2. A macro function ReceiveStr (&Str) uses function 'void portIO_ISR_Input (*portIOdata)' to return an input string Str. The 'portIO_ISR_input' receives the characters one by one from the port on successive calls. Similarly, a macro function SendStr (&ApplStr) is used by the function void Port_OutStr (unsigned char [ ] *applStr) to send an output string. portIO_ISR_Output sends a character to the port.
3. task_ReadPort begins only when a semaphore *SigReset* is posted by the resetTask. (a) task_ReadPort takes the message from the queue MsgQStart and gets the pending queue messages, *requestHeader* and *requestStart*. It encrypts these two strings and sends to the Port_IO, which transmits it to the host through the transceiver or modem. It receives host message hostStr through the transceiver. Host specifies, by the hostStr, the host PIN. This PIN is the one used for the bank authorization PIN of the card. (b) It posts a semaphore *SemPW* flag to the waiting task task_PW if the presently running task verifies the hostPIN. It waits for a message from the queue MsgQPW and receives *userPW* after deciphering the port input data string. (c) It posts semaphore *SemAppl,* in case the user password stored in a file at the EEPROM is verified. It posts a different semaphore *SemAppl,* if it verifies the user password. It sends in the end a close request message into a queue MsgQApplClose. Message is *requestApplClose* to the Port_IO and receives encrypted string "*Closure Permitted*". The tasks delete on deciphering.
4. task_PW after encryption on taking the pending *SemPW* is to send the string *requestPW.* When it takes *SemPW,* it sends the *requestPW* into the MsgQPW. task_ReadPort will send it to the host through the IO Port, Port_IO, in order to identify the user at the host.
5. task_Appl runs on taking the semaphore *SemAppl* and executes the operations. The operation may (i) modify user password, (ii) print mini statement of bank account of the user, (iii) eject requisite cash from the host, (iv) request for accepting the envelope with cash, (v) request for a print of this transaction and (vi) request for a transfer to another party. It interacts through task_ReadPort by sending the messages through the queue MsgQAppl.

The task synchronization model is also shown in Figure 12.19.

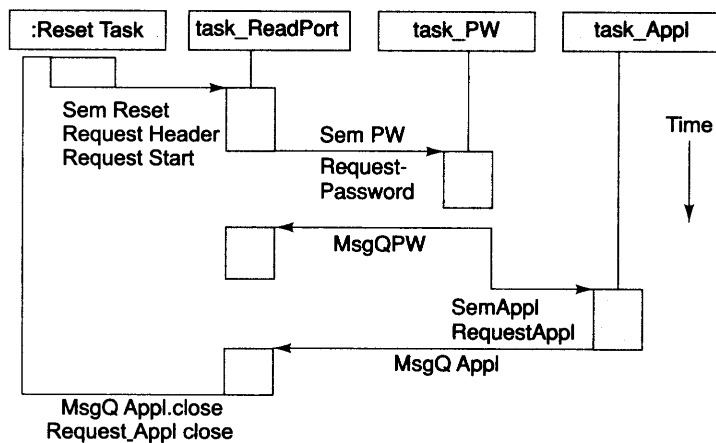


Fig. 12.19 Tasks and the synchronization model

12.4.5 Exemplary Codes

Example 12.2 gives the exemplary coding procedure for an application of this card.

Example 12.2

```
1. /* Preprocessor definitions for maximum number of interprocess events to let the SmartOS allocate memory for the Event Control Blocks */
#define SmartOS_MAX_EVENTS 24/* Let maximum IPC events be 24 */
#define SmartOS_SEM_EN 1/* Enable inclusion of semaphore functions in application. */
#define SmartOS_Q_EN 1/* Enable inclusion of queue functions for sending the string pointers to task_ReadPort * /
#define SmartOS_task_Del_En = 0 /* Disable task deletion by SmartOS at the start. */
/* End of preprocessor commands for enabling IPC functions of the SmartOS*/
2. /* Specify all user prototype of the reset task function that is called by the main function and is to be scheduled by SmartOS first at the start. In Step 11, we will be creating all other tasks within the reset task. Remember: Static means permanent memory allocation. */
static void resetTask (void *taskPointer);
static SmartOS_STK_resetTaskStack [resetTask_StackSize];
3. /* Define public variable of the task service and timing functions */
#define SmartOS_TASK_IDLE_STK_SIZE 100 /* Let memory allocation for an idle state task stack size be 100*/
#define SmartOS_TICKS_PER_SEC 1000 / * Let the number of ticks be 1000 per second. An RTCSWT will interrupt and thus tick every 1 ms to update counts. */
#define resetTask_Priority 1 /* Define reset task in main priority */
#define resetTask_StackSize 100 /* Define reset task in main stack size */
STAF _In = 0; /*Define flag for signaling modem interrupt for receiving a character. */
STAF_Out = 0; /*Define flag for signal from a modem interrupt after sending a character. */
/*-----------------------------------------------------------------------------------------*/
3. /* Prototype definitions for three tasks for the car application codes after reset. */
static void task_ReadPort (void *taskPointer);
static void task_PW (void *taskPointer);
static void task_Appl (void *taskPointer);
4. /* Definitions for three task stacks. */
static SmartOS_STK task_ReadPortStack [task_ReadPortStackSize];
static SmartOS_STK task_PWStack [task_PWStackSize];
static SmartOS_STK task_ApplStack [task_ApplStackSize];
5. /* Definitions for three task stack size. */
#define task_ReadPortStackSize 100 /* Define task 2 stack size*/
#define task_PWStackSize 100 /* Define task 3 stack size*/
#define task_ApplStackSize 100 /* Define task 4 stack size*/
6. /* Definitions for three task priorities. */
#define task_ReadPortPriority 2 /* Define task 2 priority */
#define task_PWPriority 3 /* Define task 3 priority */
6. /* Prototype definitions for the semaphores. */
SmartOS_EVENT *SigReset; /* First task that resets the card posts it. */
SmartOS_EVENT *SemPW; /* task_PW posts it to send request for getting user password through the host. */
```

```
SmartOS_EVENT *SemAppl; /* Needed when using Semaphore as flag for interprocess communication between task_ReadPort and task_PW. */
7. /* Prototype definitions for the queues. */
SmartOS_EVENT *MsgQStart; /* Needed for IPC between resetTask and task_ReadPort. */
void *MsgQStart [QStartMessagesSize]; /* Let the maximum number of message pointers at the queue be QStartMessagesSize. */
SmartOS_EVENT *MsgQPW; /* Needed for IPC between task_PW and task_ReadPort. */
void *MsgQPW [QPWMessagesSize]; /* Let the maximum number of message pointers at the queue be QPWMessagesSize. */
SmartOS_EVENT *MsgQAppl; /* Needed for IPC between task_Appl and task_ReadPort. */
void *MsgQAppl [QApplMessagesSize]; /* Let the maximum number of message pointers at the queue be QApplMessagesSize. */
SmartOS_EVENT *MsgQApplClose; /* Needed for IPC between task_Appl and task_ReadPort. */
void *MsgQApplClose [QApplCloseMessagesSize]; /* Let the maximum number of message pointers at the queue be QApplCloseMessagesSize. */
8. /* Define both queue array sizes. Assume a maximum of 16 strings can be sent in a queue. */
#define QStartMessagesSize 16; /* Define size of start message pointer queue when full */
#define QPWMessagesSize 16; /* Define size of password message pointer queue when full */
#define QApplMessagesSize 16; /* Define size of application message pointer queue when full */
#define QApplCloseMessagesSize 16; /* Define size of application message pointer queue when full */
9. /* Define Semaphore initial values, 0 when used as an event flag and 1 when resource key. */
SigReset = SmartOSSemCreate (0); /* Declare initial value of semaphore = 0 for using it as an event flag from resetTask. */
SemPW = SmartOSSemCreate (0); /* Declare initial value of semaphore = 0 for using it as an event flag from task_ReadPort */
SemAppl = SmartOSSemCreate (0); /* Declare initial value of semaphore = 0 for using it as an event flag from task_ReadPort */
/* Define a top of the message pointer array. QMsgPointer points to top of the Messages to start with. */
MsgQStart = SmartOSQCreate (&QStart [0], QStartMessagesSize);
MsgQPW = SmartOSQCreate (&QPW [0], QPWMessagesSize);
MsgQAppl = SmartOSQCreate (&QAppl [0], QApplMessagesSize);
MsgQApplClose = SmartOSQCreate (&QApplClose [0], QApplCloseMessagesSize);
.
.
10. /* Any other SmartOS Events for the IPCs. */
.
.
11. /* Code similar to steps for ISR_CharInt and Task_ReadPortA in Example 9.16. These were for reading from Port A and storing a character. Here, we have Port_IO. */
/* Prototype Declarations for modem Port_IO input and output strings. */
char [ ] Str; /* Port_IO input string to hold the data from the host through the demodulator circuit of modem. */
char [ ] ApplStr; /* Port_IO string, which the modem transfers to host after modulation. */
```

```
unsigned char *portIndata; unsigned char *portOutdata;
void portIO_ISR_Input (*portIndata); /* Prototype declaration for receiving an input character. */
void portIO_ISR_Output (*portOutdata); /* Prototype declaration for sending an output character. */
/* Start of Port_IO Input Interrupt Service Routine */
void portIO_ISR_Input (*portIOdata) {
disable_PortIO_InIntr ( ); /* Function for disabling another interrupt from port IO input. */
/* Insert Code for reading Port I/O bits
portIOdata = &Str;
*/
/* Start of Port_IO Output Interrupt Service Routine */
void portIO_ISR_Output (*portIOdata) {
disable_PortIO_OutIntr ( ); /* Function for disabling another interrupt from port I/O output */
/* Define a macro for sending a String */
unsigned byte i;
#define SendStr (&ApplStr) (
portIOdata = &ApplStr; i = 0; STAF_Out = 1;
while (STAF_Out = =1 && ApplStr [i] ! = NULL) {
portIOdata = ApplStr [i]; / *Pick a character from the queue message. */
portIO_ISR_Output (&portIOdata); /*Send it to the Port_IO for the modem output. */
i++; /* Be Ready for next Character */
STAF_Out =0; /*Port interrupt when one character sent by setting STAF_Out again. */
}
ApplStr = ""; /* Clear the Queue message for the new one*/
&ApplStr = ApplStr [0];
) /* End of the macro function SendStr. */
/* Define a macro function for string comparison. Note: 'C' function strcmp is available at a C library. In
order to optimize codes, we are not using strcmp library function, but our own macro here. */
#define boolean strcmp (ApplStr, Str) (
.
.
)
/*------------------------------------------------------------------------------------------*/
/* Define a macro for receiving a string */
#define ReceiveStr (&Str) (
while (STAF_In != 1) { }; i=0;
/* Execute interrupt service routines for each character received at modem Port_IO*/
while (STAF_In == 1 && Str [i] != NULL) {
        portIO_ISR_Input (&portIOdata);};
STAF_In =0; /* Remember: as soon as Port A is read STAF will reset itself to reflect
next interrupt status. */
Str [i] = portIOdata;/* Write port I/O input array element from the returned data*/
i++;}
) /* End of the macro function ReceiveStr. */
12. /* Start of the codes of the application from Main.
Note: Code steps are similar to in Example 9.16 */
```

```
void main (void) {
/* Initiate SmartOS RTOS to use OS kernel functions */
SmartOSInit ( );
13. /* Create Reset task, resetTask that must execute once before any other. task creates by defining its
Identity as resetTask, stack size and other TCB parameters. */
SmartOStaskCreate (resetTask, void (*) 0, (void *) &resetTaskStack [resetTask_StackSize],
resetTask_Priority);
14. /* Create other main tasks and interprocess communication variables if these must also execute at least
once after the resetTask. */
15. /* Start SmartOS RTOS to let us RTOS control and run the created tasks */
SmartOSStart ( );
/* Infinite while-loop exists in each task. So there is never a return from the RTOS function SmartOSStart
( ). RTOS takes the control forever. */
16. } / *** End of the Main function ***/
/*----------------------------------------------------------------------------*/
/* The codes of the application reset task that main created. */
17. static void resetTask (void *taskPointer){
18. /* Start Timer Ticks for using timer ticks later. */
SmartOSTickInit ( ); /* Function for initiating RTCSWT that starts ticks at the configured time in the
SmartOS configuration preprocessor commands in Step 1 */.
19. /* Create three tasks defined by three task identities, task_ReadPort, task_PW, task_Appl and the stack
sizes, other TCB parameters. */
SmartOStaskCreate (task_ReadPort, void (*) 0, (void *) & task_ReadPortStack [task_ReadPortStackSize],
task_ReadPortPriority);
SmartOStaskCreate (task_PW, void (*) 0, (void *) & task_PWStack [task_PWStackSize], task_PWPriority);
SmartOStaskCreate (task_Appl, void (*) 0, (void *) & task_ApplStack [task_ApplStackSize],
task_ApplPriority);
/* Declare requestHeader */
unsigned char [ ] requestHeader;
unsigned char [ ] requestStart;
20. while (1) { /* Start of the while loop*/
/* Code for retrieving two strings requestHeader for user PIN and request for host PIN string
from the protected file structure. */
.
.
/* Write the array elements after encryption. */
ApplStr = SmartOSEncrypt (requestHeader, DES); /* Using an RTOS function encrypt
requestHeader and post it in message queue. */
SmartOSQPost (MsgQStart, ApplStr);
*ApplStr = SmartOSEncrypt (requestStart, DES); /* Using an RTOS function encrypt requestStart and
post it in message queue. */
SmartOSQPost (MsgQStart, ApplStr);
SmartOSSemPost (SigReset); /* Post Semaphore event flag. */
21. /* Suspend the Reset task with no resumption later, as it must run once only for initiation of timer ticks
```

```
and for creating the tasks that the scheduler controls by preemption. */
SmartOStaskSuspend (resetTask_Priority); /*Suspend Reset task and control of the RTOS passes to other
tasks of waiting execution*/
22. } /* End of while loop */
23. } /* End of resetTask Codes */
/**************************************************************************/
24. static void task_ReadPort (void *taskPointer) {
while (1) {
25. /* Wait for IPC from resetTask. */
SmartOSSemPend (SigReset, 0, SemErrPointer);
26. /* Wait for a message for requestHeader from queue MsgQStart. */
&QStart = SmartOSQPend (MsgQStart, 0, QErrPointer);
SendStr (&QStart); /* Send it to transceiver Port_IO. Note: after sending the message in string QStart
becomes null, "". */
27. /* Wait for a message for requestStart from queue MsgQStart. */
&QStart = SmartOSQPend (MsgQStart, 0, QErrPointer);
SendStr (&QStart); /* Send it to modem Port_IO. */
/* Receive and decipher a string from the transceiver Port IO. */
ReceiveStr (&Str);
ApplStr = SmartOSDecrypt (Str, DES); /* Using an RTOS function, decrypt the input string from the host
*/
28. /* Code for saving the Host PIN and, if verified, then application commands from the host is also
saved. The savings are at the protected file structure. */
.
.
SmartOSSemPost (SemPW); /* Post event flag for requesting a password at MsgQPW. */
SmartOSTimeDly (100); /* Delay for 100 ms to allow lower priority task task_PW run. */
29. /* Wait for a message for requestPassword from queue MsgQPW. If available, send request and
wait for the password. */
&QPW = SmartOSQPend (MsgQPW, 0, QErrPointer);
SendStr (&QPW); /* Send password request to modem Port_IO. */
ReceiveStr (&Str); /* Receive a String from the modem Port IO. */
ApplStr = SmartOSDecrypt (Str, DES); /* Using an RTOS function, decrypt the input string
for password from the host */
30. /* Code for verifying the deciphered user password at the protected memory or file
data. If verified, then application commands from the host by posing event flag
SemAppl. */
.
.
31. /* Wait for a message for requestAppl from queue MsgQAppl. If available, send request and wait for
the application command and user data. */
&QAppl = SmartOSQPend (MsgQAppl, 0, QErrPointer);
SendStr (&QAppl); /* Send password request to transceiver Port_IO. */
ReceiveStr (&Str); /* Receive a String from the transceiver Port IO. */
```

```
ApplStr = SmartOSDecrypt (Str, DES); /* Using an RTOS function, decrypt the input string for application
command and user data from the host */
32. /* Code for using the user data and executing received application command. */
.
.
33. /* Using an RTOS function, encrypt request closing request and post it in message queue. The closing
request is from a message queue MsgQApplClose. Then retrieve it from queue in QApplClose after
encryption. */
.
.
&QApplClose = SmartOSEncrypt (requestApplClose, DES);
Send (*QApplClose);
ReceiveStr (&Str);
ApplStr = SmartOSDecrypt (Str, DES); /* Using an RTOS function, decrypt the input string for password
from the host. */
/* Compare deciphered string with message "Closure Permitted". If found equal, then closure permitted,
then delete this task and other low priority tasks. */
If (strcmp (ApplStr, "Closure Permitted") {
SmartOS_task_DelEn = 1 /* Enable task deletion by SmartOS. */
OStaskDel (task_ReadPortPriority); OStaskDel (task_PWPriority); OStaskDel (task_ApplPriority);};
} /* End of While loop */
/* End of task_ReadPorts Codes. */
/******************************************************************************/
40. static void task_PW (void *taskPointer) {
while (1) {
41. /* Wait for IPC from task_ReadPort. */
SmartOSSemPend (SemPW, 0, SemErrPointer);
42. /* Code for retrieving one string from the protected file structure. It is used for requesting the
password from the user at the host of the card. */
.
.
43. /* Write the array elements after encryption. */
ApplStr = SmartOSEncrypt (requestPassword, DES);
SmartOSQPost (MsgQPW, ApplStr);
SmartOSSemPost(SemAppl) SmartSTimeDly(100);
SmartOSTimeDlyResume (task_ReadPortPriority); /* Resume Delayed task task_
ReadPort. */
44. } /* End of While loop */
45. /* End of task_PW Codes. */
/******************************************************************************/
46. static void task_Appl (void *taskPointer) {
while (1) {
SmartOSSemPend (SemAppl, 0, SemErrPointer); /* Wait for IPC from task_ReadPort. */
47. /* Code for retrieving one string for requesting the application commands requestAppl from the
protected file structure. It is for requesting the password from the user at the host of the card. */
```

```
.
.
48. /* Write the array elements after encryption. */
ApplStr = SmartOSEncrypt (requestAppl, DES);
SmartOSQPost (MsgQStart, ApplStr);
49. /* Resume Delayed task task_PW. */
SmartOSTimeDlyResume (task_PWPriority);
        } /* End of While loop */
50. }/* End of task_Appl Codes. */

/*****************************************************************/
```

12.5 CASE STUDY OF A MOBILE PHONE SOFTWARE FOR KEY INPUTS

Mobile phones are smart. Each phone has many APIs. Example of APIs are phone, SMS (short message service), MMS (multimedia messaging service), e-mail, address book, web browsing, calendar, task-to-do list, WordPad, Pocket-Word, Pocket-Excel, note-pad for memos, Pocket-PPTs, slide show andcamera.

Mobile phone with a large touchscreen uses a virtual keypad. Mobile phone with a small screen uses T9 keypad. The present case study relates to 'SMS create application' in a mobile phone with T9 keypad for inputs.

Section 12.5.1 gives the requirements of 'SMS create and send application'. Section 12.5.2 gives the classes and class diagrams. Section 12.5.3 gives the state diagram and Section 12.5.4 gives communication hardware. Section 12.5.5 describes software architecture. Section 12.5.6 describes the software tasks and Synchronization Model for the application.

12.5.1 Requirements

A processor, keypad, screen, scratch pad memory, persistence memory and communication units are required for SMS create and send application. Scratch pad memory addresses are used for temporary saving of characters (bytes) during the application. Persistence memory addresses are used used such that as soon a change is made in the byte, it persists even after the power switches off. Further, when there is a change there, an identical change is reflected in other correlated objects. For example, a name is edited in a file for the Contacts, the same change takes in the file for Address book for sending the e-mails.

Figure 12.20 shows specific units, which are used for the SMS text create application. The screen is used for displaying the menu. Figure 12.20 shows that there are four cursor keys (up, down, left and right) denoted by *C*1, *C*2, *C*3, and *C*4. In computer keyboard, four different cursor keys are used. The mobile cursor key in the keypad is such that it functions as four keys. When the key is pressed towards the left the cursor moves left (←), when it is pressed towards the right the cursor moves right (→), and so on for up (↑) or down (↓).

In eddition there are four command keys (right-corner second-row, left-corner second-row, right-corner first-row and left-corner first-row) denoted by key2Row2, key1Row2, key2Row1 and key1Row1. Also, there are nine T9 keys for numbers 1 to 9 as well as for alphabets a to z (or A to Z). There are two mode-keys (keyM1 and keyM2) and one key0 key for keying in a text character number 0 or space. Alphanumeric text in small case or capital case is controlled by a mode-key's state. Text character entered on keying depends on state of the T9 key. [Recall Examples 3.6 and 6.8 and Section 6.3].

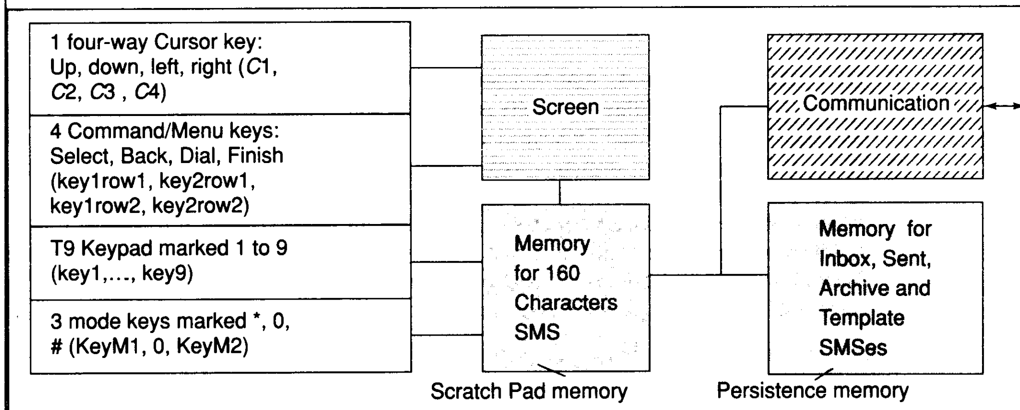


Fig. 12.20 Mobile phone Keypad, screen, memory and communication units for SMSes

Requirements of SMS create and send application module of SMS creation system is tabulated in Table 12.6.

Table 12.6 Requirements of 'SMS create and send application' module in a SMS creation system

Requirement	*Description*
Purpose	To create an SMS message and communicate using a T9 format keypad and using the tasks for *inserting a desired mobile number into a list, editing a message* and then *sending the message*.
Functions of the system	1. An option is selected by using cursor key for moving the cursor displayed on screen and then pressing command key Key1Row1.When a set of options consisting of 4 notifications, one for a command *Messages*, next for command-option *Text Messages*, next for application *Create Message* and last for *Text*, are selected generates an event E_SMS, [Section 8.4] that signals a task for creation of text messages in alphanumeric format. The characters are entered and edited with T9 keypad of nine keys from numbers 1 to 9, one four-way cursor keys, four command keys, two mode keys marked * and #, and one key marked 0. 2. When any key is clicked by the user, its state is computed based upon the key's earlier state, cursor's present position, timer-status and count, and then a notification E_NewState is posted into a message box for an ISR, task or application that initiates the required action. 3. For choosing the application SMS creating text, the cursor and command-cum-options-select (Key1Row1) keys are used as follows: (a) When command key (Key1Row1) is used to select command Messages, a command-option is then selected using a GUI. (b) User selects *Text Messages* option by clicking for the displayed option '*Text messages*'. (c) A menu then shows up for selecting one of the following using the cursor down and up keys: *Create message*, *Inbox*, *Sent items*, *Archive*, *Templates*, *My folders*, *Distribution lists*, *Delete messages* and *Message settings*. User selects 'Create message' for selecting SMS creation application. (d) A menu then shows up for selecting type of message to be created. Cursor can select one of the following menu items on display: *Text* and *Numeric page*. User selects Text for creating SMS text. When option Text is selected then event E_SMS posts (signals) to start execution of 'SMS create and send application' when the user finishes selection of four options *Messages*, *Text Messages*, *Create Message* and *Text* in sequence.

