE: Insert_more
       CONDITIONS: DL_DATA_REQ
       ACTIONS:
       {
        _dup_il_buff_list(up_dl_il_buff, up_dl_sdu, &l2_data_start_offset);
        _open_space_in_il_buff(up_dl_il_buff, 2, &available_space_offset);
        ptr_l2 = make_ptr(up_dl_il_buff, available_space_offset);
        for(i = 0; i < 2; i++)
           {
            *ptr_l2 = l2_data[i];
            ptr_l2++;
           }
        ptr_l2 -=2;
        if(_append_il_buff_llst_cnt(up_dl_il_buff, l2_data_start_offset, ptr_l2, 2) != 0)
           {
            sound_alarm();
            pos_cursor(0,30);
            displays("Insert failed at Layer 2.");
           }
        _set_maint_buff_bit(up_dl_il_buff, &l2_relay_baton);
        send_ph_prmtv_below(up_dl_il_buff, l2_relay_baton, l2_data_start_offset, 0, 0x24, 0);
       }
```

The following text will be sent out onto the line and displayed as line data:

THE QUICK BROWN FOX JUMPS OVER THE LAZY DOG 0123456789

(B) Layer 1 OSI Routines

OSI data primitives are handled automatically between Layers 1 and 2. In the "up" direction, line data is placed in an IL buffer and the associated data primitive is given automatically to Layer 2. In the "down" direction, data primitives are received at Layer 1 and put out automatically onto the line.

In the absence of line data, if you want to originate a buffer at Layer 1 and send it upward, use the following routine. In primitives being sent down the layers, Layer 1 will automatically send the primitive out onto the line.

send_ph_to_above

Synopsis

```
extern void send_ph_to_above(il_buffer_number, relay_baton, data_start_offset, size, code,
   path);
unsigned short il_buffer_number;
unsigned short relay_baton;
unsigned short data_start_offset;
unsigned short size;
unsigned char code;
unsigned char path;
```

Description

The *send_ph_to_above* emulate routine passes a specified interlayer message buffer from Layer 1 to Layer 2 in an OSI primitive. Received line data is placed in an IL buffer and passed automatically to Layer 2. If you wish to get a buffer "manually" at Layer 1 and then pass it up, use this routine.

Inputs

The first parameter is the interlayer buffer number returned by the *_get_il_msg_buff* routine.

The second parameter is the returned maintain bit from the *_get_il_msg_buff* routine. As soon as Layer 2 processing on the buffer is completed, the bit is automatically freed.

The third parameter is the returned offset (from the call to *_start_il_buff_list*) to the Layer 1 service data unit in a buffer.

The fourth parameter is the length of the data in the buffer.

The fifth parameter is the code specifying the type of primitive in which the buffer will be sent. Refer to variable *lo_ph_prmtv_code* in Table 66-3 for the appropriate primitive code.

The sixth parameter is the path number along which the buffer will be sent.

Example

Get a buffer at Layer 1. Assuming X.25 protocol, insert data into the buffer and pass it up to Layer 2.

```
{
 unsigned short il_buffer_number;
 unsigned short relay_baton;
 unsigned short data_start_offset;
 unsigned short available_space_offset;
 int length;
 int i;
 static unsigned char data[] = {0x01, 0x00, 0x10, 0x04, 0x00, 0x02, 0x01, 0x01};
 unsigned char * ptr;
}
LAYER: 1
     STATE: get_buffer
        CONDITIONS: KEYBOARD " "
        ACTIONS:
        {
         _get_il_msg_buff(&il_buffer_number, &relay_baton);
         _start_il_buff_list(il_buffer_number, &data_start_offset);
         length = sizeof(data);
         _open_space_in_il_buff(il_buffer_number, length, &available_space_offset);
         ptr = (void *)(((long)il_buffer_number << 16) + available_space_offset);
```

```
for(i = 0; i < length; i++)
    {
     *ptr = data[i];
     ptr++;
    }
ptr-=length;
_insert_il_buff_list_cnt(il_buffer_number, data_start_offset, ptr, length);
send_ph_to_above(il_buffer_number, relay_baton, data_start_offset, length, 0x25, 0);
}
```

(C) Layer 2 OSI Routines

The following routines pass OSI primitives from Layer 2 to either Layer 3 or Layer 1.

send_dl_prmtv_above

Synopsis

```
extern void send_dl_prmtv_above(il_buffer_number, l2_relay_baton, l2_data_start_offset, size,
   l2_code, path);
unsigned short il_buffer_number;
unsigned short l2_relay_baton;
unsigned short l2_data_start_offset;
unsigned short size;
unsigned char l2_code;
unsigned char path;
```

Description

The *send_dl_prmtv_above* emulate routine passes a specified interlayer message buffer from Layer 2 to Layer 3 in an OSI primitive.

Inputs

The first parameter is the interlayer buffer number to be sent. For a buffer which has been received at Layer 2 from Layer 1, the variable *lo_ph_il_buff* may be used to identify the buffer number.

The second parameter is the returned maintain bit from a call to *_set_maint_buff_bit*. It is used only to pass a received buffer from Layer 2 to Layer 3. As soon as Layer 3 processing on the buffer is completed, the bit is automatically freed.

The third parameter is the offset to the Layer 2 service data unit in a received buffer. The variable *lo_ph_sdu* contains the offset to the service data unit when the buffer reached Layer 2. The offset must be incremented by the length of the Layer 2 header.

NOTE: In general, do not modify *extern* variables, such as *lo_ph_sdu*, which may be updated by other processes. Name another variable, assign it the same value, and then increment that variable. Or, after *lo_ph_sdu* has been named in the argument of the *send* routine, add the length of the Layer 2 header, as in the example below.

The fourth parameter is the length of the data in the buffer. Use the length indicated in the *pdu* structure—*pdu.data_length*. Then subtract the length of the Layer 2 header.

The fifth parameter is the code specifying the type of primitive in which the buffer will be sent. Refer to variable *lo_dl_prmtv_code* in Table 66-4 for the appropriate primitive code.

The sixth parameter is the path number along which the buffer will be sent. For a buffer which has been received at Layer 2 from Layer 1, the variable *lo_ph_prmtv_path* may be used to specify the path number.

Example

A buffer is received at Layer 2 from Layer 1. Assuming X.25 protocol, the data specific to Layer 2 (the frame header) begins at the SDU offset (*lo_ph_sdu*) and consists of two bytes. Before the buffer is passed up to Layer 3, the offset to the SDU and the size of the SDU will be adjusted by two bytes and a new maintain bit will be set.