(*Contd*)

Requirement	Description
	4. On E_SMS event the tasks *Add Number*, *Edit Message* and *Send Message* execute in order to specify (i) a new or additional mobile number for transmitting the SMS, (ii) editing the SMS-message by keying in from T9 keypad and (iii) transmitting it over the mobile, respectively. 5. When mobile is inactive, display screen startup display shows up on key2row2 interrupt, which causes an ISR_key2row2 routine execution and state of mobile becomes wake up state *S_Wake*. When mobile is in active state S_Active and is running an application, it is brought to idle state S_Idle by at another key2row2 interrupt. Display screen switches off (shuts down) after a preset interval, say, 15 s and the state of mobile becomes sleeping state S_Sleep (only on incoming call there will be port interrupt, and mobile wakes to state S_Wake and screen shows '*start-up display*'.
GUIs	A GUI uses display of screen menu or text and cursor, command-select and cursor-position-change keys. The cursor position can be changed up, down, left and right by using *C*1, *C*2, *C*3 and *C*4. The cursor when it points on the screen to a line of menu for a command, it shows that menu item with blue or other background. When the cursor points to a character in a line of text or phone number, it shows a vertical line at the right of the character position. At the cursor position, a text can be selected or entered by a click of command-select key (Key1Row1). At the cursor position, the shown character can be cleared by a click of clear or back key (Key2Row1).
Inputs	1. State of a T9 key (key1,…., key9) . 2. State of mode key (keyM1 and keyM2) pair-marked * and # and that defines the functioning mode. 3. State of key0 (0, 0), (1, space), (1,0), (1, new-line) or (1, space), when editing an SMS. [State of key0 (0, 0) or (1, 0) when dialling a number.] 4. State for command from one of the four command keys (key1row1, key2row1, key1row2 and key2row2). 5. State of cursor-input on the cursor clicks on move up, down, left or right using *C*1key, *C*2key, *C*3key or *C*4key.
Signals, events and notifications	1. The E_NewState and message for *state* of key are posted on the interrupts from command key or any other key. 2. Event E_SMS, which starts SMS_Create_Text application, happens on posting of a set of four GUI notifications - *MsgMessages*, *MsgTextMessages*, *MsgCreate* and *MsgText*. [*MsgMessages* notification on selecting option Messages, *MsgTextMessages* on selecting option Text_messages, *MsgCreate* on selecting option Create_Message, and *MsgText* message on selecting option Text.] 3. Notifications for display on completion of tasks. For example, after completing the sending of SMS, display-notification 'Message sent' and before completing the transmission of SMS 'Sending message'. 4. A software timer's time-out interrupt ISR_T_Deacitvate switches off the phone at display of idle state (start up menu display in idle state) to the display off and sleep state. [Interrupt is after there is no action for a period longer than a programmed period = 15s.] 5. Another software timer's time-out interrupt ISR_T_Out_Help_Option when a cursor or marked menu is displayed then a pop-up help displays after a period longer than the programmed period 2s.
Outputs	1. SMS_Create_Text string, which is displayed on the screen and is also saved in scratchpad memory during the editing and also saved in sent folder after sending the SMS. 2. Screen menu text lines for displaying option(s), text of menu, marked text or character to enable its selection by clicks. 3. Help menu text display to display action, which will take place on selecting an option after a T_Out_Help_Option interrupt. Option is assumed as one at which cursor points.

(*Contd*)

Requirement	Description
Scratchpad Memory	The memory is used as scratchpad memory for 160 characters maximum in an SMS.
Persistence memory	The memory is allotted in the system at persistence memory addresses (in flash memory) for SMSes in Inbox, Sent, Archive and Template.
Design metrics	1. *Power Dissipation*: Battery operation 2. *Performance*: 3 minute for 1 SMS message creation and send 3. *Engineering Cost*: US$ 20000 (assumed) for software 4. *Manufacturing Cost*: None once the codes are ready and tested
Test and validation conditions	1. All commands and options functioning are tested.

Functioning can be explained in detail as follows:

Command: For *Messages* command, a key (for example) left-hand first-row command key (Key1Row1) is used and user selects the command Messages.

Options: On selection by clicking Key1Row1 when cursor is at a displayed command *Messages*, the command-options display. These options are used for selecting a command-option using the cursor at one of the following command-options: *Text messages*, *Voice Message* and *Minibrowser Messages*.

On selection by clicking Key1Row1when cursor is at the displayed command-option *Text messages* Text messages option selects. Then one of the following application-options are displayed: : *Create message*, *Inbox*, *Sent items*, *Archive*, *Templates*, *My folders*, *Distribution lists*, *Delete messages* and *Message settings*.

SMS Create Application: For selecting SMS create application option, Create message application-option is selected using cursor and click. Then one of the following two type messages can be created, *Text* and *Numeric page* as follows.

Message types: On selection by clicking at the displayed application option *Create message*, the type-options are displayed. These are used for selecting a message type-option using the cursor at one of the following command-options: *Text*, and *Numeric page*. Text is selected as the type-option.

Tasks: On selection by clicking at the displayed type option Text, the task-options are displayed. These are used for selecting task-option using cursor at one of the following task-options: *Add number*, *Add e-mail*, *Add list*, *Edit message*, *List recipient* and *Send*. Add number is selected as the task-option. On return from the task *Add number*, a new task option Edit message is selected. On return from the task *Edit message*, next task option Send is selected. If message is sent to two numbers, then before the Send, a number is added using task Add number. [A state diagram will be given in Section 12.5.3 to explain the concurrent processing of the tasks *Add number*, *Add e-mail*, *Add list*, *Edit message*, *List recipient* and *Send*.]

Editing and creating the alphanumeric text SMS: To enable the alphanumeric character entries using just 9 keys, the state of key is changed and notification is issued for the new state on a transition, which causes a new input. A state-transition causing input is applied on when there is a repeat pressing of either the same T9 key again or another key within a prefixed time interval.

This can be explained as follows: Example 3.6 showed how a key marked 5 (5 in first line in large font and jkl in small font in second line marking on key5 surface) on pressing can produce the different states after transition from idle state (0, 5) in sequences represented by (1, 5), (1, j), (1, k), (1, l), (1, 5), (1, j), (1, k), (1, l), (1, 5), till user does not repeatedly press key5 within an interval Δt or presses another key in state (0, n) where n is for any other T9 key.

First character inside bracket in a key-state representation shows whether key is inactive (0) or active (1). There is transition to 1 when a key is pressed and to 0 when it's key-state is accepted as the input undergoes transition to idle state. A state is accepted as input if within a time interval Δt either the user presses another key or there is no repeat press of the same key. The second character after comma in a state representation shows the number or text that will be sent in ASCII code format if the active state remains unchanged in a preset time interval Δt and mode M2 key has not modified the mode to text in another language.

Other keys also have similar characteristics. The transition of a key state occurs if it is pressed again within an interval. Let us recapitulate Section 6.3, which described a model of state machine. Example 6.8 gave the state table and Example 6.9 gave C codes for the state table.

12.5.2 Class Diagram and Classes

An SMS application program uses the principle of Orchestrator software (Figure 12.2). The inputs from the keys for commands, GUIs and data inputs are taken as equivalent to sensor inputs. A notification or signal issued after each input is taken as equivalent to actuator output. Let us therefore define an Orchestrator class for the application.

Display screen is used during every input and every output in the mobile device. There is a start up display screen. A typical example is screen display, which shows *time* on right hand corner, service company name in the centre, *date* in next line, left side bar for *antenna signal strength*, right side bar for *battery power* and also shows a *start up display graphic*. The *options* are according to provisioning by the service provider and an image for wall-display selected by user. Let us design a Task_ScreenDispl class. It is interfaced to the graphic method. The graphic is chosen from the options to user such as Beach, Car, Coral, Daisy, Dance, Disco, Dragon and others. Task_ScreenDispl extends the other classes, the objects (instances) of which are used in menu item display, GUIs or displaying current action or help text and other display.

SMS creation and send application has a number of concurrent processing tasks. Let us define Task_SMS_CreateTextSend.

Assume that SMS creation application consists of three class diagrams for Orchestrator and two for Task_SMS_CreateTextSend and Task_ScreenDispl. Figure 12.21 shows class diagrams of ORCHESTRATOR, Task_ScreenDispl and Task_SMS_CreateTextSend for creating and communicating SMS message.

1. ORCHESTRATOR class extends to Orchestrator_CommandsGUIs and to Orchestrator_SMSCreateSend.
2. Task_Messages, Task_TextMessages, Task_CreateMessage and Task_Text and interface keypad interrupt ISR_KINT.
3. Task_SMS_CreateTextSend extends to Task_AddNumber, Task_AddEmail, Task_AddList, Task_EditMessage, Task_ListRecipient and Task_ Send.
4. Four types of screen displays are used during SMS create and send application. Start up screen display, menu items display, SMS display during editing task and action display during sending the SMS. Task_ScreenDispl thus extends to four classes Task_StartUpDispl, Task_ SMSDispl, Task_ActionDispl and Task_ MenuTextLinesDispl.
5. The ISRs are ISR_WirelessPort, ISR_T_Out_Help_Option, ISR_T_Deacitvate and ISR_KINT.
6. ISR_KINT runs the service functions for any of the state transitions of twenty key and senates notifications for the state of a key Key1Row1 or Key1Row2 or , Key2Row1, Key2Row2, *C1* to *C4*, M1 or M2 or keys 0 to 9.

Class Figure 12.22 shows Task_ MenuTextLinesDispl. It has pixels field of Unsigned byte [ ]. A string is an array of characters. StrLine1, StrLine2, StrLine3 and StrLine4 are the strings in the object of MenuItems. The colour fields are textLineColor, cursorTextLineColor, screenBackgroundColor. The cursor has two fields line and a coloured bar. The methods are OSMBoxAccept ( ); OSMBoxPend( ); OSMBoxPost( ) and mouseClick ( ).

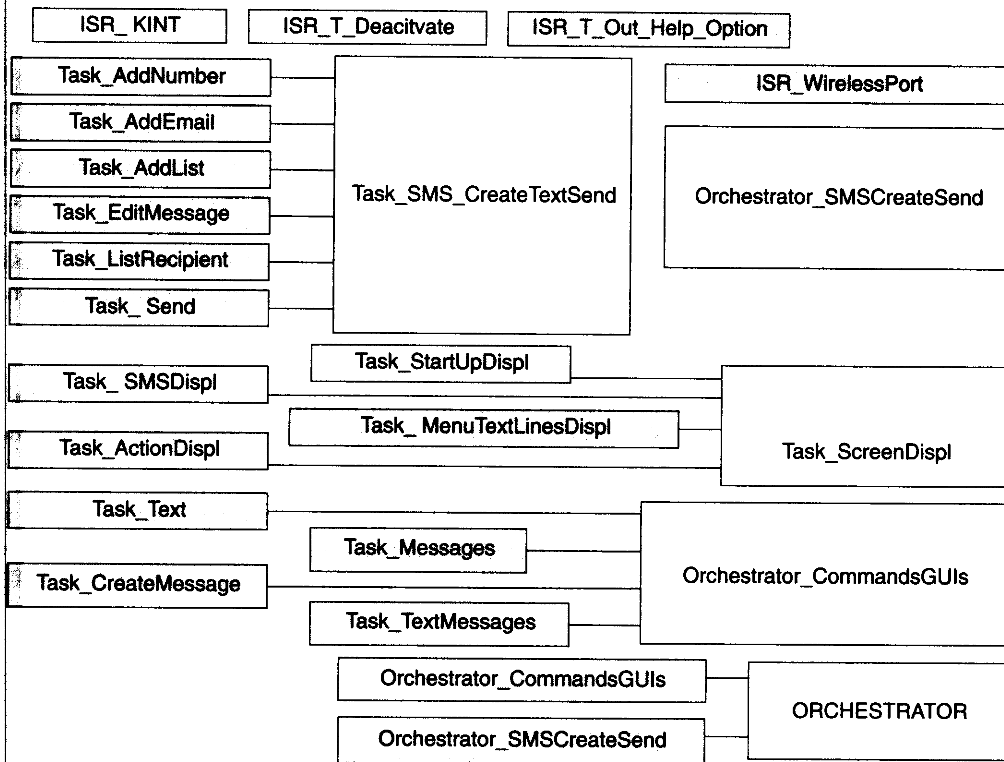


Fig. 12.21 ORCHESTRATOR, Task_SMS_CreateTextSend and Task_ScreenDispl *class diagrams of SMS create and send application*

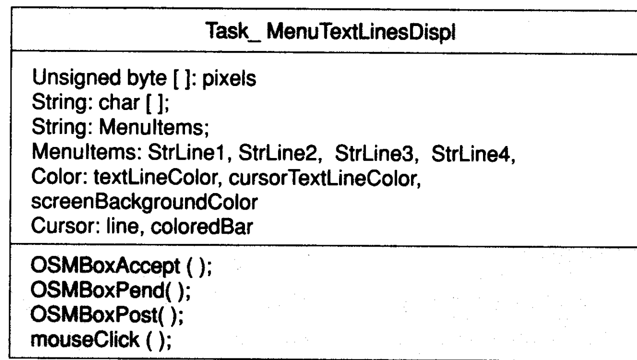


Fig. 12.22 *Class* Task_ MenuTextLinesDispl

Objects Message queue objects are used for posting to the ISRs and tasks. Message queue objects are accepted or waited for at the tasks. For example, *MsgMessages*, *MsgTextMessages*, *MsgCreate* and *MsgText* are posted on Key1Row1 interrupts. Event objects are posted on a set of notifications. For example, E_SMS is posted on *MsgMessages*, *MsgTextMessages*, *MsgCreate* and *MsgText.* E_NewState is posted on any new state generation on interrupt from any key in the mobile. [Figure 12.20]

12.5.3 State Diagram

Figure 12.23 shows a state diagram for task_SMS_CreateTextSend. A state diagram shows a model of a structure for its start, end, in-between associations through the transitions and shows events-labels (or conditions) with associated transitions. A dark rectangular mark within a circle shows the end. The state transitions take place between the tasks, task_SMS_CreateTextSend and task_AddNumber, task_AddEmail, task_AddList, task_EditMessage, task_ListRecipient and task_ Send. A state transition occurs after notification of MsgAddNumber, MsgAddEmail, MsgAddList, MsgEditMessage, MsgListRecipient and MsgSend on selection of menuItems Add Number, Add Email, Add List, Edit Message, List Recipient and Send, respectively. Task_Send posts the event to initiate ISR_WirelessPorts through Orchestrator to send SMS and end the application.

12.5.4 SMS Keying Hardware

Hardware architecture specifies the appropriate decomposition of hardware into processors, ASIPs, keys, memory, ports and devices It also specifies interfacing and mapping of these components. Specifications for SMS keying-in hardware are as follows:

Cursor key One four-way *cursor key*, which is pressed to move the cursor up, down or left or right of character when editing the SMS when it is being created. The actions are similar to ↑, ↓, ← and → keys in a keyboard. On cursor-key interrupt on click, the notifications are sent for the states *C*1key, *C*2key, *C*3key and *C*4key and current cursor display-position.

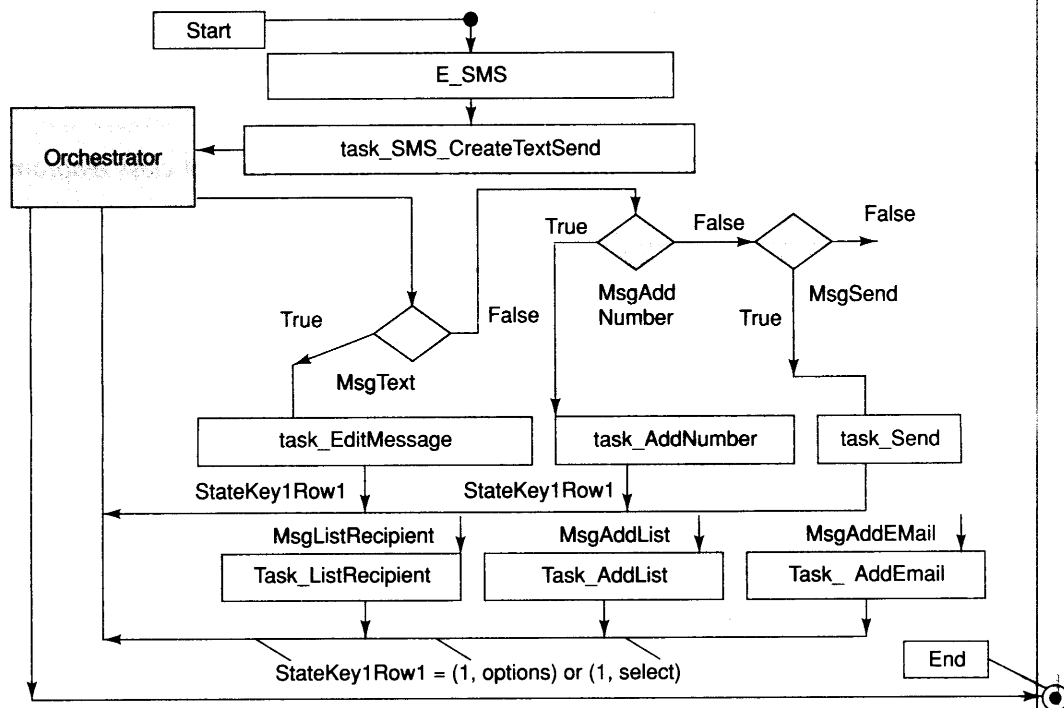


Fig. 12.23 *State diagram* for task_SMS_CreateTextSend

Command keys Four number *command keys*, key1row1, key2row2, key1row2 and key2row2 are present. Use of Key1Row1 and Key2Row1 is similar to the left and right clicks in a computer-mouse for GUIs. Use of

Key1Row2 and Key2Row2 clicks is similar to click to *start* menu item and *turn-off* or restart menu item, respectively, in a computer start up window.

On clicking a command-key, command-key interrupt routine sends notifications for the commands and their options. The keys are used as follows:

1. Right corner-second row command key (for key2row2 interrupt) is marked red with phone head down sign. It is used to activate the idle device and show start-up display. The same key is also used to switch off an active operation (including a phone call at any instant).
2. Left-corner-second row key (for key1row2 interrupt) is marked green with phone head lifted sign. It is used to dial the number, which is in view on screen or selected using cursor from viewed number (s) in the screen-menu. The same key is also used to receive an incoming call, after a ring tone is played and the number flashes on the screen.
3. Left-corner-first row key (for key1row1 interrupt) is marked green with dash sign and is used to activate all the available options, menus and submenus for starting an application. This key action is similar to left click mice button when the cursor is at a Window for selecting a command from the set of options, buttons and menus.
4. Right-corner-first row key (for key2row1 interrupt) is marked green with dash sign and is used to activate *to select* the menu commands, and is additionally used or *to clear* the keyed character during editing process or *to go back* to previous menu options. For example, the key is used for *Contacts*, *Calculator*, *Create message* and *Game* or the programmed options for their start. It opens for selecting a command from a limited set of displayed menu options. This key action is similar to action on right click of computer-mouse button when cursor is over a button or at a screen position.

T9 Keys T9 keypad is used for keying in of SMS. T9 keypad has nine keys 1 to 9 plus a key0. T9 key (key1,..., key9 key) inputs are used for dialling numbers as well as editing the text inputs during SMS create application.

Mode Keys Two keys marked * and # key modify the states of keyM1 and keyM2 when they are pressed. Mode Key use can be understood by the following example. For example, to protect accidental use of the keys when phone is kept in the pocket, the key1row1 and M1 are simultaneously pressed, the pad undergoes transition to lock state if previously it was in the unlock state and to unlock state if it was in lock state. Another example is bilingual or multilingual text SMS editing. M2 is used to convert the English text mode to other language text mode.

Display Screen Screen displays the GUIs for start up display, menus, options, text being edited or actions currently taking place.

12.5.5 SMS Create and Send Application Software Architecture

Software architecture specifies the appropriate decomposition of software into modules, components, appropriate protection strategies and mapping of software. Software architecture consists of the following in the SMS create and send application.

1. OS. [OS controls the OSMsgQAccept, OSMsgQPost and OSMsgQPend message queue functions at message boxes and event functions at the even boxes. OS synchronizes the tasks and facilitates concurrent processing of SMS application on the mobile].
2. Key-system layer using Orchestrator and ISR_KINTs (interrupt service routines on interrupts from the keys)
3. Application layer

Application layer has
1. Tasks as shown in Figure 12.21
2. ISRs for initiating action on user inputs and GUI notifications

(1) ***Key-system layer***: A layer in software architecture is used for the key-system. Figure 12.24 shows a key-system layer. A key click generates an interrupt and a service routine ISR_KINT, which then executes an Orchestrator. The ISR_KINT reads the port status bits to find which key has been clicked, and also to read the timer status and timer counts and cursor position and that position menu or text message. It signals the Orchestrator to initiate and generates the notifications and events and posts these into the message boxes and event boxes for waiting tasks. For example, ISR_KINT initiated Orchestrator posts the following messages:
1. *MsgMessages* when cursor on display screen points to a Command_Msg Messages.
2. *MsgTextMessages* when cursor on display screen points to a command_option_Msg TextMessages.
3. MsgCreate when cursor on display screen points to an application-option_Msg point to Create, [Screen displays the application options *Create message*, *Inbox*, *Sent items*, *Archive*, *Templates*, *My folders*, *Distribution lists*, *Delete messages* and *Message settings* options].

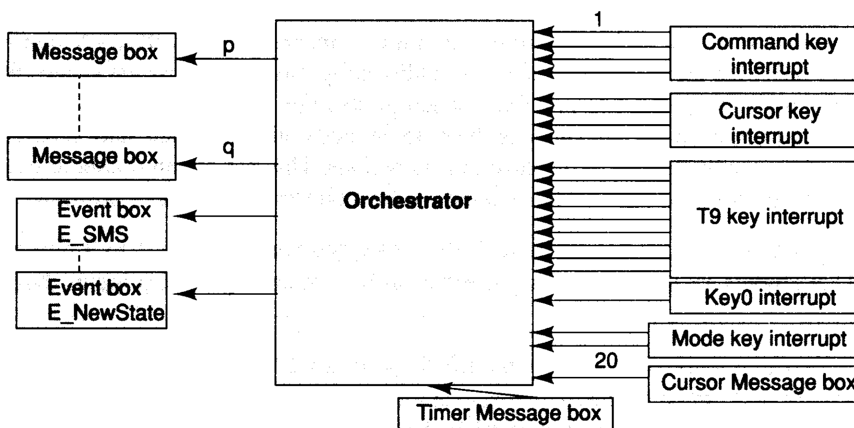


Fig. 12.24 Key-system layer with ISR_KINT and Orchestrator in software architecture of mobile

4. MsgTextType when cursor on display screen points to a type_option_Msg Text.
5. S_Key0, S_Key1, S_Key2, …., or S_Key9 state as per the timer status and counts if there is new state of key0, key1, key2, …, key9.
6. S_C1, S_C2, S_C3, or S_C4 state if *C1* or *C2* or *C3* or *C4* is clicked.
7. S_M1 or S_M2 if M1 or M2 is clicked.
8. S_key1Row1, S_key1Row2, S_key1Row2 or S_key2Row2 S_key1Row1if a command key is pressed is clicked.
9. E_SMS if command, command option, application option and type options *MsgMessages*, *MsgTextmessages*, *MsgCreate* and *MsgTextType* were posted in steps 1, 2, 3 and 4. [*MsgMessages*, *MsgTextMessages*, *MsgCreate* and *MsgTextType* initiates event object E_SMS.]
10. E_NewState if any state change step 5 or 6 or 7 or 8.

(2) ***Application layer*** is a layer in software architecture.
1. An application task is object of class Task_ MenuTextLinesDispl. It executes on posting of a message-object MsgTextMessages or MsgMessages or MsgTextMessages, MsgCreate or MsgTextType into a message box by an ISR_KINT (Fig. 12.24) on E_NewState.
2. Another application task is Task_SMS_CreateTextSend for SMS create and send application and it executes on event E_SMS. State diagram of it was shown in Figure 12.23 Section 12.5.3.

12.5.6 Software Tasks and Synchronization Model

Figure 12.23 showed the state diagram of synchronizing cycle of different tasks. The SMS communication system has a cycle of actions and tasks synchronizing model. Orchestrator posts notifications to the task_Messages, task_TextMessages, task_CreateMessage and task_Text.

1. A cycle starts in Orchestrator. A task, task_Messages, which receives a notification by message S_Key1Row1 choice of command. It posts MsgMessages.
2. A task task_TextMessages is for command option. It posts message *MsgTextMessages* on user selection. Another task task_CreateMessage accepts *MsgTextMessages* and posts *MsgCreate* on user selection. Another task task_Text accepts *MsgCreate* and posts *MsgText* on user selection.
3. Orchestrator then posts E_SMS. On receiving E_SMS the Task_SMS_CreateTextSend signals the displaying of menu items for initiating task_AddNumber, task_EditMessage and task_Send, task_AddEmail and task_ListRecipient.
4. Task task_AddNumber adds the message mobile number for sending the SMS in a list. It posts message *MsgAddNumber* on user selection.
5. On user selection, Orchestrator posts message, which signals another task task_EditMessage to start. It is used for creating and editing the message by keying-in the characters. Orchestrator accepts state of key on each key click and posts state in the message box of the key clicked during the editing. task_EditMessage accepts state message of the keys and creates the SMS_Create_Text string.
6. On user selection, Orchestrator posts message, which signals another task task_Send.
7. Task_Send posts MsgSend. Orchestrator accepts MsgSend and posts MsgCommuncation for Communication Port Interface. It accepts MsgCommuncation and posts SMS_Create_Text string through wireless.

Figure 12.25 shows a Synchronization model for SMS create and application tasks, ISRs and Orchestrator tasks (task_Addlist, Task_ListRecipient and task_AddEmail synchronization not shown) using the message and event boxes.

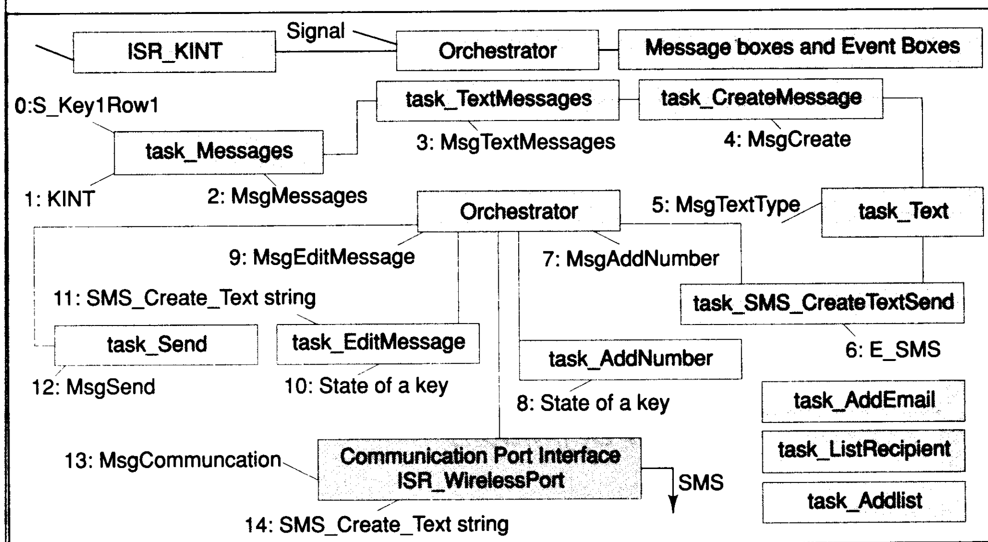
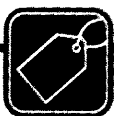


Fig. 12.25 Synchronization model for SMS create and application tasks, ISRs and Orchestrator tasks (task_Addlist, Task_ListRecipient and task_AddEmail synchronization not shown)

Summary

The following is a summary of this chapter

- Four case studies are explained: system design for the robot orchestra, automatic cruise control system, smart card and mobile phone SMS create and send application. The system design by software engineering and UML modeling approaches are explained in these case studies.
- Class diagrams, classes, state diagram, synchronization model, hardware architecture and software architecture are described using examples.
- Robot orchestra example explains the synchronization of MIDI messages between conductor and players.
- Orchestrator is software which sequences, synchronizes the inputs from 1st to k^{th} sensors or other sources and generates messages, notifications, signals and outputs for the actuators, display and message boxes at specified instances and time intervals after an input change. Message boxes store the notifications, that initiate the tasks. Application of Orchestrator is explained by taking examples of the robot orchestra and mobile phone SMS create and send application.
- There are many automobile electronics applications of embedded systems and the important ones were summarized here. A feature in modern cars, automobile cruise control system, was described. The adaptive control system is defined. Due to the greater need of reliability from the point of view of human safety, the need for special features in OS for automobiles was explained. MISRA-C standard for C language software defines the guidelines for automotive systems for using C. MISRA-C version 2 (2004) specifies 141 rules for coding and gave a new structure for C. A standard RTOS is OSEK-OS for automotive electronics. *Automotive cruise control* system code design is given using the VxWorks after retaining and incorporating the OSEK-OS features with it during the exemplary coding.
- Embedded system design for card-host communication in a smart card is given to explain special embedded hardware and special RTOS functions needs. The case study showed that for developing codes for an embedded system, the RTOS MUCOS or VxWorks functions might not suffice. A hypothetical RTOS, SmartOS, which has MUCOS features plus the embedded system special OS functions when it requires cryptographic features and file security, access conditions and restricted access permissions. After an initialization task that executes on system booting, three tasks are scheduled by the SmartOS. (a) A task reads the application strings from the card data files. It sends, after encryption, the messages to UART. It receives the encrypted strings from the UART of the host. This example also shows how multiple functions can be handled by the same task to reduce the memory needs. It is a desired feature in the smart card case. There is only 8 kB in most cases and 64 kB in extreme cases. The task for the password as well applications interacts through the IPCs. In the end, after seeking host authorization, the tasks are deleted. (b) A task sends the password request from the user interacting with the host. (c) One task gets the commands for executing the desired application routines and user data from the host.
- Embedded system design for SMS message creation and communication is described. The example explained the uses of command keys, cursor keys and GUIs in mobile phone SMS create and send applications. It is shown that the concept of Orchestrator, used in robot applications, is also useful for mobile phone applications.



Keywords and their Definitions

Adaptive Algorithm : An algorithm that adjusts and adapts to the parameters and limits the changing perturbations in a control system.

Adaptive Control : A control system that uses an adaptive algorithm to generate output control signals.

Adaptive Cruise Control : An automobile throttle control system to maintain constant preset cruising speed.

Bluetooth : A protocol used in mobile devices with features of service discovery, self-organizing network, self-establishing and self-configuring network, emulation of COM port of PC and other features and is used for data synchronization and communication between the devices within a piconet or scatternet.

Charge Pump : A combination of diode and capacitor to extract and store charge for providing a supply to the system using an appropriate voltage regulator circuit.

COM port : A port in PC, which is used for connecting to mice or modem and which has 9 or 25 pin RS232C standard serial or parallel connector and which uses data and handshaking signals standard specified by UART. Emulation means sending the RxD (received data) and TxD (transmitted) data in same format by another protocol as in 11-bit UART (one start bit, 8 data bits, parity or programmable bit and stop bit).(Section 3.2)

Command key : A key used to select menu item options to start a new task or action.

Cryptographic Software : Software for encrypting and deciphering a message or a set of byte streams. It uses an algorithm for encrypting and another algorithm for decrypting.

Cursor : A line or symbol or icon displayed on the screen to guide a user to select the button or text shown at that position.

Cursor key : A key, when pressed towards left moves a cursor to the left, right moves the cursor to the right, and so also up and down.

Data Acquisition System : A system to acquire data from multiple ports and channels.

Event : An instance of presence of a notification or message or set of messages or action(s) that initiates an ISR. There is reset of the notification on start of ISR to enable response to next event.

Fabrication Key : A key embedded in ROM at the time of card fabrication so that the card gets an unique identity.

GPS (Global positioning system) : A system for determining *locations*, *speed*, *direction* and *time* by a receiver. A set of 24 or more medium earth orbit satellites beams the signals to enable a GPS receiver. The receiver is positioned at any place on globe to receive signals for determining these four parameters.

Invalidation Lock : A lock, which, if placed in the application data files in the card, makes the card invalid for further use.

Java Card : A Java language format for smart card applications.

JVM (Java Virtual Machine) : The supervisory codes that execute the Java classes compiled as byte codes by the Java compiler. The codes run with the help of JVM in a computer system.

Logical Address : A memory address used for an instruction or data byte of the RTOS or application.

Message box : A mailbox in which notification(s) are placed by a task and another task accepts the message or waits for the message.

MIDI (musical instruments digital interface) : A protocol to define the specification required for hardware interfaces, message formats and for sending the program change, system and channel messages. The channel messages define the musical *note*, *pitch-bend*, *control change*, *program change* and *after-touch* (poly-pressure) messages.

Notification : A message generated on *listening* to clicking a key or on a change of *state* of a key or clicking of a button or selection of a menu item. Notification is for another task.

Orchestra : A musical event played using several musical instruments, each with a player and conducted by a conductor.

Orchestrator : Software, which sequences and synchronizes the inputs from the 1st to the kth source and generates the messages and outputs for the actuators, display and message boxes at the specified instances and time intervals. Message boxes store the notifications that initiate the events and tasks as per the notifications.

Personalisation Key : A key placed after testing the smart card circuit. The card is personalized for its own protected area of memory and own translation scheme for conversion between physical and logical addresses during actual running of the tasks at the card. After insertion of this key, the RTOS and application use only the logical addresses, and the processor uses this key during the translation between two addresses.

Physical address : The address used by the processor to fetch the data or send the data to memory and ports.

Piconet : A Bluetooth network with the devices within a distance of about 10 m.

PIN : Personal Identification number. Bank or hosting service allocates this PIN. The allocation unit has its own PIN, called host PIN.

Protection Bit : A bit at the ROM that the processor uses for not letting the transfer of instructions and data in the protected part to the system external buses. The processor externally blocks the write cycles for accessing these protected addresses.

Qrio : An invention of Sony meaning Quest for curiosity, which gave a smoother and faster humanoid robot than ever before in 2003, weighed 7.3 kg and was 58 cm, and had one hour battery. A set of the Qrios also played the orchestra and dance numbers.

Radar (Radio Detection and Ranging) : A system that uses radio waves of below 1 m to enable ranging of short distant objects by measuring time delay between trasmitted signal and reflected signal.

RSA : An algorithm that uses the prime numbers. RSA stands for the first letters of the last names of its three inventors; Ron Rivest, Adi Shamir, Leonard Adleman.

Scatternet : A network within 100 m between various piconets connected through a Bluetooth-enabled bridging device and formed by self discovery and self organizing features of the Bluetooth protocol.

SHA : A Security Hash Algorithm based on a hashing function.

State : A state of parameters that results after an input and that depends upon the previous sequences of states which occurred since starting from initial state. For example, a counter has a different state after a count input and its state depends on the previous number of inputs. State can also depend on time if time-out interrupt is used as state-transition input. A key has on-off states or if clicked one, two, three times then multiple states. For example the states of the T9 keys.

String Stability of Vehicles : Stability by maintaining constant distance between multiple streaming vehicles in a convoy on highway or VIP duty.

T9 Keypad : A keypad that includes nine T9 keys for nine numbers 1 to 9 as well as for alphabets a to z (or A to Z). Entered alphanumeric text in small case or capital case is per the as per a mode-key state. Text character depends on state of the T9 key.

Throttle Valve : A valve to control the engine thrust and hence acceleration.

Unblocking PIN : A host PIN used for unblocking a certain part of the card memory for using. For example, for permitting a modification of the user password after unblocking (permitting access by modifying the access condition fields) password file in the memory.

Review Questions

1. How are the MIDI messages used for conducting an orchestra? Give specific examples.
2. List the tasks required for MIDI file communication using Bluetooth piconet between the players and conductor robots. Explain task priority assignments in robot orchestra MIDI messages communication.
3. List the embedded devices in a high-end car.
4. What is adaptive control? How does adaptive control algorithm difffer from feedback proportional control.
5. List the tasks and ISRs required for ACC system. Explain the actions of each. Explain task priority assignments in an ACC system.
6. What should be features in OS for automobile applications? Why should a MISRA-C version of C be used in ACC tasks?
7. What are the Bluetooth device piconet and scatternet ranges? Discuss the advantages and disadvantages of Bluetooth based inter car communication in place or radar- or laser-based communication?
8. Why is Java popular for smart card applications?
9. How does a contactless smart card hardware derive power?
10. Why is the use of a processor with memory protection bit essential?
11. What is the advantage of encryption when using a fabrication key, personalization key, utilization liock and PIN?
12. Tabulate the features needed in the OS for a smart card.
13. Explain how the task of reading ports in smart card synchronizes with the port device driver.
14. List the tasks and ISRs required for a smart card system. Explain the actions of each. Explain task priority assignments in a smart system.
15. List the tasks and ISRs required for mobile phone SMS create and send application. Explain the actions of each. Explain task priority assignments in an SMS create and send application of mobile phone.
16. We can use a number of mailbox IPC messages from a task in a mobile phone. Explain how this has been effectively used. Why is the use of mailboxes option followed in place of message queue in a mobile phone SMS create and send application?

Practice Exercises

17. What are the list of tasks for dancing robots.
18. Give the list of tasks for the ACC with string stability among 4 cars. .
19. List classes for host of smart card used in the Bank ATM.
20. Draw Class diagrams of SMS Inbox messages read application.
21. Write the sequences and state diagrams of key pressing event in T9 keys. Assume that key5 has state sequences s on transitions from idle state (0, 5) (1, 5), (1, j), (1, k), (1, l), (1, 5), (1, j), (1, k), (1, l), (1, 5), till within an interval Δt user does not repeat pressing of key5 or presses another key in state (0, n) where n is any other T9 key. What will be the state sequences for key1, key2, key3, key4, key6, key7, key8 and key9.
22. List the classes that extend the Task_ScreenDispl in a mobile phone.
23. Give a synchronization model for editing and string creation using Task_EditMessage during creating an SMS message.
24. Design the codes for SMS text editing using Task_EditMessage.
25. List classes that will be used for start up display, display off after 15s and pop-ups a help-display if cursor stays at a menu item for more than 2s.

Embedded Software Development Process and Tools

13

R e c a l l

Recall chapter 1 afresh. We defined an embedded system as one that has software embedded into a computer hardware. Embedded system has three main components.

- Hardware
- Main application software. Application software perform multiple tasks
- RTOS

In the previous chapters, we have learnt all the three main components of embedded systems and have covered following topics.

- Embedded system hardware consisting of processor, memory, devices and basic hardware units – power supply, clock circuit and reset circuit.
- Devices consisting of I/O ports to access the peripheral and other on-chip or off-chip physical-devices. Physical-device examples are UART, modem, transceiver, timer-counter, keypad, keyboard, LED display, LCD display, DAC, ADC and pulse-dialer.

L E A R N I N G   O B J E C T I V E S

- **Real-time clock-driven software timers.**
- **Virtual devices (pipe, socket, file, etc).**
- **Device drivers and interrupt-handling mechanism in an embedded system.**
- **Need for power dissipation management by the processor instructions during high-speed computations.**
- **Selection of appropriate processor, memory and devices for optimum system performance.**
- **Interfacing of system buses with the memory and I/O devices, and use of DMA controller to improve system performance by enabling the I/O units to have direct access to system memory.**
- **High-level language programming concepts, program models and software-engineering approaches.**
- **RTOS, use of IPCs, exemplary uses of RTOS MUCOS (μC/OS-II) and VxWorks functions, and study of MUCOS, VxWorks, Windows CE, OSEK and Real Time Linux programming environments and their applications.**

Chapters between 5 and 12 focussed on software aspects. In this chapter, we focus on hardware and software integration aspect. We will learn the following for understanding embedded development process and tools.

1. *Software, source code engineering and integrated development environment (IDE) tools.*
2. *Software is developed on a host machine, say, a personal computer for a target system, which in most cases uses distinct processor and OS. There are two development platforms: host and target machines.*
3. *Linking and locator to create file for binary image for the final software.*
4. *Device programmer to burn the codes in the PROM or flash bugging of system monitor codes in ROM.*
5. *Issues in embedded system development that need to be addressed by any development team. These are independent software-hardware design, hardware-software co-design, choosing right processor, allocation of memory addresses, devices and bus and porting issues of OS/RTOS.*

Chapter 14 will describe testing and debugging.

13.1 INTRODUCTION TO EMBEDDED SOFTWARE DEVELOPMENT PROCESS AND TOOLS

13.1.1 Development Process and Hardware-Software

Figure 13.1(a) shows the development process of an embedded system and Figure 13.1(b) edit-test-debug cycle during implementation phase of the development process. There are cycles of editing-testing-debugging during the development phases. Whereas the processor part once chosen remains fixed, the application software codes have to be perfected by a number of runs and tests. Whereas the cost of the processor is quite small, the cost of developing a final targeted system is quite high and needs a larger time frame than the hardware circuit design.

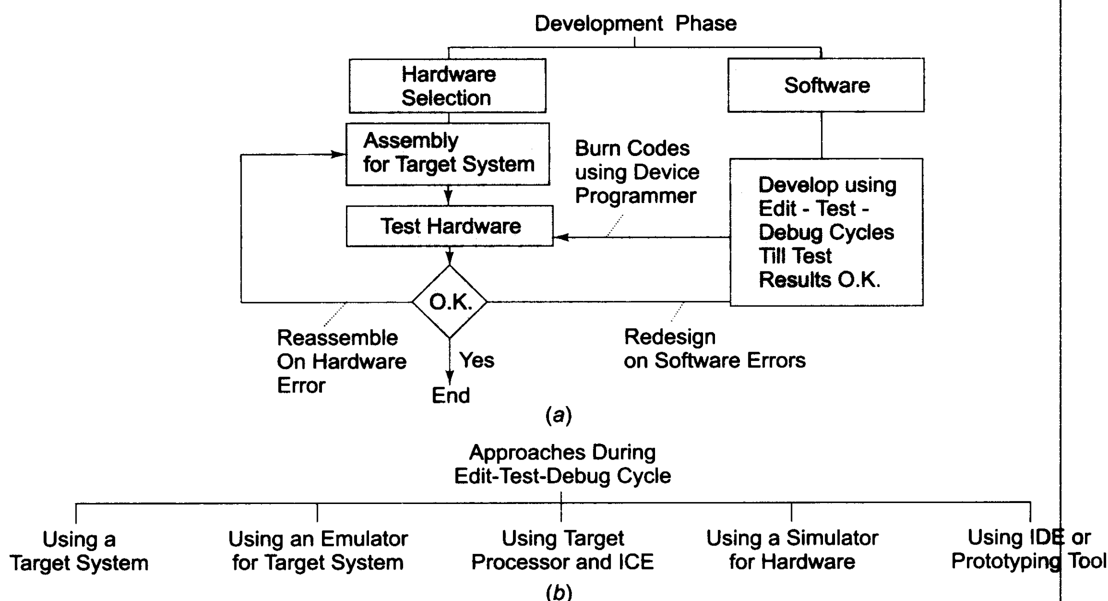


Fig. 13.1 (*a*) Development process of an embedded system (*b*) Edit-test-debug cycle during the implementation phase of the development process

The developer uses four main approaches to the edit-test-debug cycles.

1. An IDE or prototype tool (Refer to Section 13.1.4).
2. A simulator without any hardware (Refer to Section 14.2).
3. Processor only at the target system and uses an in-between ICE (in-circuit-emulator) (Refer to Section 14.3.6).
4. Target system at the last stage.

13.1.2 Software Tools

The tools are required for the application software high-level language programming. Also required are the RTOS, testing debugging, assembly language programming (for implementing the device-driver functions)

and system integration tools. Table 13.1 lists the software tools in software and hardware implementation for embedded system.

Table 13.1 Software Modules and Tools for implementation of an Embedded System

Software Tools	*Application*
Development kit	Development kit is used for editing, configuring (disabling and enabling the C++ features), GUIs development and compiling.
Source-code engineering software	Source code engineering tool is used for editing, configuring (e.g., disabling and enabling the C++ features), GUIs development and compiling as well as for source code comprehension, navigation and browsing, and debugging (Section 13.1.3).
RTOS	An operating system (OS) for multitasking, process, memory, IO, network, devices, file system and for real-time control of processes.
Integrated development environment	Software and hardware environment that consists of simulators, editors, compilers, assemblers, RTOS, debuggers, stethoscope, tracer, emulators, logic analysers, application codes' burners for the integrated development of a system (Section 13.1.1).
Prototyper	For simulating, source code engineering including compiling, debugging and navigating through the codes using a browser, summarizing complete status of final target system during the development phase. Tornado prototyper from WindRiver® for integrated cross-development environment with a set of tools (Section 14.2.4).
Compiler	For using the complete set of the codes, functions, expressions and library routines and creating a file called object file.
Assembler	For translating the assembly mnemonics into binary opcodes (instructions), that is, into an executable file, called binary file. It also creates a list file that can be printed. The list file has address, source code (assembly language mnemonic) and hexadecimal object codes. The file has addresses which are allocated again during the actual run of assembly language program.
Cross-assembler	For converting object codes or executable codes for a processor at development system to other codes for another processor for embedded system and vice versa.
Cross-compiler	For compiling source codes for a another processor and vice versa.
Testing and debugging tools	Simulator for simulating most functions of a target embedded system circuit including additional memory, peripherals and buses on the host system itself (Section 14.2.3); stethoscope for dynamically tracking the changes in any program variable; trace scope for tracing the changes in the modules and tasks as a function of time on the X-axis; memoscope for memory usage which is a critical aspect of an embedded system; ScopeProfile to find in which task the CPU spends how many of its cycles in order to understand performance bottlenecks; in-circuit emulator (Section 14.3.6); monitor (Section 14.3.7).
Locator	Uses cross-assembler output and a memory allocation map and provides the locator-program's output (Section 13.3).
Editor	For writing C codes or assembly mnemonics using the keyboard of host system (PC) for entering the program. Allows the entry, addition, deletion, insertion appending previously written lines or files, merging record and files at the specific positions. Creates a source file that stores the edited file. That has an appropriate name.
Interpreter	For expression-by-expression (line-by-line) translation to the machine executable codes.

Software tools are used to develop software for designing an embedded system. Sophisticated tools—Integrated development environment and prototype development tools—are needed for integrated development of system software and hardware. The testing and debugging tools are needed for testing and debugging.

13.1.3 Source Code Engineering Tool

A source code engineering tool is of great help for source code development, compiling and cross-compiling. The tools are commercially available for embedded C/C++ code engineering, testing and debugging.

The features of a typical tool are comprehension, navigation and browsing, editing, debugging, configuring (disabling and enabling the C++ features) and compiling. A tool for C and C++ is SNiFF+. It is from WindRiver® Systems. A version, SNiFF+ PRO has full SNiFF+ code as well as debug module. Main features of the tool are as follows:

1. It searches and lists the definitions, symbols, hierarchy of the classes and class inheritance trees. [The symbols include the class members. A tree is a data structure. A data structure tree has a root. From the roots, the branches emerge and from the branches more branches emerge. On the branches, finally there are the leaves (terminating nodes).]
2. It searches and lists the dependencies of symbols and defined symbols, variables, functions (methods) and other symbols.
3. It monitors, enables and disables the implementation virtual functions. Use of virtual functions is for dynamic run-time binding.
4. It finds the complete effect of any code change on the source code.
5. It searches and lists the dependencies and hierarchy of the included header files.
6. It navigates to and fro between the implementation and symbol declaration.
7. It navigates to and fro between the over-ridden and over-riding methods. (Overriding method is a method in a daughter class with the same name and number and types of arguments as in the parent class. Over-ridden method is the method of the parent class, which has been redefined at the daughter class.)
8. It browses through information regarding instantiation (object creation) of a class.
9. It browses through the encapsulation of variables among the members and browses through the public, private and protected visibility of the members.
10. It browses through object component relationships.
11. It automatically removes error-prone and unused tasks.
12. It provides easy and automated search and replacement.

The embedded software programmer for sophisticated applications uses a source code engineering tool for program coding, profiling, testing and debugging of embedded system software.

13.1.4 Integrated Development Environment (IDE)

IDE consists of simulators with editors, compilers, assemblers, etc., emulators, logic analysers and EPROM/EEPROM application codes burner. An IDE must have the following features.

1. It has a facility for defining a processor family as well as defining its version. It has source code engineering tools (Section 13.1.3) which incorporate the editor, compiler for C, embedded C++, assembler, linker, locator, logic analyser, stethoscope and 'Help'.
2. It has the facility of a user-definable assembler to support a new version or type of processor. It provides a multiuser environment.

3. The design process divides into number of subparts. Each programmer is assigned independent but linked tasks.
4. It simulates hardware unit-like emulator, peripherals and I/O devices on a host system (PC). It supports conditional and unconditional breakpoints. It provides test-vectors. A test-vector is program-path for the controlled flow of the program used duing testing phase and later removed or disabled on completing that phase.
5. It debugs by single stepping. It has the facility for synchronizing the internal peripherals.
6. It provides Windows on the screen. These provide the detailed information of the source code part with labels and symbolic arguments, the registers as the execution continues, the detailed information of the status of peripheral devices, status of RAM and ports, and the status of stack and program flow as it continues.
7. It verifies the performance of a target system. It has an emulator built into the development system that remains independent of a particular targeted system, plus a logic analyser for up to 256 or 512 transactions on the address and data buses after triggering.