```
{
 struct pdu
  {
   unsigned char primitive_code;
   unsigned char path;
   unsigned long parameter;
   unsigned short relay_baton;
   unsigned short il_buffer_number;
   unsigned char buffer_contents;
   unsigned short data_start_offset;
   unsigned short data_length;
  };
 struct pdu * pdu_ptr;
 extern volatile unsigned short lo_ph_pdu_seg;
 extern volatile const unsigned char lo_ph_prmtv_path;
 extern volatile unsigned short lo_ph_il_buff;
 extern volatile unsigned short lo_ph_sdu;
 unsigned short l2_relay_baton;
}
```

```
LAYER: 2
     STATE: send_buffer_up
        CONDITIONS: PH_DATA IND
        ACTIONS:
        {
         pdu_ptr = (void *)((long)lo_ph_pdu_seg << 16);
         _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, lo_ph_prmtv_path);
        }
```

send_m_dl_prmtv_above

Synopsis

```
extern void send_m_dl_prmtv_above(il_buffer_number, l2_relay_baton, l2_data_start_offset,
   size, l2_code, path);
unsigned short il_buffer_number;
unsigned short l2_relay_baton;
unsigned short l2_data_start_offset;
unsigned short size;
unsigned char l2_code;
unsigned char path;
```

Description

The *send_m_dl_prmtv_above* monitor routine passes a specified interlayer message buffer from Layer 2 to Layer 3 in an OSI monitor primitive.

Inputs

See *send_dl_prmtv_above*. Use the monitor variables *m_lo_ph_il_buff*, *m_lo_ph_sdu_offset*, and *m_lo_ph_sdu_size* as input. Refer to variable *m_lo_dl_prmtv_code* in Table 66-4 for the appropriate primitive code.

Example

Make the appropriate variable declarations. For a condition monitoring RD data primitives, the Layer 2 programming block should look like this:

```
LAYER: 2
     STATE: send_buffer_up
        CONDITIONS: PH_RD_DATA IND
        ACTIONS:
        {
         _set_maint_buff_bit(m_lo_ph_il_buff, &l2_relay_baton);
         send_m_dl_prmtv_above(m_lo_ph_il_buff, l2_relay_baton,m_ lo_ph_sdu_offset + 2,
              m_lo_ph_sdu_size - 2, 0x45, m_lo_ph_prmtv_path);
        }
```

send_ph_prmtv_below

Synopsis

```
extern void send_ph_prmtv_below(il_buffer_number, l2_relay_baton, l2_data_start_offset, size,
   l2_code, path);
unsigned short il_buffer_number;
unsigned short l2_relay_baton;
unsigned short l2_data_start_offset;
unsigned short size;
unsigned char l2_code;
unsigned char path;
```

Description

The *send_ph_prmtv_below* emulate routine passes a specified interlayer message buffer from Layer 2 to Layer 1 in an OSI primitive.

Inputs

The first parameter is the interlayer buffer number to be sent. For a buffer which has been received at Layer 2 from Layer 3, the variable *up_dl_il_buff* may be used to identify the buffer number. If the buffer originated at Layer 2, use the buffer-number variable named in the *_get_il_msg_buff* routine. (See *_insert_il_buff_list_cnt* routine example at Layer 5.)

The second parameter is the returned maintain bit from a call to *_set_maint_buff_bit*. It is used only to pass a received buffer from Layer 2 to Layer 1. As soon as Layer 1 processing on the buffer is completed, the bit is automatically freed. If the buffer originated at Layer 2, use the maintain bit variable named in the *_get_il_msg_buff* routine. (See *_insert_il_buff_list_cnt* routine example at Layer 5.)

The third parameter is the offset to the Layer 2 list header node in the buffer. For a buffer which has been received at Layer 2 from Layer 3, the variable *up_dl_sdu* may be used to indicate the offset.

The fourth parameter is the size of the data in the buffer. It will always be set to zero since the data length is unknown in a primitive being passed down the layers.

The fifth parameter is the code specifying the type of primitive in which the buffer will be sent. Refer to variable *ph_prmtv_type* in Table 66-2 for the appropriate primitive code.

The sixth parameter is the path number along which the buffer will be sent. For a buffer which has been received at Layer 2 from Layer 3, the variable *up_dl_prmtv_path* may be used to specify the path number.

Example

A buffer is received at Layer 2 from Layer 3. No text will be inserted at Layer 2. (For information on inserting text, see *_insert_il_buff_list_cnt* routine.) The buffer will be passed to Layer 1, requiring a new maintain bit to be set. If values are entered for the code and path, variables for code and path need not be declared.

```
{
 extern volatile unsigned short up_dl_il_buff;
 extern volatile unsigned short up_dl_sdu;
 unsigned short l2_relay_baton;
}
LAYER: 2
     STATE: pass_buffer_down
        CONDITIONS: DL_DATA REQ
        ACTIONS:
        {
         _set_maint_buff_bit(up_dl_il_buff, &l2_relay_baton);
         send_ph_prmtv_below(up_dl_il_buff, l2_relay_baton, up_dl_sdu, 0, 0x24, 0);
        }
```

(D) Layer 3 OSI Routines

The following routines pass OSI primitives from Layer 3 to either Layer 4 or Layer 2.

send_n_prmtv_above

Synopsis

```
extern void send_n_prmtv_above(il_buffer_number, l3_relay_baton, l3_data_start_offset, size,
   l3_code, path);
unsigned short il_buffer_number;
unsigned short l3_relay_baton;
unsigned short l3_data_start_offset;
unsigned short size;
unsigned char l3_code;
unsigned char path;
```

Description

The *send_n_prmtv_above* emulate routine passes a specified interlayer message buffer from Layer 3 to Layer 4 in an OSI primitive.

Inputs

The first parameter is the interlayer buffer number to be sent. For a buffer which has been received at Layer 3 from Layer 2, the variable *lo_dl_il_buff* may be used to identify the buffer number.

The second parameter is the returned maintain bit from a call to *_set_maint_buff_bit*. It is used only to pass a received buffer from Layer 3 to Layer 4. As soon as Layer 4 processing on the buffer is completed, the bit is automatically freed.

The third parameter is the offset to the Layer 3 service data unit in a received buffer. The variable *lo_dl_sdu* contains the offset to the service data unit when the buffer reached Layer 3. The offset must be incremented by the length of the Layer 3 header.

> NOTE: In general, do not modify *extern* variables, such as *lo_dl_sdu*, which may be updated by other processes. Name another variable, assign it the same value, and then increment that variable. Or, after *lo_dl_sdu* has been named in the argument of the *send* routine, add the length of the Layer 3 header, as in the example below.

The fourth parameter is the length of the data in the buffer. Use the length indicated in the *pdu* structure—*pdu.data_length*. Then subtract the length of the Layer 3 header.

The fifth parameter is the code specifying the type of primitive in which the buffer will be sent. Refer to variable *lo_n_prmtv_code* in Table 66-5 for the appropriate primitive code.

The sixth parameter is the path number along which the buffer will be sent. For a buffer which has been received at Layer 3 from Layer 2, the variable *lo_dl_prmtv_path* may be used to specify the path number.

Example

A buffer is received at Layer 3 from Layer 2. Assuming X.25 protocol, the header consists of three bytes. The offset to and size of the service data unit will be adjusted by three bytes, a new maintain bit will be set, and the buffer will be passed up to Layer 4.