An IDE tool is from WindRiver® Systems and that employs VxWorks RTOS (Section 9.3). An architectural feature is 'dynamic linking and incrementally loading the object modules' into the target system. Exemplary target processor families that are supported are PowerPC, Intel, Motorola, Pentiums, MIPS and ARM/Strong ARM. It helps in prototype development and tests the prototype applications. There is a text editor with GNU C/C++ compilers. Debugging is performed at three levels, source code-level, task-level (scheduling, IPCs and interrupts study) and domain-level. It includes VxSim, stethoscope and tracescope. Figure 13.2(a) and (b) show simple and sophisticated IDE, respectively.

An IDE (µVision_2) is from keil software Inc. with RTX51 RTOS for 8051 target processor families. Another IDE keil µVision_3 is for ARM family of processors and microcontrollers. It has cross-compiler, source-level debugger, object browser, monitor for run-time behaviour, event-to-event viewing. The object browser browses the applications behaviour overtime. It graphically displays the RTOS tasks, queues, semaphores and IPC objects. A real-time analysis (RTA) suite profiles the code coverage and locates run-time errors. It optimizes the use of the memory.

13.2 HOST AND TARGET MACHINES

During the development process, a host system is used before locating and burning the codes in the target board. The target board hardware and software is later copied to get the final embedded system, which will function exactly as the one tested and debugged and finalized during the development process.

13.2.1 Using a Host System

Host system is a PC or workstation or laptop. It has the following hardwares.

1. High-performance processor with caches
2. Large RAM memory
3. ROMBIOS (read only memory basic input–output system)
4. Very large memory on disk
5. Keyboard
6. Display monitor
7. Mice
8. Network connection

It a full-fledged computer. It has software tools (Table 13.1) and must include the following:

1. Program development kit for a high-level language program or IDE.
2. Host processor compiler and cross-compiler.
3. Cross-assembler.

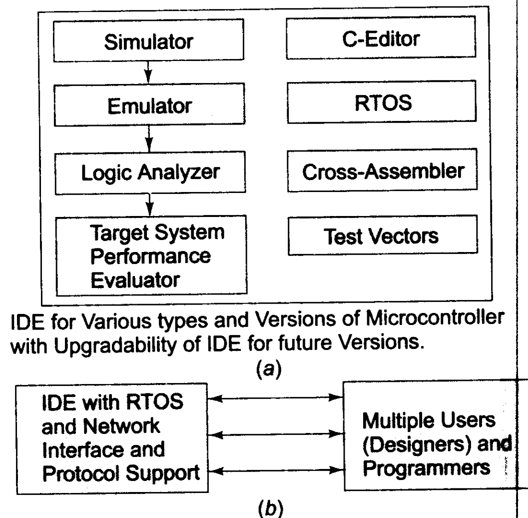


Fig. 13.2 (*a*) Simple integrated development environment (IDE) (*b*) Sophisticated IDE

Program Development Tool Kit Program development tool kit or IDE has an editor. The editor is used for writing C codes or assembly mnemonics or C++ or Java or Visual C++ using the keyboard of the host system (PC) for entering the program. Using GUIs, it allows the entry, addition, deletion, insert, appending previously written lines or files, merging record and files at the specific positions. It creates a source file that stores the edited file. It also has an appropriate name (given by the programmer). It can use previously created files and can also integrate the various source files. It can save different versions of the source files. Program development kit or IDE has the code generation tools (assembler, compiler, loader and linker).

A high-level language is machine-independent. It will have an expression like X = X + 23, or X = 2*Y +V*Z + 19 and so on. When we use a high-level language C, a tool is needed for obtaining the machine codes for a target system. The programmer writes the mnemonics or C program, using the editor. The mice and keyboard combinations of the host system (PC) or host system are for entering the program codes. Each language needs a compiler. The codes may not be executable using an interpreter.

1. An *interpreter* does expression-by-expression (line-by-line) translation to the machine-executable codes.
2. A *compiler* uses the complete set of the expressions. It may also include the expressions from the library routines; that is, standard tailor-made programs. Whereas an interpreter helps in on-line execution of the codes, a compiler helps in the off-line programming for obtaining the executable machine codes later. The C programs are used with an interpreter as well as with a compiler. A cross-compiler is a compiler that created binary executable files for the target system processor.
3. An assembly language program has the mnemonics that are machine-dependent. Example of a mnemonic is SBC A, 0x0B. It means an instruction, which subtracts, along with the previous 'carry', the *A* register of the processor with the hexadecimal number 0x0B. An assembly mnemonic is specific to a processor or microcontroller. It is according to the instructions provided in the instruction set. The assembly mnemonics needs an interpreter to translate into the machine codes that are executed on a specific processing device.
4. A *dissembler* translates the object codes into the mnemonics form of assembly language. It helps in understanding the previously made object codes.
5. An *assembler* is a program that translates the assembly mnemonics into the binary opcodes and instructions, that is, into an executable file, called object file. It also creates a list file that can be printed. The list file has address, source code (assembly language mnemonic) and object codes in hexadecimal. The object file has addresses that are to be allocated again during actual run of the assembly language program. A loader is a program that helps in this task by reallocating addresses before loading the opcode and operands in the computer memory.

6. A *linker* links the needed object code files and library code files. This is before the *loader* reallocates the addresses, and puts the codes at the physical addresses in the memory, and the program runs. Loader performs the analogous functions on host machine as the locator does on a target system in conjunction with a device programmer.

Cross-Compiler C or C++ or visual C++ source files compile according to the native platform (system including OS on which their binary image runs). Java classes compile as byte codes and are therefore platform-independent. A cross-compiler is a compiler that creates binary executable files for the target system processor.

Cross-Assembler It converts object codes or executable codes for a processor to other codes for another processor and vice versa. The cross-assembler assembles the assembly codes of the target processor as the assembly codes of the lets us use a processor of the host system (PC) used in the system development. Later, it provides the object codes for the target processor. These codes will be the ones actually needed in the finally developed system.

Code generation tools are used for creating and compiling at the host sysetm. Then codes are tested at the host sysetm using simulators and number of latest software tools like profiler, memory scope, stethoscope and memory and code coverage scope.

13.2.2 Target System

A target system has a processor, ROM memory for ROM image of the embedded software, RAM for stack, temporary variables and memory buffers, peripherals and interfaces. Figure 13.3(a) and (b) show simple and sophisticated target systems, respectively. Some target systems have 8 or 16 MB flash memory and 64 MB SDRAM. A target system may possess the RS232 as well as 10/100-base Ethernet connectivity or USB port.

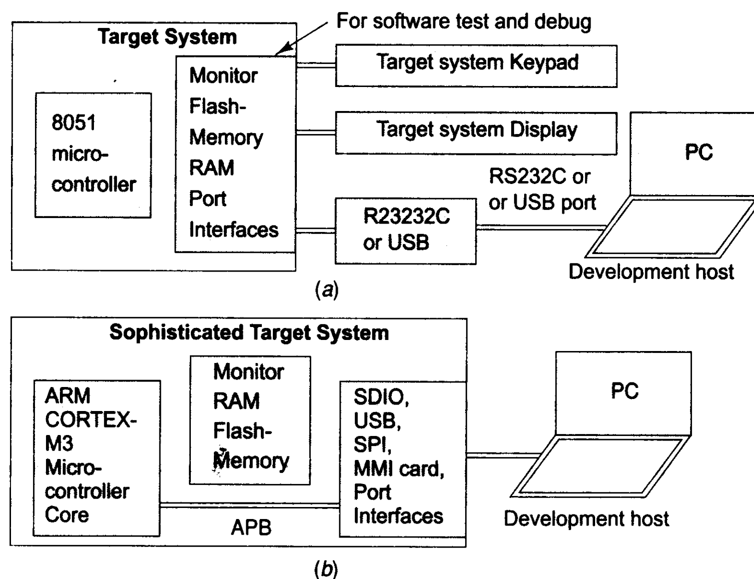


Fig. 13.3 (*a*) Simple target system (*b*) Sophisticated target system

A target system differs from a final system. It interfaces with the computer as well works as a standalone system. There might be repeated downloading of the codes into it during the development phase. The target system or its copies simply work later as the embedded system.

Consider that a targeted system is under development. In the target system development phase, say of a *router*, the codes of application software have to be written. These have to be embedded in flash. These have to be repeatedly written or modified and tested using diagnostic, simulation and debugging tools, and embedded till a final testing in an edit-test-debug cycle shows it working according to specifications. The programmer later on simply copies it into the final system or product. Also a final system may use a ROM in place of flash in the target system.

An exemplary target system is a board that has an Philips LPC21xx processor (ARM microcontroller). It is MC2100 evaluation board from keil.

Let us consider an exemplary sophisticated target system, VxWorks 5.4. It provides run-time support by scaleable RTOS support, Internet protocols support, POSIX library support, file system and graphic supports. It has a debugging agent. It has a back end support package for a specific processor or microcontroller. The target system connects the simulator in parallel the host computer through a target server tool with ICE (Section 14.3.5) using Ethernet or serial lines from the host computer.

13.3 LINKING AND LOCATING SOFTWARE

A *linker* links the compiled codes of application software, object codes from library and OS kernel. Linking is necessary because there are number of codes to be linked for the final binary file. For example, there are the standard codes to program a delay task for which there is a reference in the assembly language program. The codes for the delay must link with the assembled codes. The delay code is sequential from a certain beginning address. The assembly software code is also sequential from a certain beginning address. Both the codes are present at the distinct and the available addresses in the system. A linker links these. The linked file in binary for *run* on a computer is commonly known as executable file or simply '.exe' file. After linking, there has to be reallocation of the sequences of placing the codes before the actual placement of the codes in the memory.

A program is loaded in a computer RAM. The *loader* program performs the task of *reallocating* the codes after finding the physical memory addresses available at a given instant. The loader is a part of the OS and places codes into the memory after reading the '.exe' file. This step is necessary because the available memory addresses may not start from 0x0000, and binary codes have to be loaded at the different addresses during the run. The loader finds the appropriate start address. In a computer, after the loader loads into a section of RAM, the program is ready to run.

When the code embeds into ROM or flash, a system design process *locates* these codes as a ROM image. The codes are permanently placed at the actually available addresses in flash-ROM. In embedded systems, there is no separate program to keep track of the available addresses at different times during the run as in a computer. In embedded systems, therefore next step after linking is the use of a *locator* for the program-codes and data in place of the loader. The locator features are as follows.

1. The locator is specified by the programmer the available addresses at the RAM and ROM in target. The programmer has to define the available addresses to load and create files for permanently locating the codes using a device programmer.
2. It uses cross-assembler output, a memory allocation map and provides the locator program output file. It is the final step of software design process for the embedded system. Locator program output is in the Intel hex file or Motorola S-record format. The locator uses the cross-compile codes in different

cross-compiled segments for: (i) instructions, (ii) initialized values and addresses, (iii) constant strings and (iv) un-initialized data.

3. The locator locates the I/O tasks and hardware device-driver codes at the addresses without reallocation. This is because the port and device addresses for these are fixed for a given system. These are as per the interfacing circuit between the system buses and ports or devices.
4. The *locator* program reallocates the linked file and creates a file for permanent location of codes in a standard format.
5. The file format may be Motorola S-record format or Intel hex file or any other format (Section 13.3.2).

Figure 13.4 shows various software tools and chain of actions of linker at host and locator in an embedded system.

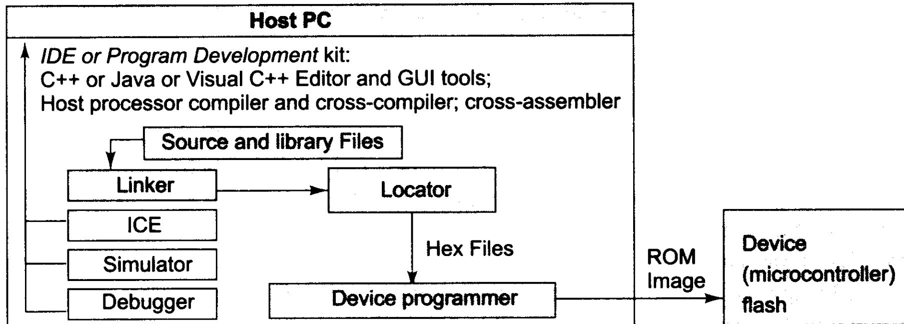


Fig. 13.4 Various software tools and chain of actions of linker at host and locator in an embedded system

13.3.1 Differences in Files, Addressing and Address Resolution Method

Table 13.2 gives the differences in addressing in linker and locator.

Table 13.2 Differences in Files, Addressing in Linker and Locator

Action	*Difference*
File creation	Linker creates linked file for the disk for use by the host system OS and the loader and then does the memory address allocation. Locator creates linked file for the use of a device-programmer, which copies the file at the device EEROM or flash and copied file runs at the located program directly.
File format	Linker file formats are as per the file system used for the disk. Locator file is as per Motorola-S or Intel hex or any other format (Section 13.3.2).
Addresses	Linker addresses are for the host system processor and are relative addresses, which the loader reallocates. Locator addresses are for the target system processor and are addresses, which are not reallocated later.
Address resolution method	An instruction may have specified address in object file, while actual host or target may be allocated different address-spaces for calling that object file. Addresses are properly resolved in the linker as well as locator. Linker uses relative addresses and actual addresses are allocated at run time when the OS does the memory allocations and the loader loads the program. Locator uses addresses, which once allocated remains permanent as the created file records of locator burns (embeds) into the system.

13.3.2 Locator Output File in Binary Image Motorola-S and Intel Hex Formats

Binary bit mapped (binary image) means bytes are sent in a sequence as per starting address to the end address.

Motorola S-Record Format Motorola S-record format is an industry standard for storing the locator file, before its use by the device programmer or ROM-mask programmer. It is called S-record because it has first character as 'S' in each line. A line is as follows: first character is S, second character is 2 (for specifying the record type), third and fourth characters are for a hexadecimal number, say 14 (to specify that there are 20 bytes in that line), the remaining 40 characters (nibbles) divide as the address (3 bytes) and data (16 bytes) and checksum (1 byte). Table 13.3 shows a typical S-record as a locator output and device programmer input. It is left as an exercise to the reader to show that *Addr* for line 6 in the record of Table 13.3 will be 0x000037.

Intel Hex File Format Intel hex file format is another industry standard for storing the locator file output, before its use by the device programmer or ROM-mask programmer. A line is as follows: first character ':' (colon), second and third characters for data counts (assume = 10 in hexadecimal in case N_d = 16) in the line (address bytes, checksum byte and data type bye excluded, only actual data bytes at the line, which are to be burned in ROM are counted), fourth to seventh address (2 bytes), sixth and seventh as 0 and 0 to specify data as ROM data and the remaining 32 characters as the data (16 bytes) and 2 characters for the checksum (1 byte). Table 13.4 shows an Intel hex file, which corresponds to the same data as at the Motorola S-record in Table 13.4 as a locator output and device programmer input. It is left as an exercise to the reader to show that *Addr* for line 6 in the record of Table 13.4 will be 0x0037.

Table 13.3 An Exemplary Motorola S-Record format

Line Number[1]	*First Character*	*Second Character*[2]	*Third and Fourth Characters for N*[3]		*Address, Addr*[4]	*N_d*[5] *Bytes for Storage in ROM from Addr (Maximum value of N_d can be 253 decimal)*	*Check-sum*[5]
0	S	2	1	0	000000	aa bb cc dd ee ff xx yy zz bb cc dd	cs0
1	S	2	0	C	00000C	cc aa cc dd ee ff xx yy	cs1
2	S	2	1	2	000014	dd bb cc dd ee ff xx yy zz bb cc dd aa xx	cs2
3	S	2	0	5	000022	0A	cs3
4	S	2	0	8	000023	dd bb cc dd	cs4
5	S	2	1	4	000027	dd bb cc dd ee ff xx yy zz bb cc dd aa ff 01 c0	cs5

[1] Line number is not in the record.
[2] 2 means the availability of data record in this line. A byte from the data sequentially burns at the ROM.
[3] N = 10 means that there are 16 hexadecimal bytes in this line including the 3 bytes for the address and 1 byte for checksum at the end of the line. Number of data bytes for storing are specified in this line, N_d = 12 decimal. 0C means N = 12 and N_d = 8 decimal.
[4] Starting address of 3 bytes, 000000 means the next 12 bytes store between address 0x000000 to 0x00000B. Therefore, in the next line starting address Addr = 0x00000C.
[5] Bytes for burning in ROM are in this line and numbering = N_d. Each character in this column represents a nibble. cs0, cs1, .. are the checksums of 1 byte each of all the bits in line number 0, 1, ..., respectively.

Table 13.4 An Exemplary Intel Hex File format

Line Number[1]	First Character	Second and Third Characters for C[2]		Address, Addr[3]	Sixth and Seventh Characters[4]		N_d[$] Bytes for Storage in ROM from Addr (Maximum value of N_d can be 253 decimal)	Check-sum[5]
0	:	0	C	0000	0	0	aa bb cc dd ee ff xx yy zz bb cc dd	cs0
1	:	0	8	000C	0	0	cc aa cc dd ee ff xx yy	cs1
2	:	0	E	0014	0	0	dd bb cc dd ee ff xx yy zz bb cc dd aa xx	cs2
3	:	0	1	0022	0	0	0A	cs3
4	:	0	4	0023	0	0	dd bb cc dd	cs4
5	:	1	0.	0027	0	0	dd bb cc dd ee ff xx yy zz bb cc dd aa ff 01 c0	cs5

[1] Line number is not in the record.
[2] Number of data bytes for storing specified in this line, C_d = 12 decimal. 0C means C_d = 12.
[3] Starting address of 2 bytes, 0000 means the next 12 bytes store between address 0x0000 and 0x000B. Therefore in the next line starting address Addr = 0x000C.
[4] 0 and 0 means the availability of data record in this line is for the ROM. A byte from the data sequentially burns at the ROM.
[5] Bytes for burning in ROM are in this line and numbering = N_d. Each character in this column represents a nibble. cs0, cs1, .. are the checksums of 1 byte each of all the bits in line number 0, 1, .., respectively.

13.3.3 Memory Map for coding a locator

Figure 13.5(a) shows memory addresses needed in the case of Princeton architecture in the system. Figure 13.5(b) shows memory addresses needed in the case of Harvard architecture. These differ in following respect.

1. Vectors and pointers, variables, program segments and memory blocks for data and stacks have different addresses in the program in Princeton memory-architecture.
2. Program segments and memory blocks for data and stacks have separate sets of addresses in Harvard architecture. Control signals and read-write instructions are also separate.

The *system memory allocation map* is not only a reflection of addresses available to the memory blocks, and the program segments and addresses available to the IO devices, but also reflects a description of the memory and IO devices in the system hardware. It maps guides to the actual presence of the various memories at the various units, EPROM, PROM, ROM, EEPROM, Flash memory, SRAM (static RAM), DRAM (dynamic RAM) and IO devices. It reflects memory allocation for the programs, and data and IO operations by the locator program. It shows the memory blocks and ports (devices) at these addresses. Figure 13.6(a) and (b) show memory and I/O devices memory allocation map for the 68HC11 (having memory-mapped IO architecture), and for an IBM 80x86 PC) (having IO-mapped IO architecture), respectively.

Four examples of memory allocation maps are given in Figure 13.7(a)–(d). *System I/O devices map* may be designed separately. An IO map not only reflects the actual presence of the I/O devices, but also guides the available addresses of the various device registers and port data. (An example of a device is a timer. I/O devices are the peripheral units of the system.)

Memory map is used for coding locator software. The memory map defined for a locator includes the device I/O addresses designed after appropriate address allocations of the pointers, vectors, data sets and data structures. From the map, the locator program input can be easily designed. When the main memory is of Harvard architecture, the program memory map will be separate, for example, 8051. The processor reads from the program memory by a separate set of instructions (input–output instructions) and control signals.

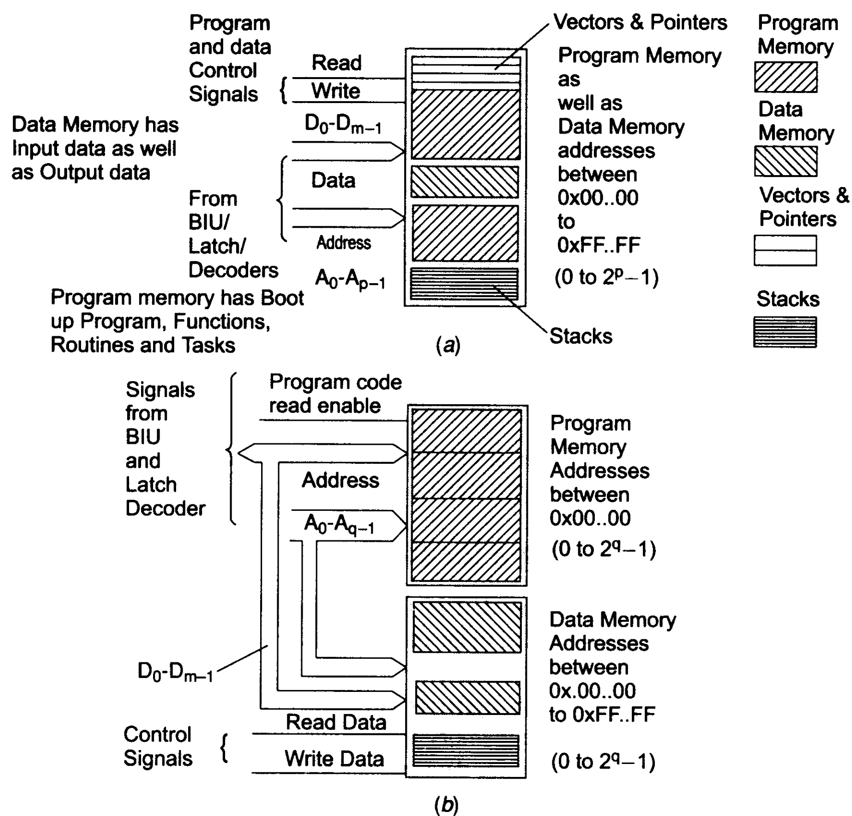


Fig. 13.5 (*a*) Memory map (Princeton architecture) (*b*) Memory map (Harvard memory architecture)

13.4 GETTING EMBEDDED SOFTWARE INTO THE TARGET SYSTEM

13.4.1 Device PROM or Flash Programmer

Device Programmer This is also called laboratory programmer which is a programming system for a device. The device is selectable and may be a PROM or EPROM chip or a flash or a unit in a microcontroller or PLA, GAL or PLC. The selected device inserts into a socket (at the device programmer circuit) and is programmed (burned the codes) by transfer of the bytes for each address using the software at the host.

The software of the device programmer runs at a host system (PC or workstation or laptop). The host system interconnects with the socket and the device programmer circuit usually through a serial port (UART or USB).

Device programmer software running at the host uses an input file from the locator software output records. The file reflects the final design and has a bootstrap program plus the compressed record, which the processor decompresses before the embedded system processor starts execution. (Bootstrap program is the program to start up a system. We start from home by strapping our boots.)

Note: An IDE incorporates the device programmer within it.