```
{
struct pdu
 {
  unsigned char primitive_code;
  unsigned char path;
  unsigned long parameter;
  unsigned short relay_baton;
  unsigned short il_buffer_number;
  unsigned char buffer_contents;
  unsigned short data_start_offset;
  unsigned short data_length;
 };
```

```
 struct pdu * pdu_ptr;
 extern volatile unsigned short lo_dl_pdu_seg;
 extern volatile const unsigned char lo_dl_prmtv_path;
 extern volatile unsigned short lo_dl_il_buff;
 extern volatile unsigned short lo_dl_sdu;
 unsigned short l3_relay_baton;
}
LAYER: 3
     STATE: send_buffer_up
        CONDITIONS: DL_DATA IND
        ACTIONS:
        {
         pdu_ptr = (void *)((long)lo_dl_pdu_seg << 16);
         _set_maint_buff_bit(lo_dl_il_buff, &l3_relay_baton);
         send_n_prmtv_above(lo_dl_il_buff, l3_relay_baton, lo_dl_sdu + 3,
              pdu_ptr->data_length - 3, 0x65, lo_dl_prmtv_path);
        }
```

send_m_n_prmtv_above

Synopsis

```
extern void send_m_n_prmtv_above(il_buffer_number, l3_relay_baton, l3_data_start_offset,
   size, l3_code, path);
unsigned short il_buffer_number;
unsigned short l3_relay_baton;
unsigned short l3_data_start_offset;
unsigned short size;
unsigned char l3_code;
unsigned char path;
```

Description

The *send_m_n_prmtv_above* monitor routine passes a specified interlayer message buffer from Layer 3 to Layer 4 in an OSI monitor primitive.

Inputs

See *send_n_prmtv_above*. Use the monitor variables *m_lo_dl_il_buff*, *m_lo_dl_sdu_offset*, and *m_lo_dl_sdu_size* as input. Refer to variable *m_lo_n_prmtv_code* in Table 66-5 for the appropriate primitive code.

Example

Make the appropriate variable declarations. For a condition monitoring RD data primitives, the Layer 3 programming block should look like this:

```
LAYER: 3
     STATE: send_buffer_up
        CONDITIONS: DL_RD_DATA IND
        ACTIONS:
        {
         _set_maint_buff_bit(m_lo_dl_il_buff, &l3_relay_baton);
         send_m_n_prmtv_above(m_lo_dl_il_buff, l3_relay_baton, m_lo_dl_sdu_offset + 3,
              m_lo_dl_sdu_size - 3, 0x65, m_lo_dl_prmtv_path);
        }
```

send_dl_prmtv_below

Synopsis

```
extern void send_dl_prmtv_below(il_buffer_number, l3_relay_baton, l3_data_start_offset, size,
   l3_code, path);
unsigned short il_buffer_number;
unsigned short l3_relay_baton;
unsigned short l3_data_start_offset;
unsigned short size;
unsigned char l3_code;
unsigned char path;
```

Description

The *send_dl_prmtv_below* emulate routine passes a specified interlayer message buffer from Layer 3 to Layer 2 in an OSI primitive.

Inputs

The first parameter is the interlayer buffer number to be sent. For a buffer which has been received at Layer 3 from Layer 4, the variable *up_n_il_buff* may be used to identify the buffer number. If the buffer originated at Layer 3, use the buffer-number variable named in the *_get_il_msg_buff* routine. (See *_insert_il_buff_list_cnt* routine example at Layer 5.)

The second parameter is the returned maintain bit from a call to *_set_maint_buff_bit*. It is used only to pass a received buffer from Layer 3 to Layer 2. As soon as Layer 2 processing on the buffer is completed, the bit is automatically freed. If the buffer originated at Layer 3, use the maintain bit variable named in the *_get_il_msg_buff* routine. (See *_insert_il_buff_list_cnt* routine example at Layer 5.)

The third parameter is the offset to the Layer 3 list header node in the buffer. For a buffer which has been received at Layer 3 from Layer 4, the variable *up_n_sdu* may be used to indicate the offset.

The fourth parameter is the size of the data in the buffer. It will always be set to zero since the data length is unknown in a primitive being passed down the layers.

The fifth parameter is the code specifying the type of primitive in which the buffer will be sent. Refer to variable *up_dl_prmtv_code* in Table 66-3 for the appropriate primitive code.

The sixth parameter is the path number along which the buffer will be sent. For a buffer which has been received at Layer 3 from Layer 4, the variable *up_n_prmtv_path* may be used to specify the path number.

Example

A buffer is received at Layer 3 from Layer 4. No text will be inserted at Layer 3. (For information on inserting text, see *_insert_il_buff_list_cnt* routine.) The buffer will be passed to Layer 2, requiring a new maintain bit to be set. If values are entered for the code and path, these variables need not be declared.

```
{
 extern volatile unsigned short up_n_il_buff;
 extern volatile unsigned short up_n_sdu;
 unsigned short l3_relay_baton;
}
LAYER: 3
      STATE: pass_buffer_down
         CONDITIONS: N_DATA REQ
         ACTIONS:
         {
          _set_maint_buff_bit(up_n_il_buff, &l3_relay_baton);
          send_dl_prmtv_below(up_n_il_buff, l3_relay_baton, up_n_sdu, 0, 0x44, 0);
         }
```

(E) Layer 4 OSI Routines

The following routines pass OSI primitives from Layer 4 to either Layer 5 or Layer 3.

send_t_prmtv_above

Synopsis

```
extern void send_t_prmtv_above(il_buffer_number, l4_relay_baton, l4_data_start_offset, size,
   l4_code, path);
unsigned short il_buffer_number;
unsigned short l4_relay_baton;
unsigned short l4_data_start_offset;
unsigned short size;
unsigned char l4_code;
unsigned char path;
```

Description

The *send_t_prmtv_above* emulate routine passes a specified interlayer message buffer from Layer 4 to Layer 5 in an OSI primitive.

Inputs

The first parameter is the interlayer buffer number to be sent. For a buffer which has been received at Layer 4 from Layer 3, the variable *lo_n_il_buff* may be used to identify the buffer number.

The second parameter is the returned maintain bit from a call to *_set_maint_buff_bit*. It is used only to pass a received buffer from Layer 4 to Layer 5. As soon as Layer 5 processing on the buffer is completed, the bit is automatically freed.

The third parameter is the offset to the Layer 4 service data unit in a received buffer. The variable *lo_n_sdu* contains the offset to the service data unit when the buffer reached Layer 4. The offset must be incremented by the length of the Layer 4 header, if any.

> NOTE: In general, do not modify *extern* variables, such as *lo_n_sdu*, which may be updated by other processes. Name another variable, assign it the same value, and then increment that variable. Or, after *lo_n_sdu* has been named in the argument of the *send* routine, add the length of the Layer 4 header, if any.

The fourth parameter is the length of the data in the buffer. Use the length indicated in the *pdu* structure—*pdu.data_length*. Then subtract the length of the Layer 4 header, if any.

The fifth parameter is the code specifying the type of primitive in which the buffer will be sent. Refer to variable *lo_t_prmtv_code* in Table 66-6 for the appropriate primitive code.

The sixth parameter is the path number along which the buffer will be sent. For a buffer which has been received at Layer 4 from Layer 3, the variable *lo_n_prmtv_path* may be used to specify the path number.

Example

A buffer is received at Layer 4 from Layer 3. The offset to and size of the service data unit will be adjusted if needed, a new maintain bit will be set, and the buffer will be passed up to Layer 5.

```
{
struct pdu
 {
  unsigned char primitive_code;
  unsigned char path;
  unsigned long parameter;
  unsigned short relay_baton;
  unsigned short il_buffer_number;
  unsigned char buffer_contents;
  unsigned short data_start_offset;
  unsigned short data_length;
 };
struct pdu * pdu_ptr;
extern volatile unsigned short lo_n_pdu_seg;
extern volatile const unsigned char lo_n_prmtv_path;
```

```
 extern volatile unsigned short lo_n_il_buff;
 extern volatile unsigned short lo_n_sdu;
 unsigned short l4_relay_baton;
}
LAYER: 4
     STATE: send_buffer_up
        CONDITIONS: N_DATA IND
        ACTIONS:
        {
         pdu_ptr = (void *)((long)lo_n_pdu_seg << 16);
         _set_maint_buff_bit(lo_n_il_buff, &l4_relay_baton);
         send_t_prmtv_above(lo_n_il_buff, l4_relay_baton, lo_n_sdu,pdu_ptr->data_length,
              0x85, lo_n_prmtv_path);
        }
```

send_m_t_prmtv_above

Synopsis

```
extern void send_m_t_prmtv_above(il_buffer_number, l4_relay_baton, l4_data_start_offset,
   size, l4_code, path);
unsigned short il_buffer_number;
unsigned short l4_relay_baton;
unsigned short l4_data_start_offset;
unsigned short size;
unsigned char l4_code;
unsigned char path;
```

Description

The *send_m_t_prmtv_above* monitor routine passes a specified interlayer message buffer from Layer 4 to Layer 5 in an OSI monitor primitive.

Inputs

See *send_t_prmtv_above*. Use the monitor variables *m_lo_n_il_buff*, *m_lo_n_sdu_offset*, and *m_lo_n_sdu_size* as input. Refer to variable *m_lo_t_prmtv_code* in Table 66-6 for the appropriate primitive code.

Example

Make the appropriate variable declarations. For a condition monitoring RD data primitives, the Layer 4 programming block should look like this:

```
LAYER: 4
     STATE: send_buffer_up
        CONDITIONS: N_RD_DATA IND
        ACTIONS:
        {
         _set_maint_buff_bit(m_lo_n_il_buff, &l4_relay_baton);
         send_m_t_prmtv_above(m_lo_n_il_buff, l4_relay_baton,m_ lo_n_sdu_offset ,
              m_lo_n_sdu_size, 0x85, m_lo_n_prmtv_path);
        }
```

send_n_prmtv_below

Synopsis

```
extern void send_n_prmtv_below(l1_buffer_number, l4_relay_baton, l4_data_start_offset, size,
   l4_code, path);
unsigned short l1_buffer_number;
unsigned short l4_relay_baton;
unsigned short l4_data_start_offset;
unsigned short size;
unsigned char l4_code;
unsigned char path;
```

Description

The *send_n_prmtv_below* emulate routine passes a specified interlayer message buffer from Layer 4 to Layer 3 in an OSI primitive.

Inputs

The first parameter is the interlayer buffer number to be sent. For a buffer which has been received at Layer 4 from Layer 5, the variable *up_t_il_buff* may be used to identify the buffer number. If the buffer originated at Layer 4, use the buffer-number variable named in the *_get_il_msg_buff* routine. (See *_insert_il_buff_list_cnt* routine example at Layer 5.)

The second parameter is the returned maintain bit from a call to *_set_maint_buff_bit*. It is used only to pass a received buffer from Layer 4 to Layer 3. As soon as Layer 3 processing on the buffer is completed, the bit is automatically freed. If the buffer originated at Layer 4, use the maintain bit variable named in the *_get_il_msg_buff* routine. (See *_insert_il_buff_list_cnt* routine example at Layer 5.)

The third parameter is the offset to the Layer 4 list header node in the buffer. For a buffer which has been received at Layer 4 from Layer 5, the variable *up_t_sdu* may be used to indicate the offset.

The fourth parameter is the size of the data in the buffer. It will always be set to zero since the data length is unknown in a primitive being passed down the layers.

The fifth parameter is the code specifying the type of primitive in which the buffer will be sent. Refer to variable *up_n_prmtv_code* in Table 66-4 for the appropriate primitive code.

The sixth parameter is the path number along which the buffer will be sent. For a buffer which has been received at Layer 4 from Layer 5, the variable *up_t_prmtv_path* may be used to specify the path number.

Example

A buffer is received at Layer 4 from Layer 5. No text will be inserted at Layer 4. (For information on inserting text, see *_insert_il_buff_list_cnt* routine.) The buffer will be passed to Layer 3, requiring a new maintain bit to be set. If values are entered for the code and path, variables for code and path need not be declared.

```
{
 extern volatile unsigned short up_t_il_buff;
 extern volatile unsigned short up_t_sdu;
 unsigned short l4_relay_baton;
}
LAYER: 4
     STATE: pass_buffer_down
        CONDITIONS: T_DATA REQ
        ACTIONS:
        {
         _set_maint_buff_bit(up_t_il_buff, &l4_relay_baton);
         send_n_prmtv_below(up_t_il_buff, l4_relay_baton, up_t_sdu, 0, 0x64, 0);
        }
```

(F) Layer 5 OSI Routines

The following routines pass OSI primitives from Layer 5 to either Layer 6 or Layer 4.

send_s_prmtv_above

Synopsis

```
extern void send_s_prmtv_above(il_buffer_number, l5_relay_baton, l5_data_start_offset, size,
   l5_code, path);
unsigned short il_buffer_number;
unsigned short l5_relay_baton;
unsigned short l5_data_start_offset;
unsigned short size;
unsigned char l5_code;
unsigned char path;
```

Description

The *send_s_prmtv_above* emulate routine passes a specified inter-layer message buffer from Layer 5 to Layer 6 in an OSI primitive.

Inputs

The first parameter is the inter-layer buffer number to be sent. For a buffer which has been received at Layer 5 from Layer 4, the variable *lo_t_il_buff* may be used to identify the buffer number.

The second parameter is the returned maintain bit from a call to *_set_maint_buff_bit*. It is used only to pass a received buffer from Layer 5 to Layer 6. As soon as Layer 6 processing on the buffer is completed, the bit is automatically freed.

The third parameter is the offset to the Layer 5 service data unit in a received buffer. The variable *lo_t_sdu* contains the offset to the service data unit when the buffer reached Layer 5. The offset must be incremented by the length of the Layer 5 header, if any.

> NOTE: In general, do not modify *extern* variables, such as *lo_t_sdu*, which may be updated by other processes. Name another variable, assign it the same value, and then increment that variable. Or, after *lo_t_sdu* has been named in the argument of the *send* routine, add the length of the Layer 5 header, if any.

The fourth parameter is the length of the data in the buffer. Use the length indicated in the *pdu* structure—*pdu.data_length*. Then subtract the length of the Layer 5 header, if any.

The fifth parameter is the code specifying the type of primitive in which the buffer will be sent. Refer to variable *lo_s_prmtv_code* in Table 66-7 for the appropriate primitive code.

The sixth parameter is the path number along which the buffer will be sent. For a buffer which has been received at Layer 5 from Layer 4, the variable *lo_t_prmtv_path* may be used to specify the path number.

Example

A buffer is received at Layer 5 from Layer 4. The offset to and size of the service data unit will be adjusted if needed, a new maintain bit will be set, and the buffer will be passed up to Layer 6.