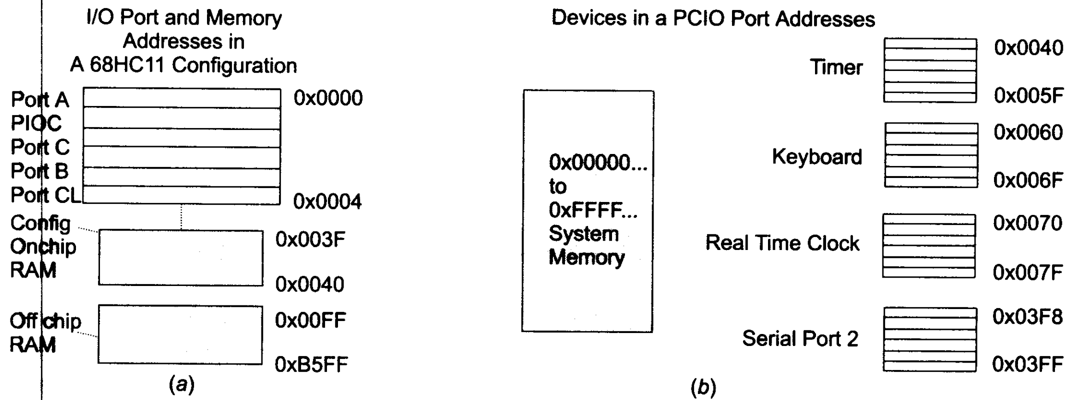


Fig. 13.6 (*a*) IO port, memory and device address spaces in 68HC11 (*b*) Device addresses in 80x86-based host system (PC)

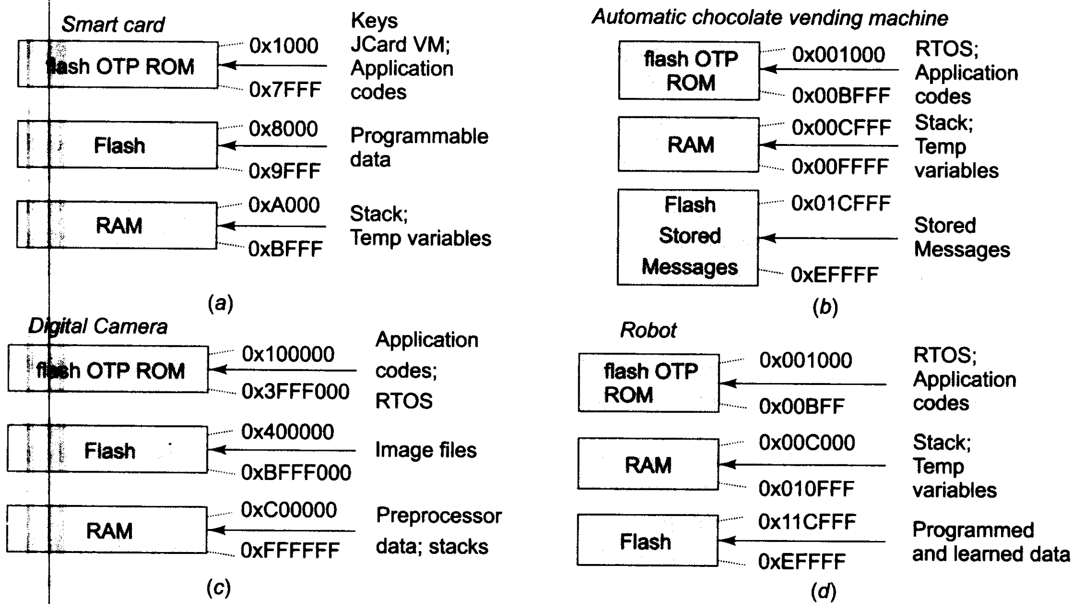


Fig. 13.7 (*a–d*) Four memory allocation maps in four exemplary systems for their locator programs

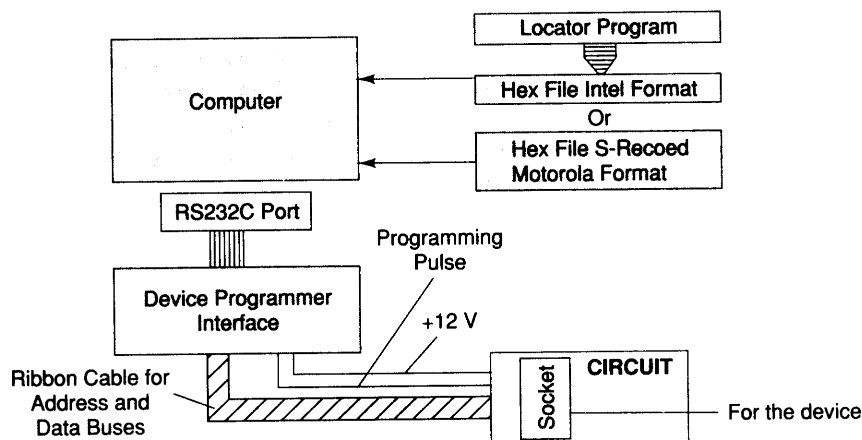


Fig. 13.8 Burning in of the application software codes, data and tables using a device programmer

Use of Device Programmer for Downloading the Finalized Codes into PROM or Flash A locator output is of the final design with a booting program plus the system program (with or without a compression) plus initial data and shadow RAM data. Assume that a system design phase up to the target system is over. Refer to a memory map of the target PROM or flash. Finalized ones are put in non-volatile memory at each memory address into the system by a process called *burning*.

Burning is a process that places the codes. Codes are the ones to be downloaded, according to ROM image (*locator* output). Burning is done in the laboratory using a device programmer into an erased EPROM or EEPROM or PROM or flash.

Figure 13.4 showed the method for burning-in the EPROM or in the EEPROM the S record or hex file generated by a locator (Section 13.3.2). EEPROM does no erasing and can be programmed directly by the device programmer. Flash uses a different file system.

Consider a device that has a 512 kB programmable memory. It means that it has eight signals (D_0 to D_7) and 19 [$= \log_2 (512^* \ 1024)$] address (A_0 to A_{18}) signals. There are a total of 5,24,288 (=512* 1024) arrays of cells with each array having eight cells. Each cell has for output the D bit as logic 1 in the un-programmed (fresh) state. Programming the device (locating the desired bytes) means replacing 1s with 0s according to each byte needed at each cell-array address. Bytes saved are as generated by the locator program of the embedded system after the programming (after programming, the device memory part will hold the final bytes according to the need after the completion of software development, testing and debugging cycle of design).

A ROM device programmer is a programming system for the PROM or EPROM or flash in a chip or unit of a microcontroller or device. A device inserted into a socket (at the device programmer circuit) is programmed on transferring the bytes for each address using a software tool at the computer and interconnecting the computer with this circuit. The device program needs the locator output records in the input. This output must reflect the final design, only then can the device program put the final inputs into the ROM. Records that are input to a device programmer as per the device being programmed, are in three formats. The file formats for burning the binary image are described in Section 13.3.2.

Alternative to a device programmer is that the application software codes are sent in a tabular form to a specialized manufacturer. The manufacture prepares the ROM. The ROM is needed especially when many thousands of pieces are needed. Integration of ROM with the processor, RAM and other hardwares of the target system gives the final product.

13.4.2 Programming Method of Device Programmer

A 512 kB device cell-array (at the address defined by A_0 to A_{18} signals) stores the '0's as per 0s at D_0 to D_7 when a strobe pulse of a few microseconds duration is applied in the presence of a voltage V_p by the device programmer circuit. A device programmer that programs its memory unit performs the following eight steps in a sequence using the software tool, computer and the device programmer circuit. (i) Applies the A_0 to A_{18} bits as needed at a selected address input of the array of cells. (ii) Applies as inputs the D_0 to D_7 bits that are meant for that address. (iii) Applies V_p to make programming feasible for the needed duration in microseconds. (iv) Applies a programming pulse for a sufficient duration to cause fusing of the desired links in the array, to convert a '1' to '0'. (v) Switches off the V_p. (vi) Applies a next higher address than the previous one. (vii) Repeats the aforementioned steps (ii) to (iv) for writing (converting) the logic states of D_0 to D_7 bits at current instance at new address. (viii) Continues till a cell array at last desired address is programmed.

The process of writing into the device is by converting logic 1 to 0, and is done by fusing the *links* on applying programming voltage and programming pulse for a short duration.

The working of a device programmer is according to the processing and memory device.

Using EEPROM 68HC11 Example The working of a device programmer for programming the internal EEPROM of 68HC11 is as follows. The 68HC11 has a control register CONFIG. It is for system configuration control. It keeps the data bits like an internal *EEPROM* address. It is also called an EEPROM register whenever the programming unit is used within 68HC11. There is another register, the EEPROM register, at the address 0x003B. This register is kept to program both CONFIG as well as the *EEPROM* addresses in 68HC11 (the on-chip *EEPROM addresses* in 68HC11 are from 0xB600 to 0B7FF). If CONFIG.0-bit (EPROM) is '0', the erase or write (burn in) does not become feasible by any instruction. To burn in, first, the 0th bit of CONFIG (EEPROM) register is made '1'. Only then is the EEPROM programming voltage can be ON. If the first bit is also made '1' the *EEPROM* addresses and their data are latched with the help of programming units within 68HC11. If the register EEPROM.3 and .4 bits becomes 00 (0 and 0, respectively) bulk erase takes place at the EEPROM addresses (bulk erase refers to all the available *EEPROM* addresses). If 01 is written a byte is erased (byte erase means erase at one *EEPROM* address only). If 10, then a row of 16 bytes is erased. If the second bit in EEPROM is made '1' then only is the erase function enabled. An erase means all bits at an EEPROM address are made'1's. The erase time is a total of 10 ms in all these three modes. When the erase function is disabled (due to second bit = '0' but programming voltage becoming enabled because the 0th bit = '1'), the burn in of bytes takes place by the execution of the write instruction for the appropriate address.

A computer's RS232C UART port that sends and receives at 1200 baud and connects to RxD and TxD pins in 68HC11 through a line receiver and a line driver, respectively. VDD, VRH, IRQ, XIRQ pins are at +5V. VSS and VRL are at '0'. The reset circuit and 8 MHz crystal circuit connect as usual. Once 68HC11 is configured in the bootstrap mode, the bits at its *EEPROM* addresses are programmable using the software in the computer that is a part of the development system. An external computer transfers at 1200 baud in bootstrap mode when using an 8 MHz crystal with 68HC11. Further, the transfer is first of a byte with all '1's (0xFF). The computer then transfers 256 bytes to 68HC11. These bytes load between 0x0000 and 0x00FF internal RAM addresses. 68HC11 automatically sets its PC at the end of transmission after reading its vector from 0xFFFE. It, therefore, starts executing a program called bootstrap program. This can be a test program that includes the write to CONFIG or EEPROM register and to *EEPROM* addresses.

Using EPROM When using EPROM of the processing device (e.g., microcontroller), PROM is first erased in ultraviolet (UV) light. Erasing makes all the 8 bits at each of the addresses into '1's. An erase facility provides reusability of the memory whenever the application software changes for another version

of the system. The software executed in the computer programs the EPROM as well as *verifies* the bytes burned into the EPROM with the help of an interfacing circuit between the EPROM and computer's RS232C serial port. The EPROM interface circuit receives a byte serially from the computer through the TxD line and later sends, along with its address, this byte for burning in the processing device's EPROM. Burn-in of codes is done as follows: During the period when the appropriate address and data are available from this circuit, it also switches ON a high-voltage ~V_pVolt and applies a program pulse for a needed period. This circuit is sequentially programmed at each address by increasing the address after every program pulse.

An EPROM interface circuit also receives another byte serially from the computer through a TxD line and sends this byte again for burning-in the processing device at the appropriate address. The bytes at the successive addresses are received by the computer in a verify mode through the interface and RxD line. Some processing devices have an auto program mode for its EPROM in which it can automatically copy the codes and data from an IC.

13.5 ISSUES IN HARDWARE–SOFTWARE DESIGN AND CO-DESIGN

There are two approaches for the embedded system design.

(1) The software development life cycle ends and the life cycle for process of integrating the software into hardware begin at the time when a system is designed.

(2) Both cycles concurrently proceed when co-designing a time critical sophisticated system.

The final design, when implemented, gives the targeted embedded system and thus the final product. Therefore, an understanding of the (i) software and hardware designs and integrating both into a system and (ii) hardware–software co-designing are important aspects of designing embedded systems.

Further there is hardware–software trade-off. Certain embedded components, for example, CCD co-processor, CODEC execute fast when implemented by hardware but the design cost is high and the processor performance requirements are also high.

Let us refer to an interview of Jean-Louis Brelet in an article 'Exploring Hardware/ Software Co-design with Vertex-II Pro FPGAs' (*Xcell Journal*, pp. 24–29, Summer issue, 2002). A Brelet reply, quoted verbatim, when asked about the expertise required for successful implementation is as follows: 'Software people must understand the nature of hardware design and type of problems encountered by hardware team. They also must understand the possibilities and capabilities of hardware. Likewise hardware team must have a good understanding of software and how the applications operate. Both teams must have a good understanding of each other's language and a willingness to adapt'.

The selection of the right hardware during hardware design and an understanding of the possibilities and capabilities of hardware during software design is critical especially for a sophisticated embedded system development such as Apple iPhone.

13.5.1 Choosing Right Platform

Software Hardware Tradeoff There is a tradeoff between the hardware and software. Hardware implementations provide advantage of processing speed. It is possible that certain subsystems in hardware—controller, IO memory access circuit, real-time clock, system clock, pulse width modulation, timer and serial communication are also implemented by the software. A serial communication real-time clock and timers featuring microcontroller may cost more than the microprocessor with external memory and a software implementation.

Hardware implementation provides the following other advantages. (i) Reduced memory for the program. (ii) Reduced number of chips but at an increased cost. (iii) Simple coding for the device drivers. (iv) Internally embedded codes, which are more secure than at the external ROM.

Software implementation provides the following advantages. (i) Easier to change when new hardware versions become available. (ii) Programmability for complex operations. (iii) Faster development time. (iv) Modularity and portability. (v) Use of standard software engineering, modelling and RTOS tools. (vi) Faster speed of operation of complex functions with high-speed microprocessors. (vii) Less cost for simple systems.

It may be or may not be possible that certain subsystems in hardware (ASIP, microcontroller, DSP, single-purpose processors) are implemented by software to get desired performance with the least system cost.

Choosing a Right Platform System design of an embedded system also involves *choosing a right platform*. A platform consists of a number of units. Table 13.5 shows a list of these units and various corresponding sections, which a programmer can refer to while selecting the unit to finally obtain a right platform and right development tools.

Table 13.5 List of Units to Choose for Finally Obtaining a Right Platform and Right Development Tools

Unit to be Choosen	*Section Describing that in Detail to Enable the Right Choice*
Processor	Sections 1.2, 2.1 and 2.3
ASIP or ASSP	Sections 1.2.3 and 1.2.4
Multiple processors	Section 1.2
System-on-chip	Section 1.6
Memory	Section 2.7.1
Other hardware units of system	Section 1.3
Buses	Sections 2.1.4, 2.2.1, 3.2, 3.3, 3.10, 3.11 and 3.12
Software language	Sections 5.1, 5.5, 5.6 and 5.7
RTOS (real-time programming operating system)	Sections 8.8, 8.9, 9.2, 9.3, 10.1, 10.2 and 10.3
Code generation tools	Section 13.1
Tools for finally embedding the software into binary image	Section 13.3

Embedded System Processors Choice

(A) Processor-less System: We have an alternative to a microprocessor or microcontroller or DSP. Figure 13.9(a) shows the use of a PLC in place of processor. We can use a PLC for the clothes-in clothes-out type system (Section 1.1.1). A PLC fabricates by the programmable-gates, PALs, GALs, PLDs and CPLDs.

A PLC has very low operation speed. It also has a very low computational ability. It has very strong interfacing capability with its multiple inputs and outputs. It has system-specific programmability. It is simple in application. Its design implementation is also fast. Automatic chocolate-vending machine can be another exemplary application of PLC.

(B) System with Microprocessor or Microcontroller or DSP: Section 1.2 gave a detailed description of the processors described for the embedded systems in detail. Figure 13.9(b) shows the use of a microprocessor or microcontroller or a DSP.

(C) System with Single-Purpose Processors and IPs in VLSI or FPGA: A line of action in designing can be use of the IPs, synthesizing using VHDL-like tool and embedding the synthesis into the FPGA.

Figure 13.9(c) shows the processing of functions by using IPs embedded into VLSI or FPGA instead of processing by the ALU. The IPs implement the functions, which if implemented with the ALU then coding by programmer will take a long development time.

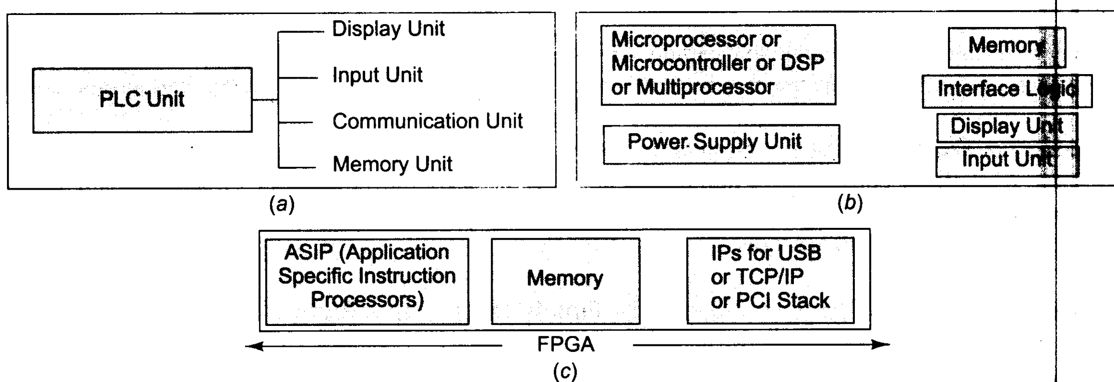


Fig. 13.9 (*a*) Use of a PLC in place of a processor (*b*) Use of a microprocessor or microcontroller or a DSP (*c*) Processing of functions by using IP embedded into the FPGA instead of processing by the ALU

The sophisticated operation parts on a VLSI chip implement using copyrighted IPs. Each IP is synthesized at gate level using VHDL or Verilog. (VHDL (VLSI high-level description language) and Verilog are the languages for simulating and synthesizing the gate-level design. VHDL also implements concurrency and synchronization problems and a structural hierarchy strategy. In addition to these features, Verilog uses C-like functions. Therefore, the exception handling and timing problems are also programmable. There are two languages for programming and implementing FSM, state transitions, concurrency, synchronization and behavioural hierarchy. These are StateCharts and SpecCharts.)

(D) Factors and Needed Features Taken into Consideration: We consider a general purpose processor choice or choose an ASIP (microcontroller or DSP or network processor). When the 32-bit system, 16kB+ on chip memory and need of cache, memory management unit or SIMD or MIMD or DSP instructions arise, we use a microprocessor or DSP. For example, the video game, voice recognition and image-filtering systems will need a DSP. Table 13.6 gives the factors that are considered by a system programmer before choosing a microprocessor or microcontroller as a processing unit.

Refer Section 2.1 for 8051. Microcontroller provides the advantage of on-chip memories and subsystems like the timers. Table 13.6 can help in selecting the microcontroller. A microcontroller available at a reasonable cost may not support these on-chip needs. Then decide which one will suffice, 8-bit or 16-bit or 32-bit ALU. Now take a decision about which microcontroller and its version with what features is needed. First selection criterion is on-chip memories needed for the embedded system. A second criterion is the on-chip timers and serial communication subsystems needed in each system. Third and fourth may or may not be needed in the system being designed. The third criterion is the need for input captures (interrupt and load time on an input) and out-compares (output and interrupt when timer contents equal a comparison register). Fourth is about PWM and/or ADC on-chip availability. Latest versions for 8-, 16- and 32-bit microcontrollers can be found from the websites of the giants, ARM, Intel, Motorola, Phillips and Microchip (for 8-bit systems).

Table 13.6 Factors and Needed Features in the Microprocessor or Microcontroller or DSP-Processing Unit of the System

Factors for On-Chip Feature	*Needed or which One Needed*	*Available in Chosen Chip*
8-bit or 16-bit or 32-bit ALU	8/16/32	8/16/32
Cache, memory management Unit or DSP calculations	Yes or no	Yes or no
Intensive computations at fast rate	Yes or no	Yes or no
Total external and internal memory up to or more than 64 kB	Yes or no	Yes or no
Internal RAM	256/512 B	256/512 B
Internal ROM/EPROM/EEPROM	4 kB/8 kB/16 kB	4 kB/8 kB/16 kB
Flash	16 kB/64 kB/1 MB/8 MB	16 kB/64 kB/1 MB/8 MB
Timer 1, 2 or 3	1/2/3	1/2/3
Watchdog timer	Yes or no	Yes or no
Serial peripheral interface full duplex or serial Synchronous communication interface (SI) half duplex	Full/half	Full/half
Serial UART	Yes or no	Yes or no
Input captures and Out-compares	Yes or no	Yes or no
PWM	Yes or no	Yes or no
Single- or multi-channel ADC with or without programmable voltage reference (single or dual reference)	S/M W/WO V_{ref} S/D	S/M W/WO V_{ref} S/D
DMA controller	Yes or no	Yes or no
Power dissipation	Very low/low or normal	Very low/low or normal

13.5.2 Memory- and Processor-Sensitive Software

Table 13.7 gives the examples of memory-, processor-sensitive programs.

13.5.3 Allocation of Addresses to Memory, Program Segments and Devices

Functions, Processes, Data and Stacks at the Various Segments of Memory *Program routines and processes can have different segments. For example, a program code can be segmented and each segment stored at a different memory block.* A *pointer* points to the start of the memory block storing a segment and an *offset* value is used to retrieve a memory address within that segment.

There can be different segments at the memory for the functions and processes (treads or tasks). These can comprise of different segments for data and different segments for the stacks. Each segment has a starting memory address and ending memory address. Each segment has a pointer address and an offset address. Using offset, a code or data word is retrieved from a segment.

There can be different sets and different structures of data at the memory (Sections 5.4.2 and 5.4.3). Following are the examples of the data structures and data sets that are commonly used during processing in a system and that are stored at the different memory blocks in a system.

Data structure, called *stack* is a special program element. A *stack* means an allotted memory block from which a data element is always read in a LIFO mode by the processor. Various stack structures may be created during processing. A call can be made for another routine during running of a routine. In order that on

completion of the called routine, the processor returns only to the one calling, the instruction address for return must be saved on the stack. There can also be nesting. It means one routine calling another, and that calling another and return from the called routine is always to the calling routine. Therefore, at the memory a block of memory address is allocated to the stack that saves the *return addresses* of the nested calls.

1. There may be at the beginning an input data saved as a stack at RAM in order to be retrieved later in the LIFO mode. An application may create the run-time stack structures. There can be *multiple data stacks* at the different memory blocks, each having a separate pointer address (Figure 5.1).

Table 13.7 Hardware-Sensitive Programming

Program	*Examples*
Processor-sensitive	Recall Sections 2.1, 2.3 and 2.4. A processor has different types of structural units. It can have memory-mapped IOs or IO-mapped IOs. IO instructions are processor-sensitive. A processor may be having fixed-point ALU only. Floating-point operations when needed are handled differently than in a processor with floating-point operations. A processor may not provide for execution of SIMD (single instruction multiple data) and VLIW (very large instruction word) instructions. Programming of the modules needing these instructions is handled differently in different processors. Assembly language may sometimes facilitate an optimal use of the processor's special features and instructions. Advanced processors usually provide the compiler or optimizing compiler subunit to obviate need for programming in assembly.
Memory-sensitive	(i) An example of a memory-sensitive program is video processing and real-time video processing. The picture resolution actually used for processing and number of frames processed per second will depend on the memory available as well as the processor performance. If a large memory is available, then higher resolution pictures can be processed. When higher resolution without acceptable missing frames is used, then for the given performance of processor in MIPS the less number of frames can be processed. Real-time programming model and algorithm used by a programmer will depend on memory available and processor performance. (ii) Memory address of IO device registers, buffers, control registers and vector addresses for the interrupt sources or source groups are prefixed in a microcontroller. Programming for these takes into account these addresses. The same addresses must be allotted for these by the RTOS. Memory-sensitive programs need to be optimized for the memory use by skillful programming. (iii) When using certain instruction sets like Thumb® in ARM processor helps 16-bit instructions, which save less memory space than the use of 32-bit ARM instruction set.

2. Each task or thread in a multitasking or multithreading software (Sections 7.1 to 7.3) should have its own stack where its *context* is saved. The context is saved on the processor switching to another task or thread. The context includes the return address for PC for retrieval on switching back to the task. There are *multiple stacks* at the memory for the different contexts at the different memory blocks, each having a separate pointer address. Application programs and supervisory programs (OS) have separate stacks at separate memory blocks.

Each processor has at least one stack pointer so that the instruction stack can be pointed and calling of the routines can be facilitated. When a processor has only one stack pointer, the OS allocates the memory addresses that are used as the pointers for the multiple instruction and data stacks of the tasks (or processes or threads).

A stack is a special data structure at the memory. It has a pointer address that always points to the top of a stack. This pointer address is called a stack pointer.

The other data sets, which are also allotted memory, are as following.

1. A *string* is allotted memory for ASCII (8-bit) or Unicode (16-bit) characters, followed by a null character at the end. A string object is allotted memory for the fields for string characters and for the methods to manipulate the string (e.g., concatenation).
2. A *circular queue* is allotted addresses for a queue in which both pointers cannot increment beyond the memory block (buffer) and reset to starting value on insertion beyond the boundary (Figure 5.2).
3. A *one-dimensional array* is allotted addresses for a special data structure at the memory. It has a pointer address that always points to the first element of the array. From the first element pointer and index of that element, an address is constructed from which the processor can access one of the array elements. Index is an integer that starts from 0. Data word can be retrieved from any element address in the block that is allocated to the array.
4. A *table* is a two-dimensional array (matrix) data set that is allocated a memory block. Three pointers, table-base, column-index and destination-index pointers can retrieve an element of the table. There is always a base pointer for a table. It points to its first element at the first column first row. There are two indices, one for a column and the other for a row. Figure 5.3(a) shows a memory block with the pointers for a table.
5. A *hash table* is allocated a memory block for a data set that is a collection of pairs of a key and a corresponding value [Figure 5.3(b)]. A hash table has a key or name in one column. The corresponding value or object is at the second column. The keys may be at non-consecutive memory addresses. Just as an index identifies an array element, a hash key identifies a hash element.
6. Look-up tables have columns and store the pointers to the values. The first column of a table is used as a pointer to the value to get the set of values.
7. A list is allotted memory for a data structure in which each element also stores a pointer to the next element at list. It has one memory block allotted to each of its elements. The list-top pointer points to its first element and the last element points to null [Figure 5.3(c)]. A *list* is a data structure with a number of memory blocks, one for each element. A list has a top (head) pointer for the memory address from where it starts. Each list element at the memory also stores the pointer to the next element. The last element points to *null*. A *list* is for non-consecutively located objects at the memory.

Device, Internal Devices and I/O Device Addresses and Device Drivers All I/O ports and devices have addresses. These are allocated to the devices according to the system processor and the system hardware configuration.

I/O device addresss are considered as part of the memory addresses by certain processors. Certain processors provide for configuring the memory addresses. On the other hand, the 8051, 80196 and 80196 microcontrollers have pre-assigned device addresses for its internal devices and these are un-configurable addresses.

I/O device addresses are not the part of the memory addresses in 80x86 processors. (Figure 2.8(b)).

Sections 3.2 and 3.3 described I/O serial and parallel devices in detail. Device addresses are used for processing by the *driver* (Section 4.9). A device has an address, which is usually according to the system hardware or may also be the processor-assigned ones. These addresses allocate to the following.

1. Device data register(s) or RAM buffer(s).
2. Device control register(s). It saves control bits and may save configuration bits also.
3. Device status register(s). It saves flag bits as device status. A flag may indicate the need for servicing and show occurrence of a device interrupt.

Each device, and thus each device register must be allocated addresses at the memory map. A very important point to remember is that in most cases, each set of IO device addresses is often fixed by the system hardware. A locator or loader cannot reallocate these to any other set of addresses. Another point to remember is that

depending on the device, at a device address there can be one or a number of device registers. A physical or virtual device can be configured to attach or detach from receiving input and sending output. A device address can also be just like a file, making it read only or write only or both read and write only.

The address of I/O device registers, buffers, control registers, vector addresses for the interrupt sources or source groups are prefixed. Similarly, the addresses of device control register bits and status register bits are prefixed. Programming of each bit is used in different functions of the device. Device-driver codes implementation is hardware-dependent. Open source drivers are available for ports, buses and physical media attachments in Linux. Device drivers in Linux let us use each module of a class of device register, de-register and schedule like a process. Programmers can port these directly as these are open sources also.

Appropriate interface functions are needed for porting into system the processor-sensitive memory-sensitive programs and ISRs. Appropriate drivers are needed for device-sensitive programs.

Example 13.1 gives the details of addresses of the registers of an I/O device, *serial-line* UART *device* (Section 3.2).

Example 13.1

A serial-line device has the addresses of device registers as follows: These addresses are fixed by its hardware configuration of UART port interface circuit in a system employing 80x86 processor. They are from 0x2F8 to 0x2FE at COM1 in a PC.

1. (a) Two I/O data buffer registers (one for receiving and the other for transmitting) are at a common address, 0x2F8. Provided a control bit at address 0x2FBH is 0, during read from the address, the processor accesses from the RBR (receiver data buffer register) of the device and during write to the address, the processor accesses the TRH (transmitter-holding register) of the device at 0x2F8H. (b) Provided a control bit at address 0x2FB is 1, data of two bytes of *divisor latch* are at the distinct addresses, 0x2F8 (LSB) and 0x2F9 (MSB). Divisor latch holds a 16-bit value for dividing the system clock. This then selects the rate of serial transmission of bits at the line. (While writing a device driver, remember that a bit in another register (control register) changes the 0x2F8 from an IO register to lower byte of the divisor latch register.
2. *Three control registers of the device are at three distinct addresses* 0x2FA, 0x2FB and 0x2FC. These are as follows: (a) IER (interrupt-enabling register). It enables the device interrupts. (b) LCR (line control register). It defines how and how many bits will be on the line. (c) MCR (modem control register). It defines how the modem handshakes and communicates.
3. *Three status registers of the device are at three distinct addresses* 0x2FA, 0x2FD and 0x2FE. These are as follows: (a) IIR (interrupt identification register) at 0x2FA. It has the flags. A flag sets on a device interrupt and resets at system reset and at servicing of corresponding device interrupt. (b) LCR at 0x2FD. It defines how and how many bits will be on the line. (c) MCR at 0x2FE. It defines how the modem handshakes and communicates.

An I/O device is at a distinct addresses. The device has three sets of registers: data buffer register(s), control register(s) and status register(s). There can be one or more device registers at a device address. The addresses of a device are according to the system processor and the system hardware configuration. Most processors process the memory devices and other devices with the same instructions. 80x86 processors process the IOs with a different set of instructions (input–output instructions).

13.5.4 Porting Issues of OS in an Embedded Platform

The following portability issues may arise when OS is used in an embedded platform. Table 13.8 gives the platform-dependency issues and the need for appropriate OS-Hardware interface functions for each issue.

Table 13.8 Platform-Dependency Issues and Need for Appropriate OS–Hardware Interface Functions

Platform Dependency	*Need of Appropriate OS–Hardware Interface Functions*
I/O instructions	A port instruction data type may be different on the different platforms, as follows: (i) unsigned char* (PowerPC, M68HC11/12, M68K, S390), (ii) unsigned int (ARM) (iii) unsigned long (Itanium, Alfa, SPARC) (iv) unsigned short (80x86).
Interrupt-servicing routines	Interrupt vectors are to be defined differently. (Section 4.4) OS supports these differently on different platforms.
Data types	OS should have appropriate APIs for data types. There may also be need as Linux declares all data types in <asm/ types.h> and it includes in <linux/types.h> as the following: (i) unsigned byte (means 8-bit character also) (ii) unsigned word (means unsigned 16-bit and also unsigned short) (iii) unsigned int (means unsigned 32-bit) (iv) unsigned long (means unsigned 32-bit).
Interface-specific data types	For example, a network interface card supports 32-bit unsigned integers and with a big endian.
Byte order	It may depend on the processor. Lower byte first in an integer (little endian) and the upper byte first in an integer (big-endian). Some processors support both (ARM).
Data alignment	(i) Two or three bytes stored at an address from which the processor accesses 4 bytes in an access. (ii) Same data structure at 'C' source file may show differently on different platforms ('C' takes 16-bit integer on a 16-bit processor and 32-bit integer on a 32-bit processor). Compiler must force the alignment of data by the OS–hardware interface function.
Linked lists	An OS maintains the lists for different data structures. OS provides the standard implementation of doubly linked lists and circular linked lists. Platform-dependent device manager and drivers must include support to these. (Circularly linked means last element of the list linked not to the NULL pointer but to the first element of the list. Doubly linked list means that each element has two pointers, one for the next element and one for the previous element.)
Memory page size	PAGE_SIZE is 4 kB in Linux. A processor may support different page sizes than this.
Time intervals	Linux OS defines the system-clock ticks and interrupts at each 10 ms. The timer functions need to be verified for actual functioning on porting an OS into a platform.

When porting RTOS codes into the system, the porting of I/O instructions,ISRs, data types, interface-specific data types, byte order, data alignment, linked lists, memory page size and time intervals must be taken care of as these are platform-specific. OS–hardware interface functions are needed for these.

13.5.5 Performance and Performance Accelerators

Performance Modeling

(A) System Performance Index: The performance of the finally developed embedded system is a measure of success. Its performance at each life cycle of the development process is tested for the following: each required function must show after the test that its characteristics are in conformity with the required and agreed specifications.

The system performance index can be defined as the ability to meet required functions and specifications while using the minimum amount of resources of memory, power dissipation and devices and minimum design efforts and optimum utilization of each resource (e.g., high CPU load).

The best embedded software and hardware is the one that achieves the balance among different performance metrics.

(B) Multiprocessor System Performance: The multiprocessor system performance is measured by: (i) an optimized partition of the program into the tasks or set of instructions between the various processors, and then (ii) an optimized scheduling of the instructions and data over the available processor times and resources. Performance cost is more if there is idle time left than the available time. Performance matrix is first obtained to calculate the total cost.

(C) MIPs, MFLOPs and DMIPS as Performance Indices: One performance design metric is how long a system takes to execute the desired system functions. Processor clock frequency and MIPs (million instruction per second) and MFLOPs (million floating point instructions per second) are often quoted as design characteristics for expected system performance. It is not however correct metrics. A processor performance design metric is Dhrystone/second. Processing performance is often measured in DMIPS Dhrystone million instruction per second (1 MIPS = 1757 Dhrystone/second) (Section 2.6). EDN Embedded Benchmark Consortium (EEMBC) proposed five-benchmark program suites for: (i) telecommunications, (ii) consumer electronics, (iii) automotive and industrial electronics, (iv) consumer electronics, (v) office automation. It is also used for measuring and comparing embedded system processor performances.

(D) Performance Metrics: Buffer Requirement, IO Performance and Bandwidth Requirement: The buffer helps in accelerating the performance of the system. Memory or I/O buffer requirement may be sometimes a constraint. IO performance is measured by throughput and buffer utilization. Larger bandwidth requirement in client-server systems may be a constraint.

(E) Real-Time Program Performance: Recall Sections 8.10.8 to 8.10.10. Three performance metrics were described: (i) ratio of sum of interrupt latencies as a function of the execution times, (ii) CPU load, (iii) worst case execution time with respect to the mean execution time.

Data communication and multimedia communication have differing performance indices. Loss of any bit needs retransmission. Also no frame or packet miss is tolerable. On the other hand missing frames within acceptable limits are tolerable in video and multimedia systems.

The time of scheduling of a task can be measured by appropriate scope or analyser or by instruction counts or by instruction execution time profiler at the simulators.

Choice of appropriate real-time programming model, partitioning into tasks and scheduling algorithm reflect in the following three metrics as follows:

1. *System throughput*. Comparative performance with respect to the previous life cycle in the development process or previous performance of the system. Relative performance equals relative increase in throughput.
2. *Latency or response time of each task* or ISR (Section 4.6). Both throughput and latency may be unrelated.
3. Delay zitters may be a performance metric instead of response times in some cases. The delays (latencies) between retrievals of the data frames or packets or video-frames can vary. This variation is random or statistically Gaussian distributed and is called delay zitter. The noticeable zitter in delay from the expected variation is undesired. It degrades the system performance. Image zitters may not be tolerable, but delayed retrieval within the acceptable threshold is tolerable.

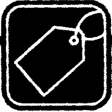
Performance Accelerators There can be several ways to accelerate the performance. Examples of these are as follows.

1. Conversion of CDFGs into DFGs, for example, by using loop flattening (loops are converted to straight program flows) and using look-up tables instead of control condition tests to decide a program flow path.
2. Reusing the used arrays and memory and appropriate variable selection, appropriate memory allocation and de-allocation strategy.
3. Using stacks as data structure when feasible instead of queue and using queue instead of list, whenever feasible.
4. Computing slowest cycle first and examining the possibilities of its speed-up.
5. Code such that more words are fetched from ROM as a byte than the multibyte words.
6. Co-processors and IPs such as Java accelerator accelerate the performance.

Summary

- Host system and software development tools are used in developing, testing and debugging the embedded software in development phase.
- There are a number of software and hardware tools to implement the designed system easily with simple efforts. These are: simulators, editors, compilers, assemblers, source code engineering tool, profiler (for viewing time spent at each function or set of instructions), memory scope, stethoscope-like view of code execution, memory and code coverage scope, emulators, ICEs, oscilloscopes, logic probes, logic analysers and EPROM/EEPROM application codes burner.
- Linker and locator are used for developing the codes for the target hardware. Locator files have Intel hex or Motorola S format. Device programmer is used to burn the binary image of the codes from the locator-created files.
- System implementation and integration is done using program development kit, source code engineering tool and IDE.
- Prototype development tools and IDE are used to develop the fully simulated, tested and debugged sophisticated embedded systems with simpler efforts.
- Selection of right hardware during hardware design and understanding of possibilities and capabilities of hardware during software design is critical especially for a sophisticated embedded system development.
- There are several ways of measuring system performance. It can be a system performance as per the required and agreed specifications, power dissipation, throughputs, IO throughputs, response time of tasks, deadline misses, response to sporadic tasks, memory buffers, bandwidth requirements and memory optimization. Latency intervals and deadline misses are measured to understand the performance of the real-time programming, scheduling models and algorithms.

- Performance index gives the desired performance with respect to the required specifications or parameters.
- Performance accelerators are used to improve the performance. Acceleration means using the same system by alternative ways such that it reduces execution times of a set of codes, reduces latencies of the tasks or increases throughput or minimizes memory usage or power dissipation or reduces missing deadlines. Some ways are loop flattening, look-up tables, reusing the used arrays and memory and appropriate variable selection, appropriate memory allocation and de-allocation strategy and using stacks as data structure when feasible instead of queue and using queue instead of list whenever feasible. We must look at computing slowest cycle first and examining possibilities of its speed-up.
- Choosing the right processor, memory, devices and bus and porting by OS/RTOS the processor-sensitive, memory-sensitive and device-sensitive instruction is a must. Byte order and data alignment must be according to the platform chosen.



Keywords and their Definitions

Action plan : A plan for action of the development process.

Assembler : A tool for assembling the edited codes in mnemonics.

Big endian : An ordering in which the highest byte of a number is taken as first.

Burning : An act of placing the ROM image for code and data in uncompressed or compressed format into an EPROM or EEPROM or flash or microcontroller or some other similar device.

Circular linked list : A data structure for a list in which the last element points to the first element instead of pointing to NULL in a usual list.

Doubly linked list : A data structure for a list in which each element points to the next as well as the previous element in the list in place of pointing to only the next element or NULL in a usual list.

Co-designing : Software team designs with complete knowledge of hardware capabilities and features and hardware team designs with complete knowledge of software CDFGs and functions to be achieved. Certain software functions are implemented by hardware and certain hardware functions are implemented by software with the aim of achieving desired system performance at lowest cost.

Cross-assembler : An assembler that assembles code for host machine for simulation and other purposes and later generates assembled codes for the targeted processor.

Data alignment : Data alignment means alignment in specific way, for example, when (i) two or three bytes stored at an address but the processor accesses 4 bytes during one access in a processor designed for 32-bit per instruction or word and (ii) the same data structure at 'C' source file showing differently on different platforms having different data alignments.

Debugging tools : Tools for debugging embedded system hardware and software functioning.

Delay zitters : The delay zitters mean the variations in the delays in retrievi , or arrival of successive data sets. The noticeable variations are undesired.

Device programmer : A device for burning in the codes (Refer to Section 13.4).

Dissembler : A tool for obtaining higher-level codes from the machine codes, which were assembled earlier.

Edit-test-debug cycle : A cycle in implementation phase in which codes are edited, tested and debugged for reported error on test.

Embedded system project management : Organizing people, processes, product and project. People in embedded system development project means a team of software development, hardware development and system integration engineers.

Host system : A PC or workstation or laptop, which is a computer loaded with software tools and includes the program development kit for a high-level language program or IDE.

Human-machine interactions : Interactions of a user through tools like keypad, display unit and GUIs.

I/O instructions : Processor read, write, byte manipulation and other instructions for using a device at a port.

Platform dependency : A function or ISR or device driver or OS function or data type or data structure utilization, dependent on the processor or memory or devices in the system.

Integrated development environment : Refer to IDE.

IDE : A fully integrated tool consists of simulators with editors, compilers, assemblers, RTOS source code engineering tool, profiler (for viewing time spent at each function or set of instructions), memory scope, stethoscope-like view of code execution, memory and code coverage scope, emulators, logic analysers and EPROM/EEPROM application codes burner.

Little endian : An ordering in which the lowest byte of a number is taken as first.

Networking stack : A stack according to a protocol chosen, for example, protocol RFC -1323, CIDR, IP Multicast, IP, UDP, TCP, DNS client, DHCP server, SMTP server, RIPv1 support, RIPv2 support, ARP, proxy ARP, BOOTP or RLOGINN client and server (for Telnet). An embedded system socket can then connect to multiprotocol LANs, ATM network or SONET or wireless access and intelligent networks using the protocol stacks for the network.

Interpreter : Interpreter does at run-time expression-by-expression (line-by-line) translation to the machine-executable codes.

Latency : Time taken to activate code execution after an event or time taken in finishing certain codes before the next one starts.

Performance index : Index to measure the desired performance with respect to required specifications.

Performance accelerators : Using the same system, alternative ways to *improve* execution time for a set of codes and *reduce* latency or *increase* throughput or *minimize* memory usage or power dissipation.

Performance metrics : Indices for measuring the performance using different measures.

Page : A unit of memory in kilobytes, which can be, referred to as a single block from start address and a memory address in it can be referred by start address plus offset.

Page size : Size of the page taken by memory manager.

Prototyping tools : Tools for developing by co-designing a protype for embedded system.

PLC : A programmable unit to perform sequential logic control functions.

Porting issues : Issues when a software developed at one platform is embedded at another platform.

Right platform : An appropriate hardware platform with appropriate software to give best

		performance at minimum efforts or costs.
Software–hardware tradeoff	:	To appropriately plan and optimize performance at the least cost and choosing which set of processing elements, functions and codes (e.g., VLIWs) are implemented by a hardware subunit and which by a software module.
System cost	:	Cost for hardware and software. It includes all the costs for the development team and management efforts.
System integration	:	Integration of embedded software into the hardware and getting a validated product with optimized performance.
Target system	:	A system for the targeted embedded system that is used during development phase and the final products of software and hardware are made from it.
Test vector	:	A set of statements in the program for controlled flow of programs—path during test phase.
Throughput	:	Number of processes or specified functions executed per unit time. For IO systems, it is the number of bytes outputted or read per unit time.
VHDL and VeriLog	:	Languages for designing and synthesizing the VLSI implementation of a system or a part of the system.

Review Questions

1. Describe functions of compiler, linker, locator, loader, interpreter, dissembler, cross-assembler and integrated development system.
2. Explain functions of device programmer.
3. Why do we use host system for most of the development? What are the software tools needed at the host?
4. What is a target system? How does the target system differ from the final embedded system? What do we mean by application software for a target system?
5. How do the readily-available networking stacks and device drivers at RTOS help in faster error-free design?
6. Why is *system performance index* defined as the ability to meet required functions and specifications while using the minimum amount of resources of memory, power dissipation and devices and minimum design efforts and optimum utilization of each resource (e.g., high CPU load)?
7. Why is the I/O instructions platform dependent? Define throughput of an I/O system.
8. How do the data align? Take the example of 32-bit integer stored as big endian as an example for aligning bytes from an input stream.
9. How do you solve the problem of interface-specific data types?
10. Why is the selection of the right platform essential during the embedded system development process?
11. Explain the software–hardware trade off? What are the advantages and disadvantages of software implementation instead of hardware implementation?
12. What are the advantages and disadvantages of hardware implementation instead of software implementation? What are the advantages of using FPSLIC (filed programmable system logic IC) in an embedded system?
13. Why are the device drivers of the programs memory- and processor-sensitive?
14. What are the factors for selecting a processor during the system design phase?
15. Describe performance-accelerating methods.

Practice Exercises

16. Take a commercial IDE, for example, from Kiel and study its functions, features and capabilities.
17. Explain with one example the use of each of the following: application development tools, native development environment, APIs to RTOS, debugging capability device simulation, network simulation and user interface.
18. Explain with one example the use of each of the following software tools: profiler scope, memory usage scope, stethoscope, scope for trace of program flow, scope for memory allocations and uses and scope for code coverage.
19. Explain hardware–software tradeoff by taking the examples of digital camera and ACC.
20. You can design an SoC by three routes: using gate arrays, using standard cell and using IPs and basic component layouts. List cases of embedded systems for each of these three routes.
21. How does a buffer help in improving a system performance? What is the performance metric for a multiprocessor-based embedded system *router*? When is the minimum interrupt latency taken as embedded system performance metric? (Assume that router that has 10/100 Mbps bandwidth, ethernet interfaces for LANs, Gbps ethernet interface for connection to servers, WAN and internet interface of frame relay, ATM and packet over SONET/SDH.)

Read on-line topic, 'Software Engineering Approach in Embedded System Development Process' and 'Embedded Systems Project Management' at web material accompanying the book and answer the following.

22. What do you mean by embedded system-independent design followed by system integration and by embedded system concurrent hardware–software co-design? Give five examples for each design strategy.
23. What should be the goal during an embedded system development process? How does it vary from the software development process?
24. What is the action plan to follow while designing an embedded system?
25. Who are the *people* involved in an embedded system development project? How will you select them for the case studies of systems described in Chapters 11 and 12? How will the team change when real-time video-processing system is under development?
26. Give system specifications for: (i) product functions and tasks, (ii) delivery time schedule, (iii) product life cycle, (iv) load on system, (v) human–machine interaction, (vi) operating environment, (vii) sensors, (viii) power requirement and environment, (ix) system cost for a digital camera. Camera should be capable of storing 4 minute video or 500 still images. The system should include the USB port, imaging cum video software, single shot timer standard as well as 10 second delay modes. Multiple resolutions are: 1024×768, 640×480, 320×240 and 160×120 pixels. Answer after web search.
27. Explain product design life cycle.
28. What do you mean by system project management?
29. Explain the terms: (i) product functions and tasks, (ii) delivery time schedule, (iii) product life cycle, (iv) load on system, (v) human–machine interaction (e.g., by keypad and display subunits), (vi) operating environment (e.g., temperature and humidity), (vii) sensors, (viii) power requirement and environment, (ix) system cost.
30. Explain meaning of conceptual design.
31. List UML diagrams, which help in developing the conceptual design, structure and layout.
32. Explain two design approaches: independent design and co-design.

Testing, Simulation and Debugging Techniques and Tools

14

Recall

Embedded system hardware and software architecture, programming and design have been learnt in previous chapters. At the stage of porting codes into hardware, there is edit-test-debug cycle, which is repeated till a bug-free code is obtained.

LEARNING OBJECTIVES

*Testing and debugging ensures the system quality. A rule, which the developer must follow is that **wrong until confirmed right by testing and debugging**. Documentation in detail for each stage of testing and debugging is also a necessity. We will learn the following:*

1. *System codes are tested on the host system as host system has application development tools, large memory and windows or powerful GUIs.*
2. *Simulation by a simulator, which runs on host, helps in system development by simulating target processor or microcontroller, peripherals, devices and network interfaces.*
3. *Laboratory tools, in-circuit emulator and monitor help in target system hardware development and target system software testing and debugging in the target environment.*

14.1 TESTING ON HOST MACHINE

We have two systems with different CPUs or microcontroller and hardware architecture. One system is host and the other is the target (Sections 13.2 and 13.4). The host is generally PC or laptop or workstation. Target is actual hardware to be used for embedded system under development.

Testing and debugging have to be there at each stage as well as at the final stage when the modules are put together. Test at initial stages is done on the host machine. Host machine is used to test hardware-independent codes. Host machine is also used to run simulator (Section 14.2). Figure 14.1 shows the test systems in a development process. It shows host and hardware systems, and host-dependent, target-independent and target-dependent code. The code has two parts: hardware-independent and hardware-dependent codes. For example, port and devices will have fixed addresses on hardware.

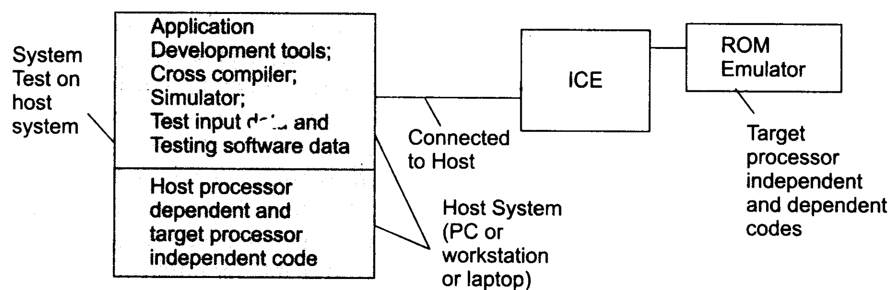


Fig. 14.1 Host and hardware systems and host-dependent, target-independent and target-dependent codes and test systems in a development process

Table 14.1 gives nine steps during testing.