```
{
 struct pdu
  {
   unsigned char primitive_code;
   unsigned char path;
   unsigned long parameter;
   unsigned short relay_baton;
   unsigned short il_buffer_number;
   unsigned char buffer_contents;
   unsigned short data_start_offset;
   unsigned short data_length;
  };
 struct pdu * pdu_ptr;
 extern volatile unsigned short lo_t_pdu_seg;
 extern volatile const unsigned char lo_t_prmtv_path;
 extern volatile unsigned short lo_t_il_buff;
 extern volatile unsigned short lo_t_sdu;
 unsigned short l5_relay_baton;
}
```

```
LAYER: 5
     STATE: send_buffer_up
        CONDITIONS: T_DATA IND
        ACTIONS:
        {
         pdu_ptr = (void *)((long)lo_t_pdu_seg << 16);
         _set_maint_buff_bit(lo_t_il_buff, &l5_relay_baton);
         send_s_prmtv_above(lo_t_il_buff, l5_relay_baton, lo_t_sdu, pdu_ptr->data_length,
              0xa5, lo_t_prmtv_path);
        }
```

send_m_s_prmtv_above

Synopsis

```
extern void send_m_s_prmtv_above(il_buffer_number, l5_relay_baton, l5_data_start_offset,
   size, l5_code, path);
unsigned short il_buffer_number;
unsigned short l5_relay_baton;
unsigned short l5_data_start_offset;
unsigned short size;
unsigned char l5_code;
unsigned char path;
```

Description

The *send_m_s_prmtv_above* monitor routine passes a specified inter-layer message buffer from Layer 5 to Layer 6 in an OSI monitor primitive.

Inputs

See *send_s_prmtv_above*. Use the monitor *m_lo_t_il_buff*, *m_lo_t_sdu_offset*, and *m_lo_t_sdu_size* variables as input. Refer to variable *m_lo_s_prmtv_code* in Table 66-7 for the appropriate primitive code.

Example

Make the appropriate variable declarations. For a condition monitoring RD data primitives, the Layer 5 programming block should look like this:

```
LAYER: 5
     STATE: send_buffer_up
        CONDITIONS: T_RD_DATA IND
        ACTIONS:
        {
         _set_maint_buff_bit(m_lo_t_il_buff, &l5_relay_baton);
         send_m_s_prmtv_above(m_lo_t_il_buff, l5_relay_baton,m_ lo_t_sdu_offset,
              m_lo_t_sdu_size, 0xa5, m_lo_t_prmtv_path);
        }
```

send_t_prmtv_below

Synopsis

```
extern void send_t_prmtv_below(il_buffer_number, l5_relay_baton, l5_data_start_offset, size,
   l5_code, path);
unsigned short il_buffer_number;
unsigned short l5_relay_baton;
unsigned short l5_data_start_offset;
unsigned short size;
unsigned char l5_code;
unsigned char path;
```

Description

The *send_t_prmtv_below* emulate routine passes a specified inter-layer message buffer from Layer 5 to Layer 4 in an OSI primitive.

Inputs

The first parameter is the inter-layer buffer number to be sent. For a buffer which has been received at Layer 5 from Layer 6, the variable *up_s_il_buff* may be used to identify the buffer number. If the buffer originated at Layer 5, use the buffer-number variable named in the *_get_il_msg_buff* routine. (See *_insert_il_buff_list_cnt* routine example at Layer 5.)

The second parameter is the returned maintain bit from a call to *_set_maint_buff_bit*. It is used only to pass a received buffer from Layer 5 to Layer 4. As soon as Layer 4 processing on the buffer is completed, the bit is automatically freed. If the buffer originated at Layer 5, use the maintain bit variable named in the *_get_il_msg_buff* routine. (See *_insert_il_buff_list_cnt* routine example at Layer 5.)

The third parameter is the offset to the Layer 5 list header node in the buffer. For a buffer which has been received at Layer 5 from Layer 6, the variable *up_s_sdu* may be used to indicate the offset.

The fourth parameter is the size of the data in the buffer. It will always be set to zero since the data length is unknown in a primitive being passed down the layers.

The fifth parameter is the code specifying the type of primitive in which the buffer will be sent. Refer to variable *up_t_prmtv_code* in Table 66-5 for the appropriate primitive code.

The sixth parameter is the path number along which the buffer will be sent. For a buffer which has been received at Layer 5 from Layer 6, the variable *up_s_prmtv_path* may be used to specify the path number.

Example

A buffer is received at Layer 5 from Layer 6. No text will be inserted at Layer 5. (For information on inserting text, see *_insert_il_buff_list_cnt* routine.) The buffer will be passed to Layer 4, requiring a new maintain bit to be set. If values are entered for the code and path, variables for code and path need not be declared.

```
{
 extern volatile unsigned short up_s_il_buff;
 extern volatile unsigned short up_s_sdu;
 unsigned short l5_relay_baton;
}
LAYER: 5
     STATE: pass_buffer_down
        CONDITIONS: S_DATA REQ
        ACTIONS:
        {
         _set_maint_buff_bit(up_s_il_buff, &l5_relay_baton);
         send_t_prmtv_below(up_s_il_buff, l5_relay_baton, up_s_sdu, 0, 0x84, 0);
        }
```

(G) Layer 6 OSI Routines

The following routines pass OSI primitives from Layer 6 to either Layer 7 or Layer 5.

send_p_prmtv_above

Synopsis

```
extern void send_p_prmtv_above(il_buffer_number, l6_relay_baton, l6_data_start_offset, size,
   l6_code, path);
unsigned short il_buffer_number;
unsigned short l6_relay_baton;
unsigned short l6_data_start_offset;
unsigned short size;
unsigned char l6_code;
unsigned char path;
```

Description

The *send_p_prmtv_above* emulate routine passes a specified interlayer message buffer from Layer 6 to Layer 7 in an OSI primitive.

Inputs

The first parameter is the interlayer buffer number to be sent. For a buffer which has been received at Layer 6 from Layer 5, the variable *lo_s_il_buff* may be used to identify the buffer number.

The second parameter is the returned maintain bit from a call to *_set_maint_buff_bit*. It is used only to pass a received buffer from Layer 6 to Layer 7. As soon as Layer 7 processing on the buffer is completed, the bit is automatically freed.

The third parameter is the offset to the Layer 6 service data unit in a received buffer. The variable *lo_s_sdu* contains the offset to the service data unit when the buffer reached Layer 6. The offset must be incremented by the length of the Layer 6 header, if any.

> NOTE: In general, do not modify *extern* variables, such as *lo_s_sdu*, which may be updated by other processes. Name another variable, assign it the same value, and then increment that variable. Or, after *lo_s_sdu* has been named in the argument of the *send* routine, add the length of the Layer 6 header, if any.

The fourth parameter is the length of the data in the buffer. Use the length indicated in the *pdu* structure—*pdu.data_length*. Then subtract the length of the Layer 6 header, if any.

The fifth parameter is the code specifying the type of primitive in which the buffer will be sent. Refer to variable *lo_p_prmtv_code* in Table 66-8 for the appropriate primitive code.

The sixth parameter is the path number along which the buffer will be sent. For a buffer which has been received at Layer 6 from Layer 5, the variable *lo_s_prmtv_path* may be used to specify the path number.

Example

A buffer is received at Layer 6 from Layer 5. The offset to and size of the service data unit will be adjusted if needed, a new maintain bit will be set, and the buffer will be passed up to Layer 7.