Table 14.1 Testing Steps at Host Machine

Steps	*Action*
1. Initial tests	Test each module or segment at initial stage itself and on host itself.
2. Test data	All possible combinations of data are designed and taken as test data.
3. Exception condition tests	Consider all possible exceptions for the test.
4. Tests–1	Test hardware-independent code.
5. Tests–2	Test scaffold software (scaffold software is software running on the host of the target-dependent codes and which have the same start code and port and device addresses as at the hardware. Instructions are given from file or keyboard inputs. Outputs are at LCD display and saves a file).
6. Test interrupt service routines hardware-independent part	Those sections of interrupt service routines are called, which are hardware-independent and tested (e.g., deciphering the data routine).
7. Test interrupt service routines hardware-dependent part	Those sections of interrupt service routines are called, which are hardware-dependent and tested (e.g., receiving the port data into a buffer).
8. Timer tests	Hardware-dependent code has timing functions and uses a timing device. Timer-related routines such as *clock tick* set, *counts* get, *counts* put, *delay* are tested.
9. *Assert* macro tests	The use of an *assert* macro is an important test technique. For example, consider a command, 'assert (*pPointer* != NULL);'. When the *pPointer* becomes NULL, the program will halt. We insert the codes in the program that check whether a condition or a parameter actually turns true or false. If it turns false, the program stops. We can use the assert macro at different critical places in the application program.

14.2 SIMULATORS

Before flying an aircraft or fighter plane, a pilot uses the flight simulator for training. (A flight simulator may cost hundreds of millions of dollars!)

Simulator uses knowledge of target processor or microcontroller, and target system architecture on the host processor. Simulator first does cross-compilation of the codes and places these into the host system RAM. The behaviour of the target system processor registers is also simulated in RAM. It uses linker and locator to port the cross-compiled codes in RAM and functions like the code that would have run at the actual target system. Host system is a PC or workstation or laptop and generally works in Windows.

Simulator software also simulates hardware units such as emulator, peripherals, network and input–output devices on a host (PC or workstation or laptop). A simulator remains independent of a particular targeted system. It is extremely useful during the development phase for application software for the system that is expected to employ a particular processor or microcontroller or device. The results expected from codes at target system RAM, peripherals, network and input–output devices are obtained at the host system RAM.

A simulator helps in the development of the system before the final target system is ready with only a PC as the tool for development. Simulators are readily available for different processors and processing devices employing embedded systems, and a system designer and/or developer need not code for the simulator for application software and hardware development in the design laboratory. Figure 14.2 shows the detailed design development process using the simulator.

Section 14.2.1 gives the simulator features. Section 14.2.2 gives the possible inabilities of the simulator. Section 14.2.3 describes features of a simulator software VxSim. Section 14.2.4 describes features in the prototype development, testing and debugger tools.

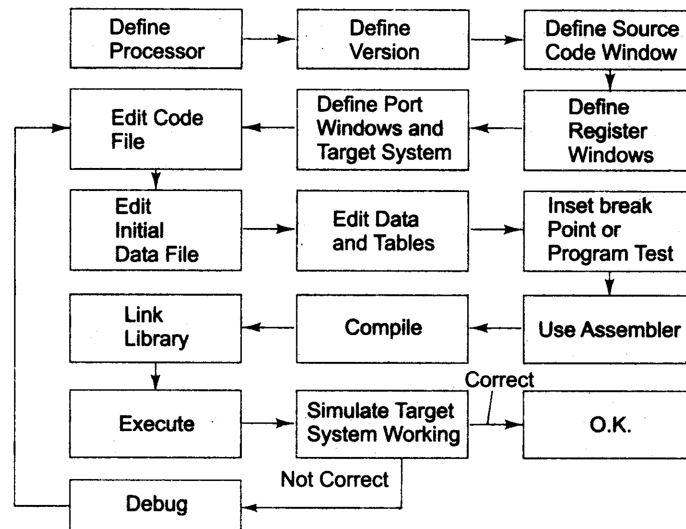


Fig. 14.2 The detailed design development process using the simulator

14.2.1 Simulator Features

A typical simulator is mostly run on a PC Windows environment. A typical simulator includes the following features.

(1) It defines the processor or processing device family as well as its various versions for the target system.

(2) It monitors the detailed information of a source code part with labels and symbolic arguments as the execution goes on for each single step.

(3) It provides the detailed information of the status of RAM and ports (simulated) of the defined target system as the execution goes on for each single step.

(4) It provides the detailed information of the status of peripheral devices (simulated, assumed to be attached) with the defined system.

(5) It provides the detailed information of the registers as the execution goes on for each single step or for each single module. It also monitors system response and determines throughput.

(6) The Windows on the screen provide the following.

  (a) The detailed information of the status of stack, devices and ports (simulated) of the defined microcontroller system.

  (b) Program flow trace as the execution continues. A trace means the output of contents of PC versus the processor registers. It is an important debugging tool of an assembly language program. Trace of application software means an output of chosen variables in a function of stepping sequence. Tracescope gives the time on X-axis and chosen parameter on Y-axis as the program continues further. (TraceScope is a tool module to obtain a trace of the changes in the modules and tasks with time on the X-axis. An action-list also produces with specifications of expected time scales.)

(7) It provide help Windows on the screen. A help Window gives the detailed meaning of the present command pointed by the mice-cursor.
(8) It monitors the detailed information of the simulator commands as these are entered from the keyboard or selected from the menu.
(9) It incorporates the assembler, dissembler, user-defined keystroke or mouse-selected macros, and interpreters for C language expressions, as well as for assembly language mnemonics (expressions). It thus tests the assembly codes. The user-defined keystroke macro is a very useful facility. For example, we can define keystroke 1, say, for providing a particular input byte at a port n and a particular RAM address byte.
(10) It supports the conditions (up to 8 or 16 or 32 conditions) and unconditional breakpoints. There is a feature that halts a program after a definite number of times an instruction executes. *Breakpoints* and *trace* are used in the testing and debugging tool.
(11) It facilitates synchronizing the internal peripherals and delays.
(12) It employs preempting RTOS scheduler support for the high priority tasks.
(13) It simulates the inputs from the interrupts, the timers, ports and peripherals. Hence it tests the codes for these.
(14) It provides network driver and device driver support.

Simulator simulates most functions of a target-embedded system circuit including additional memory, peripherals and buses on the host system itself. It makes application development independent of prior availability of a particular target system. It also simulates the real time processes and shows the outputs on the host system that will be obtained when the codes will actually execute on the targeted particular processor.

14.2.2 Simulator Possible Inabilities

A simulator may not resolve timing issues and hardware-dependent problems. Processor speed at the target processor may not be adequately mapped with the processor speed at the host for calculating time responses and calculating output instances and throughputs at the target.

A simulator may fail to show a bug from the shared data (Section 7.8) as it arises from an interrupt in some particular situation only.

A simulator may not be able to simulate the ASICs and IP(s), which may be embedded at the target system. An ASIC or IP core manufacturer usually provides an alternative debugging tool in that case. For example, ICE for the processor ARM7 or ARM9 (Section 2.3.3) emulates the ARM functions on the host processor and system.

A simulator may not be able to take into account of existence internal devices. For example, the target system may use a Java accelerator, whereas the host system may not have that.

A simulator may not be able to take into account portability problems. For example, target system may have 8-bit data bus between RAM and un-pipelined processor and host have pipelined processor and 32-bit bus.

14.2.3 Simulating tool Software

VxSim. *VxSim* is a simulator tool, which provides a virtual target for developing and debugging the codes. It helps in avoiding the repeated code located in actual target board of the embedded system. Simulating the application with VxSim is of great help in the early development stage, as the VxWorks RTOS task scheduling can be thoroughly simulated before implementation into the target.

Table 14.2 gives the features.

Table 14.2 Features in an Exemplary Simulator VxSim

Supporting Features	*Activities*
Application development tools	It supports the UML and 'RougeWave'. It gives a short design cycle.
Native development environment	It supports several native development environments and debugging. Environment may be MS Visual C++ or GNU tools.
Simulation APIs of the RTOS	Simulates use of many APIs of the RTOS for a given hardware.
Debugging capability	Debugging capability enables fault finding a much easier task.
Device simulation	Simulates devices and device driver behaviour.
Network simulation	Network simulation capabilities make it a virtual test bed, which permits modelling of complex multinode networked systems. For example, a router or gateway. A network application can simulate internal subnet or a real network. When simulating a network, it generates stacks for various standard network protocols that include even the IP multicast and IP broadcast.
User interface simulation	Simulates, for example, a set-up box interface.

14.2.4 Prototype Development, Testing and Debugger Tools for Embedded System

A prototype development tool can be used in place of target system hardware. These tools simulate, compile and debug with a browser. The browser summarizes the final targeted embedded system's complete status during the development phase. Table 14.3 gives the features of a set of prototyping tools from WindRiver.

14.3 LABORATORY TOOLS

14.3.1 Simple Volt-Ohm Meter

Simple Volt-Ohm Meter can be used to test the target hardware. It has two leads marked red and black. One end of each is connected to the meter and the other to points between which the voltage or resistance is to be measured. The meter is set for *Volt* for checking the power supply voltage at source and voltage levels at chips power input pins, and port pins' initial at start and final voltage levels after the software runs. The meter is set for *Ohm* when checking broken connections, improper ground connections and burn out resistances and diodes.

14.3.2 Simple LED Tests and Logic Probe

Let us remember that a digital signal is simply a discrete signal between two voltage ranges. CMOS logic, '1' is the discrete range V_{DD} to 0.66 V_{DD} and '0' is 0.33V_{DD} to V_{DD}, where V_{DD} is usually 5 V with respect to V_{DD} (ground potential). An analog signal varies continuously. A *logic probe* is the simplest hardware test device. It is a handheld pen-like device with LEDs. Its LED glows green, when the probe tip touches at the test point (port or at hardware pin) and the bit there is '1'. It glows red if it is '0'. A logic probe LED blinks fast in a probe version, when a test point is at '1' and does not blink when at '0'. The device at the other end connects by a wire to the ground potential.

Table 14.3 Set of Prototyping Tools from WindRiver®

Tool	*Features*
ScopeProfile	This dynamic execution profiler lets us see, like an oscilloscope waveform, where the CPU is spending its cycles. Performance bottlenecks can then be understood. It shows how much time the processor spends in each function in the task or ISR.
MemScope	Memory usage is a critical aspect of an embedded system. Is there any wasteful use of memory? Is there any memory leak error? Memory leak means that a pointer is incrementing into the unassigned area for a task or stack overflow or writing at the end of an array. MemScope gives the memory block usage. It detects the leak due to system call or another ported module.
StethoScope	Just as a stethoscope helps a doctor in diagnosis, it dynamically tracks the changes in any program variable. It tracks the changes in a parameter. It lets us understand the sequences of multiple threads (tasks) that execute. It records the entire time history.
TraceScope	It helps in tracing the changes with time on the X-axis and an item from the list of actions on Y. TraceScope lets us find the RTOS scheduler behaviour during task switching and notes the times for various RTOS actions.
CodeTest memory, trace and coverage	These tools help in code testing by dynamic memory allocation analysis, controlled flow view trace and code coverage under various real-word situations. The code coverage study helps in removing the extra codes and functions not needed for a specific application. It facilitates development of a scalable system.
VxWorks networking stacks	Another power tool that enhances the code development process. VxWorks RTOS prepares the stack for sending data on the internet to test high-performance switching devices. The stack is according to the protocol chosen. The protocols are the following: RFC-1323, CIDR, IP Multicast, IP, UDP, TCP, DNS client, DHCP server, SMTP server, RIPv1 support, RIPv2 support, ARP, Proxy ARP, BOOTP, RLOGINN client and server (for Telnet). An embedded system socket can then connect to multiprotocol LANs, ATM network or SONET or wireless access and intelligent networks.
VxSim	A powerful simulator tool, which provides a virtual target for developing and debugging the codes. It helps in avoiding the repeated code located in the actual target board of the embedded system. Simulating the application with VxSim is of great help in the early development stage, as the RTOS task scheduling can be thoroughly simulated before implementation into the target.

A logic probe becomes an important tool when studying long delay effects (>1 second) at a port. Its application is as follows. A short program for a delay and then sending the results at the port using logic probe will test the OS timer ticks.

14.3.3 Oscilloscope

Finally, code-downloaded hardware needs testing after completing the edit-test and debug cycle, using a simulator or IDE. An oscilloscope is a scope with a screen to display two signal voltages as a function of time. It displays analog as well as digital signals as a function of time.

We must use it with DC (directly coupled) inputs. Another terminal of the input is always properly kept at ground potential. The term DC should not be confused with the direct current supply. Oscilloscope has two selections for an input, DC and AC. DC means directly coupling the input to the scope input amplifier, AC means input to amplifier (vertical section amplifier) after a capacitor. When using at AC, the signal shape can distort. Its use is only when viewing the signal in a form, in which the signal amplitude = 0 when averaged over time (alternating component of the given signal). (Averaging means a simple average, not root mean

square average.) If the bus signals are viewed with AC selection, a false overshoot or undershoot may show up. Therefore, most of the times, we connect to DC input for observing the waveforms.

A clock, if running will show the states 0 and 1 on the scope. The horizontal gap between the successive rising edges gives us the clock time period. For example, an 8051 using a 12 MHz crystal, there will be states, each of period 0.0825 µs. The check for this and ALE (address latch enable) simultaneously at two input amplifiers will test the processor activity. Another use of scope is in checking the real-time clock routines and pulse width output routine. Real-time software test and debugging are easy using the scopes. The output signal on a serial port or the output bit on a parallel port test will provide significant information. Scope is usable for testing the delay time routines. We can set three delay parameters in three registers and note the interval taken for the port bit to change with each run. From these three measured intervals, we estimate the actual setting in the register for requisite delay. We then run with this delay parameter setting and test, using the scope and fine-tune to the exact delay setting.

An advantage of scope is its use as a noise detection tool and as a voltmeter. Another use is detection of a sudden in-between transition between '0' and '1' states during a clock period. This debugs a bus malfunction. A *storage scope* is another version of oscilloscope. It stores the signals versus time. Later we analyse the stored activity.

14.3.4 Bit Rate Meter

A bit rate meter is a measuring device that finds the numbers of '1's and '0's in the preselected time spans. How to measure the throughput, the number of bytes per second on a network? Assume that 0xA55A (binary 1010010101011010) is sent repeatedly as output bits. The number of 1s multiplied by 16 is the throughput in byte/second. Similarly we can find another bit pattern, and find the expected number of bits in the given time. We can estimate the bits, '1's and '0's in a test message and then use bit rate meter to find whether that matches with the message.

14.3.5 Logic Analyser

After using the simulator, ICE and debug codes in ROM, in the last stage of debugging, we may use a troubleshooting hardware diagnostic tool that records the state (i) as a function of time and (ii) as a function of other states. Logic analyzer can be used in any of these two modes.

Logic analyser is a power tool to collect through multiple input lines (say, 24 or 48) from the buses, ports and records many bus transactions (about 128 or more). It displays these on the monitor (screen) to debug real-time triggering conditions. It helps in sequentially finding the signals as the instructions execute with respect to a reference. One of the bus signal or clock signal is taken as the reference.

A logic analyser can easily debug small-level embedded system. It is a more powerful tool than the scope. Scope views and checks only two signal lines. A logic analyser is a powerful software tool for checking multiple lines carrying the address data and control bits and the clock. The analyser recognizes only discrete voltage conditions, '1' and '0'.

In the *first* mode, the analyser collects the logic states as a function of time and stores these in memory and displays on screen. It tracks the multiple signals simultaneously and successively. There are multiple input lines (24 or 48 or more). We connect the lines from the system and IO buses, ports and peripherals. It collects simultaneously for the duration of the many bus transactions (about 128 or more). It later displays, using this tool, each transaction on each of these on the computer monitor (screen). It also prints the displayed results. The phase differences in each input line also give important clues. It debugs the real-time triggering conditions. It helps in finding the bus signals and port signal status sequentially as the instructions are executed. A variant of the logic analyser also provides the analog measurement when needed.

In the *second* mode, buses are connected to logic analyzer probe pins and the analyser gives the captured states of all the signals at the clock edge. The triggering point for capturing the states can be defined by the user. The triggering point can be defined as *observation* of an illegal op-code or processor at particular startup address or a certain port byte at output.

For example, the analyzer is set to measure at first, second, third, fourth and so on clock edges, up to 64 or 128 or any number (say 2^{20}) clock edges from a start address 0x10000. The analyser gives the address and data bus states in hexadecimal and it gives each control signal state. An advanced version of logic analyzer can also trace the instruction sequences from the observed address and data bus states at the clock edges from the given start address. A software engineer can trace illegal instruction or protected address accesses, when running the codes.

Certain bugs that intermittently arise can also be recorded with a logic analyser by continuous and repeated runs of the system.

Logic Analyser Inabilities A logic analyser does not help on a *program halt due to a bug*. It does not show the processor register and memory contents. If the processor uses the caches, bus examination alone may not help. We cannot modify the memory contents or input parameters during trace and display as we do in a simulator. The effects of these changes are invisible.

With SOC use in embedded systems design, the inner-connections are just not visible to the logic analyzer.

14.3.6 In-Circuit Emulator (ICE)

Instead of the target system that is copied to obtain an embedded system, can we have a separate unit that remains independent of a particular targeted system processor or microcontroller? Yes.

We use a *target* or ICE. Instead of a target circuit, an ICE provides a greater flexibility and ease for developing various applications on a single system instead of testing multiple targeted systems. Figure 14.3(a) and (b) shows emulator and ICE, respectively.

ICE is a circuit for emulating the target system that remains independent of a particular targeted system processor, usable during the development phase for most of the target systems that will incorporate a particular microcontroller chip. It works independently as well as by connecting to the PC through a serial link. It is a target circuit minus target microprocessor or microcontroller.

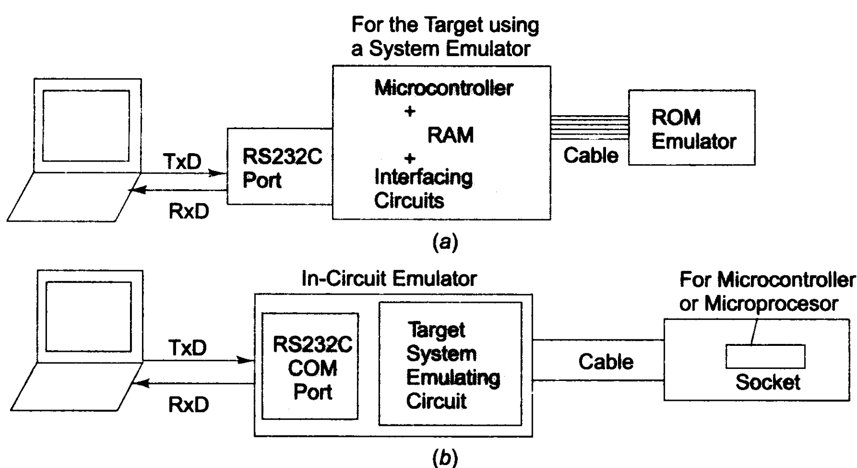


Fig. 14.3 (*a*) An emulator (*b*) An in-circuit emulator

ICE is an emulator of the microprocessor of target circuit, such that a host system connects to the ICE through a serial link for debugging purposes. ICE emulates various versions of a microcontroller family during development phase using the remaining part of the target circuit.

ICE is an emulator of microprocessor or microcontroller of target circuit in a target emulating circuit.

How does an ICE differ from target? The target uses the circuit consisting of the microcontroller or processor itself. The emulator emulates the target system with extended memory and with codes-downloading ability during the edit-test-debug cycles. ICE emulates the processor or microcontroller. It uses another circuit with a card that connects to the target processor (or circuit) through a socket.

The back support hardware package and ICE have the subunits listed and explained in Table 14.4.

Table 14.4 Back Support Hardware Package and In-Circuit Emulator (ICE) Subunits

Emulator subunits[1]	*Action(s)*
Interface circuit	It is for downloading ROM images into EPROM and RAM bytes from the host system into the emulator. It uses a serial (COM RS232C) port of PC (Figure 14.3). It helps in embedding in the program memory part the large application codes directly from the PC. Codes may be developed on the host using a high-level language. For example, the development of the application codes' designer finds it much more easy to write the large application programs instead of keying them in machine codes using 20 keys pad at the emulator.
Socket	A multipin male–female socket to insert a general-purpose processor or DSP or embedded processor or microcontroller, which connects to the ICE through a cable (usually a ribbon cable) and connectors. (See Figure 14.3(*b*), right corner socket).
External memory	*Additional RAM and EPROM or EEPROM*, enough for use by most possible targeted systems and their applications.
Emulator-board display unit	*A single-line 8- or 12-character display.* It is to show the content of memory addresses one by one. Also, it is to show the contents of registers at the various program steps.
Twenty-keys pad	It is to *enter data and codes directly* by the user locally at the memory addresses. These codes have to be machine codes.
Registers	Additional *system registers* for the single step as well as full speed test runs during testing of the system.
Connectors	To plug-in this emulator to the interface circuits and other devices and peripherals that are typical to the system. A connector for the target system display module is an example. Another example is for the PC interface circuit.
Target system keyboard	Keyboard user input board equivalent to the target system expected keyboard.
Target system driver circuit	For example, *driver* hardware for network or motor or solenoid valve or furnace or printer.
Monitor codes	These are in the emulator EPROM or EEROM or at the target system ROM to test and debug with actual target processor or microcontroller and target circuit.

[1] Emulators from Orion Instruments, USA embed the logic analyser-like facility (Section 14.3.4). Intel provides the emulators and back support packages for its different processors and microcontrollers.

ICE consists of the following: (i) An emulator pod with a ribbon cable, which extends to a processor or microcontroller socket of the target system [Figure 14.3(b)]. We later on insert the processor IC in that socket.

Later on, we can test the target system also. (Remember that this developed and debugged system is a one-to-one copy for the final, embedded system.) To avoid coupling capacitive effect due to a long cable, we must use as short a cable as possible. The pod circuit emulates the target system microcontroller or processor. (ii) The pod links serially to COM RS232C port of a computer. Through this port, the pod gets the downloaded codes from the computer. The computer program for the emulator monitors completely the bytes at the registers and memory locations. The pod may have some card between its basic circuit and ribbon cable jumper. The replacement of this card makes it feasible to use the ICE for another version of a processor or microcontroller family.

What about processor core itself accompanying the ICE core? A feature in ARM7 and 9 processors (Section 2.3.3) is that these processors have accompanying ICE subunit. It helps in debugging the targeted hardware.

ICE or emulator disables after the development phase is complete. An actual circuit forms just by copying the codes developed using the ICE. This circuit after the interconnection to the target processor consists of the used processor, required memory chips and keys and display units or other peripherals. This should work exactly the same and as perfectly as at the end of the development phase that we completed using the emulator or ICE. An emulator helps in the development of the system before the final target system is ready.

Motorola provides M68HC11EVM and M68HCEVB as the emulators for 68HC11 microcontroller-based target system. These emulators have the following external connections.

When using an ICE or emulator, software required for implementation phase are the editors, assemblers, dissembler, simulators and so on (Section 12.4). The host system is just for down-loading the codes to the emulator and for echoing back the codes and data at the various addresses in the emulator memory. A designer needs the host system to save machine-level programming time that can be too much for the sophisticated applications. We can have the additional socket connectors for the different versions of the microcontroller: for example, for emulating a 48-pin version as well as a 52-pin version of the 68HC11.

An ICE 'visionICE I' is an ICE that has the networking capabilities. The latter imbibes by 10/100 Mbps ethernet connectivity. This lets the ICE accessible to a LAN. Remote debugging is another advantage. It also connects to the serial port of the target system.

A *ROM emulator* [Figure 14.3(a), right side] emulates only a ROM. The target connects through a ROM socket and also connects to the computer. There is a need during the edit-test-debug cycle for downloading the codes into the target system flash or EEPROM cyclically. Th0e ROM emulator obviates this need. Monitor (Section 14.3.7) codes can be downloaded in ICE ROM. It may run a 'Power On Self Test' (POST) program on bootstrapping. The embedded system when coupled to the RS232C COM port or network port of a computer can use *gdb*, a GNU debugger (it is a downloadable freeware).

14.3.7 Monitor

Monitor is a debugging tool for actual target microprocessor or microcontroller in ICE ROM emulator or in target development board. It also lets host system debugging interface just like an ICE. Monitors from different sources differ in their functioning. One typical monitor does the following.