```
{
 struct pdu
  {
   unsigned char primitive_code;
   unsigned char path;
   unsigned long parameter;
   unsigned short relay_baton;
   unsigned short ll_buffer_number;
   unsigned char buffer_contents;
   unsigned short data_start_offset;
   unsigned short data_length;
  };
 struct pdu * pdu_ptr;
 extern volatile unsigned short lo_s_pdu_seg;
 extern volatile const unsigned char lo_s_prmtv_path;
```

```
 extern volatile unsigned short lo_s_il_buff;
 extern volatile unsigned short lo_s_sdu;
 unsigned short l6_relay_baton;
}
LAYER: 6
     STATE: send_buffer_up
        CONDITIONS: S_DATA IND
        ACTIONS:
        {
         pdu_ptr = (void *)((long)lo_s_pdu_seg << 16);
         _set_maint_buff_bit(lo_s_il_buff, &l6_relay_baton);
         send_p_prmtv_above(lo_s_il_buff, l6_relay_baton, lo_s_sdu, pdu_ptr->data_length,
              0xc5, lo_s_prmtv_path);
        }
```

send_m_p_prmtv_above

Synopsis

```
extern void send_m_p_prmtv_above(il_buffer_number, l6_relay_baton, l6_data_start_offset,
   size, l6_code, path);
unsigned short il_buffer_number;
unsigned short l6_relay_baton;
unsigned short l6_data_start_offset;
unsigned short size;
unsigned char l6_code;
unsigned char path;
```

Description

The *send_m_p_prmtv_above* monitor routine passes a specified interlayer message buffer from Layer 6 to Layer 7 in an OSI monitor primitive.

Inputs

See *send_p_prmtv_above*. Use the monitor variables *m_lo_s_il_buff*, *m_lo_s_sdu_offset*, and *m_lo_s_sdu_size* as input. Refer to variable *m_lo_p_prmtv_code* in Table 66-8 for the appropriate primitive code.

Example

Make the appropriate variable declarations. For a condition monitoring RD data primitives, the Layer 6 programming block should look like this:

```
LAYER: 6
     STATE: send_buffer_up
        CONDITIONS: S_RD_DATA IND
        ACTIONS:
        {
         _set_maint_buff_bit(m_lo_s_il_buff, &l6_relay_baton);
         send_m_p_prmtv_above(m_lo_s_il_buff, l6_relay_baton,m_ lo_s_sdu_offset,
              m_lo_s_sdu_size, 0xc5, m_lo_s_prmtv_path);
        }
```

send_s_prmtv_below

Synopsis

```
extern void send_s_prmtv_below(il_buffer_number, l6_relay_baton, l6_data_start_offset, size,
   l6_code, path);
unsigned short il_buffer_number;
unsigned short l6_relay_baton;
unsigned short l6_data_start_offset;
unsigned short size;
unsigned char l6_code;
unsigned char path;
```

Description

The *send_s_prmtv_below* emulate routine passes a specified interlayer message buffer from Layer 6 to Layer 5 in an OSI primitive.

Inputs

The first parameter is the interlayer buffer number to be sent. For a buffer which has been received at Layer 6 from Layer 7, the variable *up_p_il_buff* may be used to identify the buffer number. If the buffer originated at Layer 6, use the buffer-number variable named in the *_get_il_msg_buff* routine. (See *_insert_il_buff_list_cnt* routine example at Layer 5.)

The second parameter is the returned maintain bit from a call to *_set_maint_buff_bit*. It is used only to pass a received buffer from Layer 6 to Layer 5. As soon as Layer 5 processing on the buffer is completed, the bit is automatically freed. If the buffer originated at Layer 6, use the maintain bit variable named in the *_get_il_msg_buff* routine. (See *_insert_il_buff_list_cnt* routine example at Layer 5.)

The third parameter is the offset to the Layer 6 list header node in the buffer. For a buffer which has been received at Layer 6 from Layer 7, the variable *up_p_sdu* may be used to indicate the offset.

The fourth parameter is the size of the data in the buffer. It will always be set to zero since the data length is unknown in a primitive being passed down the layers.

The fifth parameter is the code specifying the type of primitive in which the buffer will be sent. Refer to variable *up_s_prmtv_code* in Table 66-6 for the appropriate primitive code.

The sixth parameter is the path number along which the buffer will be sent. For a buffer which has been received at Layer 6 from Layer 7, the variable *up_p_prmtv_path* may be used to specify the path number.

Example

A buffer is received at Layer 6 from Layer 7. No text will be inserted at Layer 6. (For information on inserting text, see *_insert_il_buff_list_cnt* routine.) The buffer will be passed to Layer 5, requiring a new maintain bit to be set. If values are entered for the code and path, variables for code and path need not be declared.

```
{
 extern volatile unsigned short up_p_il_buff;
 extern volatile unsigned short up_p_sdu;
 unsigned short l6_relay_baton;
}
LAYER: 6
     STATE: pass_buffer_down
        CONDITIONS: P_DATA REQ
        ACTIONS:
        {
         _set_maint_buff_bit(up_p_il_buff, &l6_relay_baton);
         send_s_prmtv_below(up_p_il_buff, l6_relay_baton, up_p_sdu, 0, 0xa4, 0);
        }
```

(H) Layer 7 OSI Routines

send_p_prmtv_below

Synopsis

```
extern void send_p_prmtv_below(il_buffer_number, relay_baton, data_start_offset, size, code,
   path);
 unsigned short il_buffer_number;
unsigned short relay_baton;
unsigned short data_start_offset;
unsigned short size;
unsigned char code;
unsigned char path;
```

Description

The *send_p_prmtv_below* emulate routine passes a specified interlayer message buffer from Layer 7 to Layer 6 in an OSI primitive.

Inputs

The first parameter is the interlayer buffer number to be sent. Use the buffer-number variable named in the *_get_il_msg_buff* routine. (See *_insert_il_buff_list_cnt* routine example at Layer 5.)

The second parameter is the returned maintain bit from the call to *_get_il_msg_buff*.

The third parameter is the returned offset (from a call to *_start il_buff_list*) to the Layer 7 list header node in the buffer.

The fourth parameter is the size of the data in the buffer. It will always be set to zero since the data length is unknown in a primitive being passed down the layers.

The fifth parameter is the code specifying the type of primitive in which the buffer will be sent. Refer to variable *up_p_prmtv_code* in Table 66-7 for the appropriate code.

The sixth parameter is the path number along which the buffer will be sent.

Example

A buffer is obtained at Layer 7. The buffer will be passed to Layer 6, without any data inserted. (For information on inserting text, see *_insert_il_buff_list_cnt* routine.) If values are entered for the code and path, variables for code and path need not be declared.