1. Monitor loads the application codes, is also used for corrections in codes and then to test the system. A command for download can download a new application code into the monitor. A command for resetting the program restarts the program. Monitor loads the application (in hex file) from the developing system (at host) that can also be modified later to correct the codes.
2. A part of the monitor runs on host system. Debug monitor codes are downloaded along with the locator binary image. A write and a read command is used to correct or examine the codes at the memory addresses of the system. Monitor controls (as per command from debugger) the execution of application at full speed, as well as by single stepping during debug phase.

3. Monitor controls (inserts, removes, modifies) breakpoints as per command from the debugger. A breakpoint partitions the program into separate segments. When a program segment runs, there is a pause at breakpoint and then test the result is observed after the run and is examined; then, the segment is run. Breakpoints enable program test running between the different program segments.
4. Monitor can be run in single step mode also.
5. Monitor facilitates controlled execution of application and controlled display of executing program status.

Table 14.5 lists the target board units with monitor and monitor segments.

Table 14.5 Target Board Subunits Including Monitor

Target Board Subunits	*Action(s)*
Socket	It is for downloading monitor and ROM images into EPROM and RAM bytes from the host system into the target. It uses a serial (COM RS232C) port of the host system. Codes may be developed on the host using a high-level language.
RAM and interfaces	RAM and interfaces.
Display	Display subunit displays the application codes (as per command from debugger) in full or in segments, the registers and internal RAM or memory addresses data during debugging phase running through the single stepping or breakpoints.
Monitor segments in ROM	Interface commands for interfacing with the host system; command interpreter; loaded application codes; and data.
Twenty-keys pad	It is to enter data and commands of monitor and corrects the codes directly by the user locally at the memory addresses. These codes have to be machine codes.
Connectors	Connectors for display subunit and printer.

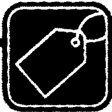
Monitor means a ROM resident program at the target board or ROM emulator connected to ICE. It monitors the device applications, the runs for different hardware architecture and is used for debugging.

Summary

The following is a summary of what we had discussed in this chapter.

- System codes are tested on the host system as host system has application development tools, large memory and windows or powerful GUIs. Each module must be tested at the initial stage of its development as well as by integrating all modules. Software can be tested on host machine. It is divided into two parts: hardware (target)-independent code and hardware-dependent code. Hardware-dependent code has fixed start addresses, fixed port and device register and other addresses.
- Simulation by a simulator, which runs on host, helps in system development by simulating target processor or microcontroller, peripherals, devices and network interfaces. Instruction set of target processor or microcontroller simulates on the host in a simulator.
- Volt Ohm meter is useful for checking the power supply voltage at source and voltage levels at chips power input pins, and port pins initial at start and final voltage levels after the software runs, checking broken connections, improper ground connections and burnout resistances and diodes.

- Oscilloscope is use to test the fast changing signals, their wave forms, overshoots and undershoots at transitions.
- Logic analyser measurs logic states on many connections simultaneously. It has two modes of functioning. One mode is to show time on X-axis, and logic states of the clock signal, bus signals and other signals on Y-axis. Second mode is to give address, data bus and other signal states from a trigger point to examine illegal op-codes, access in protected address space and other states as a function of a reference state.
- ICE is used for debugging a target system without using the target processor microcontroller.
- ASIC and SoC system hardware cannot be tested by laboratory tools, such as logic analyser and ICE.
- Monitor is used to debug software and hardware for the given target processor or microcontroller.



Keywords and their Definitions

Host system : PC or workstation or laptop on which an application development is done for a target system.

ICE : An emulator of microprocessor of target circuit, such that a host system connects to the ICE through a serial link for debugging purposes and emulating various versions of a microcontroller family during development phase using the remaining part of the target circuit.

Logic analyser : A power tool to collect through its multiple input lines (say, 24 or 48) from the buses, ports and many bus transactions (about 128 or more) to display these on the monitor (screen) to debug real-time triggering conditions.

Monitor : Codes placed in the emulator EPROM or EEROM or at the target system ROM to test and debug with actual target processor or microcontroller-based circuit.

Oscilloscope : A scope with a screen to display two signal voltages as a function of time. It displays analog as well as digital signals as a function of time.

Simulator : *Software*, which runs on host in powerful GUIs environment, helps in system development by simulating target processor or microcontroller, peripherals, devices and network interfaces.

Target system : A system which has hardware similar to that of the final product and on which the embedded software has to run.

Volt-Ohm meter : A meter to measure voltage and resistance between two points to test voltage levels at supply rails, broken connection, resistances and diodes.

Review Questions

1. Why is host system used for most stages of development and test and simulation?
2. Give examples of hardware-dependent and hardware-independent codes.
3. What is a target system? How does the target system differ from the final embedded system?
4. What do we mean by application software for a target system?
5. What is back support package? What are the various components of a target emulator? What are the advantages of using an ICE?

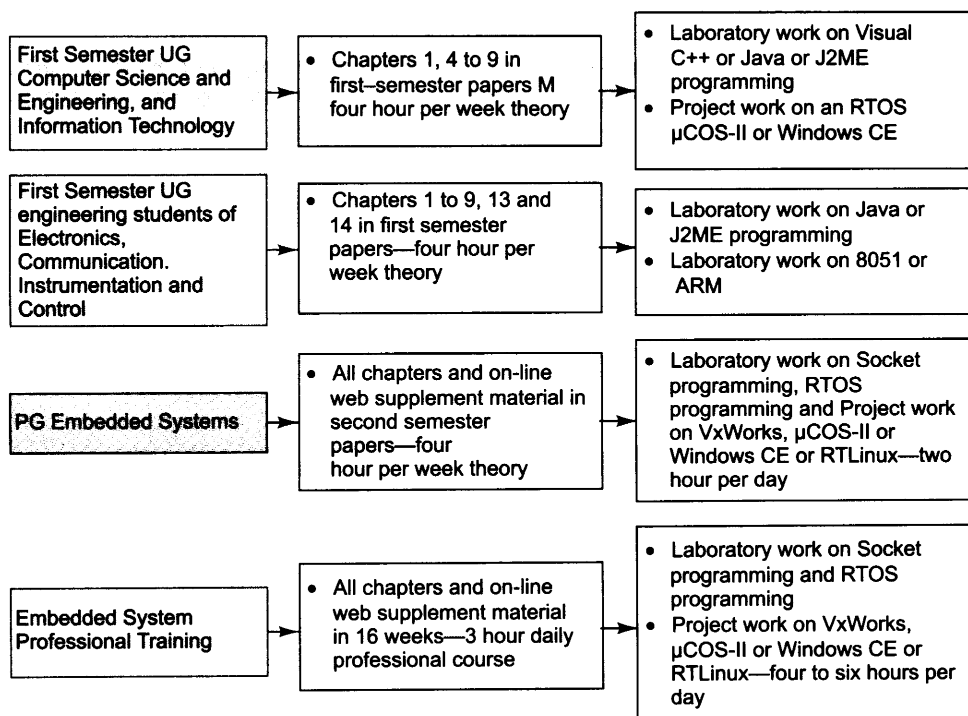
6. Explain the use of the following hardware tools: target emulator and ICE.
7. What is the use of a simulator in a development phase?
8. How does a calling of interrupt routine help in testing a design?
9. What is a cross-assembler?
10. What is time mode of a logic analyzer? What is *state mode* of a logic analyzer?
11. What do we mean by a logic analyzer? What is the use of a logic analyser during the development phase?
12. A LED circuit is also a powerful analysis tool. How is it so?
13. What are the uses of an oscilloscope?
14. How will you use a bit rate meter to measure throughput from a real-time system?
15. Why is the I/O instructions platform dependent? Define throughput of an I/O system.

Practice Exercises

16. Which are the popular simulators used?
17. Prepare a list of emulator systems available for various microprocessors, microcontrollers and DSPs.
18. Explain with one example the use of each of the following: debugging capability, device simulation, network simulation and user interface.
19. Explain with one example the use of each of the following software tools: profiler scope, memory usage scope, stethoscope and scope for trace of program flow, scope for memory allocations and uses and scope for code coverage.
20. List prototyping tools with a popular RTOS.

Appendix 1: Roadmap for Various Course Studies

Learned professors and syllabi designers are the best judges. From the author's experience, the roadmap shown in the following figure can be adapted by various disciplines of UG, PG and professional training courses.



Suggested Roadmap for Various Disciplines of UG, PG and Professional Training Courses

Appendix 2: Select Bibliography

A. PRINTED BOOKS

1. Graham Phillips, Bill Pierce and John Hardin, "*Linux Appliance Design: A Hands-On Guide to Building Linux Appliances*", BS Starch Press, 2007.
2. Grzegorz Rozenberg, and Frits Vaandrager (Eds.) "*Lectures on Embedded Systems: European Educational Forum School on Embedded Systems, Veldhoven*", Springer, Nov. 2006.
3. Michael Barr and Anthony Massa, "*Programming Embedded Systems: With C and GNU Development Tools*", 2nd Edition, O,Reilly, Oct. 2006.
4. Nicolas Carter, and Raj Kamal (adoption author), "*Computer Architecture*", Schaum Series TMH Edition, May, 2006.
5. Peter Marwedel, "*Embedded System Design*" – Springer Verlag, New York 2006.
6. Raghavan P., Amol Lad, and Sriram Neelakandan, "*Embedded Linux System Design and Development*", Auerbach Publications, Taylor and Francis, Dec. 2005.
7. Bruno Buoyssounouse and Joseph Sifakis, "*Embedded Systems Design: The Artist Roadmap for Research and Development*", Springer, 2005.
8. Tammy Noergaard, "*Embedded Systems Architecture: A Comprehensive Guide for Engineers and Programmers*", Newnes, Butter-worth Heinemann, Newton, Mass. USA, 2005.
9. Raj Kamal, "*Microcontrollers- Architecture, Programming, Interfacing and System Design*", Pearson Education, Singapore, 2005.
10. Jack Ganssle (Ed.), "*The Firmware Handbook*", Newnes, Butter-worth Heinemann, Newton, Mass. USA, 2004.
11. Jack Ganssle and Michael Barr, "*The Embedded Systems Dictionary*", CMP Books, 2003.
12. Prasad K. V. K. K., "*Embedded Real Time Systems: Concepts, Design and Programming – The Ultimate Reference*" Dreamtech, 2003.
13. Douglas Boling "*Programming Microsoft WINDOWS CE.NET*", Microsoft, USA, 2003.
14. John Catsoulis, "*Designing Embedded Hardware*", 2nd Edition, O'Reilly, 2003.
15. Prasad K. V. K. K., Vikas Gupta, Avinash Dass, Ankur Verma, "*Programming for Embedded Systems—Cracking the Code*", Wiley, New Delhi, 2002.
16. Jonathan W. Valvano, "*Embedded Microcomputer Systems- Real Time Interfacing*", Thomson, Brooks/Cole, 2002.
17. Stephen Palmer and John Felsing, "*A Practical Guide to Feature-Driven Development*", Prentice Hall, 2002.
18. Stuart R. Ball, *Embedded Microprocessor Systems: Real World Design*, Butter-worth Heinemann, Newton, Mass. USA, 1996. (2nd Edition, May 2002).
19. Phillip A. Laplante, *Real-Time Systems Design and Analysis* – An Engineer's Handbook, 2nd Edition, IEE Press, USA, 1997 (Prentice Hall of India, Third Indian Reprint, April, 2002).

20. Raj Kamal, *Internet and Web Technologies*, Tata McGraw-Hill, 2002.
21. Bob Zeidman, *Designing with FPGAs and CPLDs*, CMP Books, Sept. 2002.
22. Demuth B. and D. Eisenreich, *Designing Embedded Internet Devices*, Butterworth Heinemann, July 2002.
23. Al Williams, *Embedded Internet Design*, McGraw Hill, July 2002.
24. Miro Samek, *Practical StateCharts in C/C++—Quantum Programming for Embedded Systems*, CMP Books, July, 2002.
25. Tim Jones M., *TCP/IP Applications Layer Protocols for Embedded Systems*, Charles River Media, June 2002.
26. Steve Heath, *Embedded System Design: Real World Design*, Butterworth Heinemann, Newton, Mass. USA, May 2002.
27. Michael J. Pont, *Embedded C*, Addison Wesley, April 2002.
28. Lewis D., *Fundamentals of Embedded Software: Where C and Assembly Meet*, Prentice Hall, Feb. 2002.
29. Dreamtech Software Team, *Programming for Embedded Systems—Cracking the Code*, Hungry Minds, April 2002.
30. Craig Hollabaugh, *Embedded Linux Hardware and Software*, Addison Wesley, March 2002.
31. Macii, Benini and Poncino, *Modern Design Technologies for Low Energy Embedded Systems*, Kluwer Academic Publishers, March 2002.
32. George Pajari, *Unix Device Drivers*, Pearson Education, Indian Reprint, 2002.
33. Ed Sutter, *Embedded System Firmware Demystified* (with CD), CMP Books, Feb. 2002.
34. Frank Vahid and Tony Givargis, *Embedded System—A unified Hardware/ Software Introduction*, John Wiley and Sons, Inc. 2002.
35. Steve B. Farber, *ARM System-on-Chip Architecture*, 2nd Edition, Addison Wesley & Benjamin Cummings, 2002.
36. Wayne Wolf, *Modern VLSI: System on Chip Design* Pearson, Jan. 2002.
37. Jim Ledin, *Simulation Engineering- Build Better Embedded Systems faster*, CMP Books, Aug. 2001.
38. Todd D. Morton, *Embedded Microcontrollers*, Prentice Hall, New Jersey USA 2001.
39. Adam Drozdek, *Data Structures and Algorithms in C++*, Brooks/Cole Thomson Learning, 2001.
40. Joseph Lemieux, *Programming in the OSEK/VDX Environment*, CMP Books, Oct. 2001.
41. Thomas D. Burd and Robert W. Brodersen, *Energy Efficient Microprocessor Design* Kluwer Academic Publishers, Oct. 2001.
42. Eric Giguere, *Java 2 Micro Edition-The ultimate Guide to Programming Handheld and Embedded Devices*, John Wiley, USA, Canada 2000.
43. John Uffenbeck , *The 80x86 Family*, 3rd Ed., Pearson Education India, 2002.
44. Ali Mazidi M. and J.G. Mazidi, *The 8051 Microcontroller and Embedded Systems*, Pearson Education, 2000, First Indian Reprint, 2002.
45. Jeremy Bentham, *TCP/IP Lean Web Servers for Embedded Systems*, CMP Books, USA 2000. (Also 2nd Edition, 2002).
46. Sundrajan Sriram, and Survra S. Bhattacharya, *Embedded Multiprocessors- Scheduling and Synchronization*, Marcel Dekker, Inc., NewYork, USA 2000.
47. Raj Kamal, *The Concepts and Features of Microcontrollers (68HC11, 8051 and 8096) -Includes Programmable Logic Controllers*, S. Chand & Co. (Originally Wheeler Pubs.), New Delhi, 2000.
48. Gary Nutt, *Operating Systems—A Modern Perspective*, Addison Wesley Longman, Inc., USA, 2000 (Pearson Education Asia Singapore, India Reprint 2000).
49. Steve White, *Digital Signal Processing*, Thomson Learning – Delmar, 2000 (First Indian Reprint, Vikas Publishing House, 2002).
50. Filip Thoen and Francky Catthoor, *Modeling, Verification and Exploration of Task-Level Concurrency in Real-Time Embedded Systems*, Kluwer Academic Publishers 2000.
51. Sommerville, *Software Engineering*, Addison Wesley, Reading, MA, USA, 2000.
52. Rainer Laeupers, *Code Optimization Techniques for Embedded Processors: Methods, Algorithms and Tools*, Kluwer Academic Publishers, Oct. 2000.
53. William A. Shay, *Understanding Data Communications and Networks*, 2nd Edition, Thomson Learning – Brooks/ Cole, 1999 (First Indian Reprint, Vikas Publishing House, 2001).

54. Randall S. Janka, *Specification and Design Methodology for Real-Time Embedded Systems*, CMP Books, Nov. 2001.
55. Scott Rixner, *Stream Processor Architecture* Kluwer Academic Publishers, Nov. 2001.
56. Tim Wilmshurst, *An Introduction to the Design of Small Scale Embedded Systems - with examples from PIC, 8051, and 68HC05/08 Microcontrollers*, Palgrave, Great Britain, 2001.
57. Pfleeger S. L., *Software Engineering Theory and Practices*, Pearson Education, USA Singapore, India Reprint 2001.
58. Rogers S. Pressman, *Software Engineering*, 20th Edition, McGraw-Hill, 2001.
59. Arnold S. Berger, *Embedded Systems Design—An Introduction to Processes, Tools and Techniques*, CMP Books, Nov. 2001.
60. Kirk Zurell, *C Programming for Embedded Systems*, CMP Books, Feb. 2002.
61. Wayne Wolf, *Computers as Components—Principles of Embedded Computing System Design*, Academic Press (A Harcourt Science and Technology Company), USA, 2001.
62. Jack Ganssle, *"The Art of Designing Embedded Systems"* (Edn Series for Design Engineers), Newnes, Butter-worth Heinemann, Newton, Mass. USA, 2000.
63. Jane W.S. Liu, *Real Time Systems*, Pearson Education, 2000 (First Indian Reprint 2001).
64. Joseph L. Weber, *Using Java™ 2 Platform*, Que Corporation, Reprint by Prentice Hall of India, New Delhi, May 2000.
65. Jack W. Crenshaw, *Math Toolkit for Real-Time Programming*, CMP Books, Aug. 2000.
66. David E. Simon, *An Embedded Software Primer*, Addison Wesley Longman, Inc., USA, (Pearson Education Asia) Singapore, USA 1999 (India Reprint 2000).
67. Barry Kauler, *Flow Design for Embedded Systems—A Simple Unified Object Oriented Methodology*, CMP Books, Feb. 1999.
68. Franz J. Rammig (Ed.), *Distributed and Parallel Embedded Systems*, Kluwer Academic Publishers, Netherlands, 1999.
69. Alessandro Rubini, *Linux Device Drivers*, O'Reilly, USA, June 1999.
70. Luis Miguel Silveira, Srinivas Devadas, Ricardo A. Reis, *VLSI: Systems on a Chip*, Kluwer Academic Publishers, Dec. 1999.
71. John Hyde, *USB Design by Example*, John Wiley & Sons, Inc., New York, 1999.
72. Jean J. Labrosse, *Embedded Systems Building Blocks*, 2nd Edition, CMP Books, Dec. 1999.
73. Jack G. Ganssle, *Art of Programming Embedded Systems*, Butter-worth Heinemann, Newton, Mass., USA, 1999.
74. Michael Barr, *Programming Embedded Systems in C and C++*, O'Reilly, USA Aug. 1999 Reprinted Shroff Pubs. India Reprint August 1999.
75. Myke Predko, *Programming and Customizing the 8051 Microcontroller*, McGraw-Hill, 1999, Third Reprint Tata McGraw-Hill, 2002.
76. Jean J. Labrosse, *MicroC/OS-II The Real Time Kernel*, R&D Books, an Imprint of Miller Freeman, Inc. Lawrence, KS 66046, USA, 1999. (Also 2nd Edition in 2002 from CMP Books).
77. Bruce Powel Douglass, *Real-Time UML—Developing Efficient Objects for the Embedded Systems*, Addison Wesley Object Technology Series, 1998.
78. Calcutt M.C., F.J. Cowan, and G.H.Parchizadeh, *8051 Microcontrollers—Hardware, Software and Applications*, Arnold (and also by John Wiley), 1998.
79. Rick Grehan, Robert Moote and Ingo Cyliax, *Real-Time Programming—A guide to 32-bit Embedded Development*, Addison Wesley, 1998.
80. John A. Stankovic, Marco Spuri, Krithi Ramamritham and Giorgio C. Buttazzo, *Deadline Scheduling for Real-Time Systems—EDF and Related Algorithms*, Kluwer Academic Publishers, Netherlands, Oct. 1998.
81. Stuart R. Ball, *Debugging Embedded Microprocessor Systems*, Butter-worth Heinemann, Newton, Mass. USA, 1998.
82. Niall Murphy, *Front Panel—Designing Software for Embedded User Interface*, CMP Books, June 1998.
83. M.Costanzo, *Programmable Logic Controllers—The Industrial Computers*, Arnold (and also John Wiley) 1997.
84. Cady F. M., *Software and Hardware Engineering—Motorola M68HC11*, Oxford University Press, 1997.

85. Cady F. M., *Microcontrollers and Microcomputers—Principles of Software and Hardware Engineering*, Oxford University Press, New York, 1997.
86. Balarin F., M. Cliodo, A. Jurecska, H. Hsieh, A. L. Lavagno, C. Paasserone, A. E. Sangiovanni- Vincentelli, E. Sentovich, K. Suzuki, and B. Tabbara, *Hardware-Software Co-Design of Embedded Systems: A Polis Approach* Norwell, MA, Kluwer Academic Publishers, June 1997.
87. John Forrest Brown, *Embedded System Programming in C and Assembly*, Van Nostrand, Reinhold, New York, USA, 1996.
88. Peter Spasov, *Microcontroller Technology- The 68HC11*, 2nd Edition, Prentice Hall, Englewood Cliffs, NJ, 1996.
89. Fred Halsall, *Data Communication, Computer Networks and Open Systems*, 4th Edition, Pearson Education, 1996 (Fourth Indian Reprint, 2001).
90. Silberschatz and P.B.Galvin, *Operating Systems*, Addison Wesley, Reading, MA, USA, 1996.
91. Peter Marwedel, and Gerl Gossens, *Code Generation for Embedded Processors*, Kluwer Academic Publishers, June, 1995.
92. Daniel Tabak, *Advanced Microprocessors*, McGraw-Hill, USA 1995.
93. Gajski, Daniel D., Frank Vahid, Sanjiv Narayan and Jie Gong, *Specification and Design of Embedded Systems*, Englewood Cliffs, NJ, Prentice Hall, 1994.
94. Franklin G. F., J. D. Powell and A. Emami-Naeini, *Feedback Control of Dynamic Systems*, 3rd Ed., Addison Wesley, Reading, MA, USA, 1994.
95. Stewart J. W., *The 8051 Microcontroller—Hardware, Software and Interfacing*, Prentice Hall, 1993.
96. Walter J. Grantham and Thomas L. Vincent, *Modern Control Systems—Analysis and Design*, John Wiley, 1993.
97. Hintz K. J. and Daniel Tabak, *Microcontrollers—Architecture, Implementation and Programming*, McGraw-Hill, 1992.
98. Jack G. Ganssle, *Art of Programming Embedded Systems* Academic USA, 1992.
99. Greenfield G. D., *The 68HC11 Microcontroller*, Saunders College Publishing, 1991.
100. Peatman J. B., *Design with Microcontrollers and Microcomputers*, McGraw-Hill, 1988.

B. WEBSITE REFERENCES

1. http://www.dspvillage.ti.com [For Texas Instruments DSP Processor, Section 1.2.4, 2.3.6].
2. http://www.mentorg.com/seamless [For Section 1.6].
3. http://www.ti.com/sc/docs/asic/modules/arm7.htm and arm9.htm [For Section 2.3.3].
4. http://www.arm.com [Section 2.3.3, For ARM Processors].
5. http://www.ti.com/sc/docs/psheets/abstract/apps/spra638a.htm [For Section 2.3.6].
6. http://www.eembc.org [For benchmarking of performances of embedded Systems, Sections 2.6 and 13.5.3].
7. http:// www.java.sun.com/ products/ javacard [For Section 5.7.5].
8. http://www.webopedia.com/TERM/N/operating_system.htm [For Section 8.1].
9. http://www.wrs.com [For Section 9.3].
10. http://www.osek-vdx.org [For Section 10.2].
11. http://www.linuxdoc.org [For Section 10.3].
12. http://www.cs.ucr.edu/esd [For Computer Sciences Embedded System Design website of University of California, Riverside, Section 11.2].
13. *http://www.ee.surrey.ac.uk/Personal/R.Young/java/html/cruise.html* {For Section 11.3].
14. http://www.borg.com/~jglatt/tutr/miditutr.htm [For tutorial on MIDI, Section 12.1].
15. 15.http://www.xtec.es/rtee/eng/tutorial/midi.htm [For MIDI interface, Section 12.1].
16. http://www.misra.org.uk [For Section 12.2 and for Guidelines for the Use of the C Language in Vehicle Based Software of MISRA (Motor Industry Software Reliability Association)].
17. http://www.research.ibm.com/securesystems/scard.htm [For Section 12.4].
18. http://www.home.hkstar.com/~alanchan/papers/smartCardSecurity/ [For Section 12.4].