```
{
 unsigned short il_buffer_number;
 unsigned short data_start_offset;
 unsigned short relay_baton;
}
LAYER: 7
     STATE: pass_buffer_down
        CONDITIONS: KEYBOARD " "
        ACTIONS:
        {
         _get_il_msg_buff(&il_buffer_number, &relay_baton);
         _start_il_buff_list(il_buffer_number, &data_start_offset);
         send_p_prmtv_below(il_buffer_number, relay_baton, data_start_offset, 0, 0xc4, 0);
        }
```

67 Print

The PRINTER port is a serial interface through which the programmer may direct output from the INTERVIEW to a printer. The printer port is located at the rear of the INTERVIEW between the REMOTE RS-232 and AUXILIARY ports.

> NOTE: Before directing output to the printer port, configure the Printer Setup menu as explained in Section 15.2.

Each spreadsheet PRINT action or call to one of the C print routines causes output to be added to a queue of unprinted text in the print buffer. If not doing so already, the print server also begins to poll the print buffer for text to print. As long as there is unprinted text in the buffer, the print server polls the buffer, removes text, and sends it to the printer port of the INTERVIEW. Use the *_print_buffer* structure to monitor the flow of text in and out of the print buffer.

Use any of the four C print routines explained in this section to add text to the print buffer. Three of them—*printc*, *printf*, and *prints*—are similar to the *displayc*, *displayf*, and *displays* routines which direct output to the Display Window. See Section 64.3(C). With the *set_print_header* routine, you determine the heading which will appear at the top of each printed page. One other routine, *sprintf*, writes output to a string. The string can then be referenced in subsequent calls to *printf*. (You may also use the string named in *sprintf* in calls to *displayf*, *tracef*, or *fprintf*.)

67.1 Structures

Refer to Table 67-1 for the structure of the print buffer. Compare *_print_buffer.in* with *_print_buffer.out* to determine whether or not the print buffer has emptied. When the values of these two variables are equal, the buffer is empty.

> NOTE: Consider the variables in the *_print_buffer* structure read-only variables. In general, do not modify *extern* structures or variables which may be updated by other processes.

At times, processes may add transactions to the print buffer more quickly than the print server takes them out. If a process cannot add to the buffer without overwriting unprinted text, a buffer overrun occurs. When your INTERVIEW is configured for

data playback, you can minimize print-buffer overruns by periodically suspending playback and allowing the print server to empty the buffer. In judging how often to suspend playback, keep in mind the following points: 1) In general, the more conditions a program has that trigger print actions, the more frequently playback should be suspended. 2) When planning to print Run-mode buffers, remember that the faster the playback speed, the quicker the print buffer fills.

Table 67-1
Print Structures

Type	Variable	Value (hex/decimal)	Meaning
Structure Name: print_buffer			Structure of the print buffer. Declared as type *struct*.
unsigned short	in	*a-207/10-8199*	offset into the print buffer (from the physical beginning of the buffer) to the location where next transaction text will be added. Advances with each spreadsheet PRINT action or call to a C print routine. When *in* equals *out*, the print buffer is empty.
unsigned short	out	*a-207/10-8199*	offset into the print buffer (from the physical beginning of the buffer) to the last transaction text printed from the buffer. Advances each time text is actually sent out the printer port of the INTERVIEW. When *out* equals *in*, the print buffer is empty.
unsigned short	buffer_end	209/8201	offset to the physical end of the print buffer—i.e., to the end of the array named *buffer* (see below)
unsigned short	lock		when process is printing, locks out other processes from accessing the print buffer
char	polling	0 *non-zero*	print server is not polling print server is polling print buffer for text to print
char	overrun	0 *non-zero*	print buffer is not in overrun state print buffer is in overrun state—i.e., a process attempting to add text to the print buffer can't because unprinted text in the buffer would be overwritten. Following message will appear on printout: "*print buffer overrun has occurred.*"
char	buffer [8192]		array of text transactions
Structure Name: _print_buffer			An instance of the *print_buffer* structure, declared as type *extern struct print_buffer*. Use the variables contained in this structure to monitor flow of text in and out of the print buffer. Reference structure variables as follows: *_print_buffer.in*.

The following example shows how you might use a TIMEOUT condition to check the print buffer periodically. Each time the timeout expires, the program determines whether or not the buffer is half full. If so, playback is suspended. If the buffer is only one-quarter full, playback is resumed. (Other conditions in the program, not illustrated here, would cause print actions to send output to the print buffer.)

```
{
 #define PRINT_BUFFER_SZ 8192
 #define STOP_POINT (PRINT_BUFFER_SZ/2)
 #define START_POINT (PRINT_BUFFER_SZ/4)
}
LAYER: 1
     {
      struct print_buffer
       {
        unsigned short in;
        unsigned short out;
        unsigned short buffer_end;
        unsigned short lock;
        char polling;
        char overrun;
       };
      extern struct print_buffer _print_buffer;
      int crnt_buffer_sz;
     }
     STATE: check_print_buffer
            CONDITIONS: ENTER_STATE
            ACTIONS: TIMEOUT ck_buffer RESTART 0.01
            CONDITIONS: TIMEOUT ck_buffer
            ACTIONS:
            {
             crnt_buffer_sz = ((_print_buffer.in + PRINT_BUFFER_SZ) - _print_buffer.out) %
                   PRINT_BUFFER_SZ;
             if(crnt_buffer_sz > STOP_POINT)
               suspend_rcrd_play();
             else if(crnt_buffer_sz < START_POINT)
               start_rcrd_play();
            }
            TIMEOUT ck_buffer RESTART 0.01
```

67.2 Variables

There are no variables associated exclusively with print functions.

67.3 Routines

printc

Synopsis

```
extern void printc(character);
const char character;
```

Description

The *printc* routine outputs a single ASCII character to the print buffer for printing, converting the value provided as the argument into its ASCII equivalent. Decimal and octal values are converted to hexadecimal format before the ASCII equivalent is sought.

Inputs

The only parameter is a numerical value. The value may be given as a hexadecimal, octal, or decimal constant; as an alphanumeric constant inside of single quotes; or as a variable. A hexadecimal value must be preceded by the prefix 0x or 0X; an octal value must be preceded by the prefix 0. If no prefix appears before the input, the number is assumed to be decimal. Valid numeric entries are 00 to 127, decimal. An alphanumeric character placed between single quotes will be output as is to the printer.

Example

The *printc* entries on the left output the printed character given on the right:

```
printc('a');        a
printc(65);         A
printc(0x65);       e
printc(065);        5
```

printf

Synopsis

```
extern int printf(format_ptr, . . . );
const char * format_ptr;
```

Description

The *printf* routine writes output to the print buffer for printing, under control of the string pointed to by *format_ptr* that specifies how subsequent arguments are converted for output. If there are insufficient arguments for the format, the behavior is

undefined. If the format is exhausted while arguments remain, the excess arguments are evaluated but otherwise ignored. The *printf* routine returns when the end of the format string is encountered.

Inputs

The format is composed of zero or more directives: ordinary characters (not %), which are copied unchanged to the output stream; and conversion specifications, each of which results in fetching zero or more subsequent arguments. Each conversion specification is introduced by the character %. After the %, the following appear in sequence:

- Zero or more *flags* that modify the meaning of the conversion specification. The flag characters and their meanings are:

  - The result of the conversion will be left-justified within the field.

  + The result of a signed conversion will always begin with a plus or minus sign.

  space If the first character of a signed conversion is not a sign, a space will be prepended to the result. If the *space* and + flags both appear, the *space* flag will be ignored.

  # The result is to be converted to an "alternate form." For d, i, u, c, and s conversions, the flag has no effect. For o conversion, it increases the precision to force the first digit of the result to be a zero. For x (or X) conversion, a nonzero result will have 0x (or 0X) prepended to it.

- An optional decimal integer specifying a minimum *field width*. If the converted value has fewer characters than the field width, it will be padded on the left (or right, if the left adjustment flag, described above, has been given) to the field width. The padding is with spaces unless the field width integer starts with a zero, in which case the padding is with zeros.

- An optional *precision* that gives the minimum number of digits to appear for the d, i, o, u, x, and X conversions, or the maximum number of characters to be written from an array in an s conversion. The precision takes the form of a period (.) followed by an optional decimal integer; if the integer is omitted, it is treated as zero. The amount of padding specified by the precision overrides that specified by the field width.

- An optional h specifying that a following d, i, o, u, x, or X conversion specifier applies to a *short int* or *unsigned short int* argument (the argument will have been promoted according to the integral promotions, and its value shall be converted to *short int* or *unsigned short int* before printing); or an optional l specifying that a following d, i, o, u, x, or X conversion specifier applies to a *long int* or *unsigned long int* argument. If an h or l appears with any other conversion specifier, it is ignored.

- A character that specifies the type of *conversion* to be applied. (Special AR extensions have been added.) The conversion specifiers and their meanings are:

  d, i, o, u, x, X

  The *int* argument is converted to signed decimal (d or i), unsigned octal (o), unsigned decimal (u), or unsigned hexadecimal notation (x or X); the letters abcdef are used for x conversion and the letters ABCDEF for X conversion. The precision specifies the minimum number of digits to appear; if the value being converted can be represented in fewer digits, it will be expanded with leading zeros. The default precision is 1. The result of converting a zero value with a precision of zero is no characters.

  c The *int* argument is converted to an *unsigned char*, and the resulting character is written.

  s The argument shall be a pointer to a null-terminated array of 8-bit *chars*. Characters from the string are printed up to (but not including) the terminating null character: if the precision is specified, no more than that many characters are printed. The string may be an array into which output was written via the *sprintf* routine.

  p The argument shall be a pointer to void. The value of the pointer is converted to a sequence of printable characters, in this format: *0000:0000*. There are always exactly 4 digits to the right of the colon. The number of digits to the left of the colon is determined by the pointer's value and the precision specified. Use this conversion to print 80286 memory addresses. The segment number will appear to the left of the colon and the offset to the right.

  % A % is written. No argument is converted.

  \n Writes hexadecimal 0D 0A, the ASCII carriage-return and linefeed characters. No argument is converted.

If a conversion specification is invalid, the behavior is undefined.

If any argument is or points to an aggregate (except for an array of characters using *%s* conversion or any pointer using *%p* conversion), the behavior is undefined.

In no case does a nonexistent or small field width cause truncation of a field; if the result of a conversion is wider than the field width, the field is expanded to contain the conversion result.

Returns

The *printf* routine returns the number of characters output.

Example

To print a date and time in the form "Sunday, July 3, 10:02," where weekday and month are pointers to strings:

```
LAYER: 1
{
 unsigned char date_time [100];
 unsigned char weekday [10];
 unsigned char month [10];
 unsigned short day;
 unsigned char hour;
 unsigned char min;
}
      STATE: output_to_printer
          CONDITIONS: KEYBOARD " "
          ACTIONS:
          {
           printf( "%s, %s %d, %.2d:%.2d\n", weekday, month, day, hour, min);
          }
```

sprintf

Synopsis

```
extern int sprintf(string_ptr, format_ptr);
unsigned char string [128];
const char * format_ptr;
```

Description

The *sprintf* routine is similar to the *printf* routine, except that *sprintf* writes output to a string, while *printf* writes output directly to the print buffer for printing. The *sprintf* routine is useful for writing formatted output to a display, printer, or file.

The output is under control of the string pointed to by *format_ptr* that specifies how subsequent arguments are converted for output. If there are insufficient arguments for the format, the behavior is undefined. If the format is exhausted while arguments remain, the excess arguments are evaluated but otherwise ignored. The *sprintf* routine returns when the end of the format string is encountered.

Inputs

The first parameter is a pointer to the array to which output will be written.

For the second parameter, see *printf* routine.

Returns

This routine returns the number of characters written into the array, not counting the added null terminating character.

Example

Refer again to the sample program for the *displayf* routine in Section 64.3(C). This time you also want to send the output to a printer. By using the *sprintf* routine, you only have to enter the format string once.

```
LAYER: 1
{
 unsigned char date_time [100];
 unsigned char weekday [10];
 unsigned char month [10];
 unsigned short day;
 unsigned char hour;
 unsigned char min;
}
      STATE: output_to_display_window_and_printer
         CONDITIONS: KEYBOARD " "
         ACTIONS:
         {
          sprintf(date_time, "%s, %s %d, %.2d: %.2d\n", weekday, month, day, hour,
                min);
          displayf("%s", date_time);
          printf("%s", date_time);
         }
```

set_print_header

Synopsis

```
extern int set_print_header(format_ptr);
const char * format_ptr;
```

Description

This routine writes output to the print buffer, to be printed after each form feed, under control of the string pointed to by *format_ptr*. Paging is done automatically by the INTERVIEW. The *set_print_header* routine returns when the end of the format string is encountered.

Inputs

The format is composed of zero or more ordinary characters. Octal or hexadecimal values also may be input, with octal preceded by \ and hex by \x. Pad each value to three integers with leading zeroes.

The status information shown above the prompt line on the display screens of the INTERVIEW can be sent to a printer with the following inputs:

#d	date	(mm/dd/yy)
#t	time	(hh:mm)
#p	page (not shown on the display screens)	
#b	block number	
##	#	

Returns

The *set_print_header* routine returns the length of the header (0-255), or a -1 if the header exceeds the buffer size.

Example

If you want the date, time, and page number to appear in the heading on each page sent to a printer, enter the following:

```
LAYER: 2
    STATE: header
        CONDITIONS: ENTER_STATE
        ACTIONS:
        {
         set_print_header("#### #d  #t                          #p ####\n");
        }
```

The printer output will look like this:

```
## 09/01/89  09:30                                    Page :  1 ##

.
.
.
## 09/01/89  09:31                                    Page :  2 ##

.
.
.
```

reset_print_page

Synopsis

```
extern int reset_print_page();
```

Description

The *reset_print_page* routine resets the INTERVIEW's automatic page numbering for printer output to 1.

Returns

If the page number is successfully reset, the routine returns zero. If the print buffer is overrun, it returns -1.

Example

In the following example, a header with page numbering is assigned to printed output. (See *set_print_header* routine above.) Elsewhere in the program (not shown) the programmer has designated text to be printed. When the user presses the spacebar, a new header will appear on the next page output to the printer. That output will begin again with page 1.

```
LAYER: 1
    STATE: print_output
        CONDITIONS: ENTER_STATE
        ACTIONS:
        {
         set_print_header("#d  #t          First Header          #p\n");
        }
        CONDITIONS: KEYBOARD " "
        ACTIONS:
        {
         set_print_header("#d  #t          New Header          #p\n");
         reset_print_page();
        }
```

prints

Synopsis

```
extern void prints(string_ptr);
const char * string_ptr;
```

Description

The *prints* routine is similar to the *displays* routines, except that *prints* writes output to the print buffer for printing while *displays* writes output to the Display Window. The output is under control of the string pointed to by the argument. The *prints* routine returns when the end of the string is encountered. The softkey equivalent of this routine is the PRINT PROMPT action on the Protocol Spreadsheet. A PRINT PROMPT action automatically time-stamps the output. Although *prints* does not, you can create your own time or date stamp with *set_print_header*.

Inputs

The input is a pointer to a string composed of zero or more ordinary characters. The newline nonliteral sequence "\n" writes hex 0D 0A (ASCII ␍␊) to the output string. Octal or hexadecimal values also may be included in the string, with octal preceded by \ and hex by \x. Pad each value to three integers with leading zeroes.

Example

The following entry

```
LAYER: 1
    STATE: print_message
        CONDITIONS: KEYBOARD " "
        ACTIONS:
        {
         prints("End of test.");
        }
```

produces the following output to a printer:

End of test.