Application Design for Helio

Designing Your Application

The first stage of writing an application involves designing screen layout (Figure 1.1), user interface (UI) and resource file. This chapter will introduce these in two parts:

- Designing Screen Layout and User Interface
- Making Resource File

Designing Screen Layout and User Interface

Screen Layout

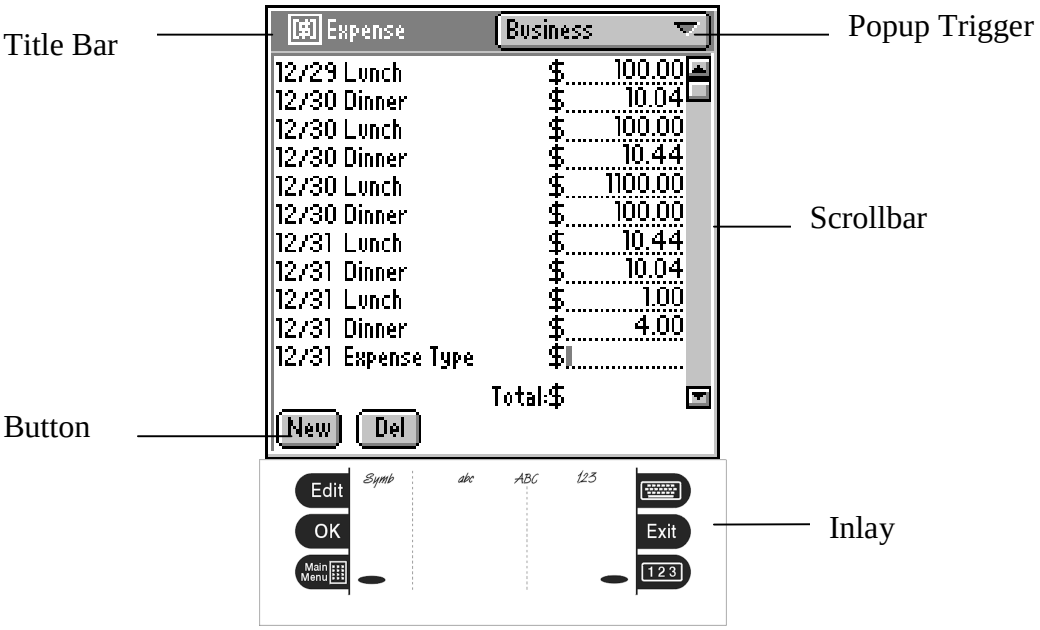


Figure 1.1 The Screen Layout

This section describes the rules of designing the screen layout.

- The objects within a Form should not overlap each other.

- The buttons should be aligned with the bottom edge of the screen.
- There should not be too many objects on the screen at the same time.

User Interface

To create the User Interface, you should first define the UI elements used in your application and create a resource file which defines the characteristics of the UI elements. The components of user interface are objects which are identified by their object ID's and object types in the application. The UI objects are added to the application by providing their object ID's as the value of certain field in the resource file. The details of making a resource file will be described in next section Making Resource File.This section describes the UI elements used in your application.

The following is a brief introduction to the UI objects. More detailed description of each of the objects is shown in Descriptions of UI Objects.

- ***Form***
  Form is a container-typed object which contains other UI objects. It is used to view the application. An application can have many forms.

- ***Bitmap***
  There are two kinds of bitmap which are predefined bitmap and button bitmap.

- ***Control***
  Control object includes button, repeat button, push button, check box and popup trigger.

- ***Field***
  Field object allows user to input editable text. It can support multiple lines of text.

- ***Line***
  Lines drawn on the screen are line objects. Application can define the length, thickness and color of a line.

- ***List***
  List provides a list of items for selection. The items always stay on the screen.

- ***Menu***
  Menu bar will popup a list of application commands when triggered. After user clicks on one of the application commands, the application command is selected and the popup menu will disappear.

- ***Scheduler Line***
  Scheduler line is a feature provided in the Scheduler application. It is only used in the Scheduler Application to show the week view.

- ***Scroll Bar***
  Scroll bar allows user to scroll up and down to view the items beyond the screen bounds. It applies to List, Field and Table.

- ***String***
  String is used to display a non-editable predefined text string. It doesn't respond to any pen action.

- ***Table***
  Large number of text strings and numbers can be aligned and displayed in table format.

- ***Text Box***
  Similar to Field object, Text Box allows user to input editable text. However Text Box only supports single line of text. The length of the text is virtually unlimited.

Making Resource File

If you're planning to write an application in VT - OS, designing the screen layout of your application must be the first step. This section shows how to create a resource text file and how to convert this text file to a C file.

- Making resource text file through an example
- Converting a resource file to a C file

Making resource text file through an example

Suppose an application only consists of a bitmap on the screen, then its resource file includes two UI objects: form and bitmap. Listing 1.1 shows the resource text file of this application as an example. When you write a resource file, several main points you should bear in mind:

- `#XXX` indicates the following resource details describe a certain UI object while `#END_XXX` indicates the end of the object description where `XXX` represents the name of a UI object.
- Within the resource details of a UI object, it divides into certain fields. Each field starts with an indicator, `~Fx`, where `x` is the field number.
- The bracket letters represent different data type (See Table 1.1).
- Each UI object inside the resource file has an object ID which acts as an identity within an application. So the object ID cannot be used repeatedly.
- Spacing is not allowed within the resource file.
- Different UI objects describe their resource details in a different format. The total numbers of fields are also different. See the chapter Resource File Format for more information.
- At the end of the resource file, remember to press enter if you use window's editor; otherwise the resource file cannot be compiled.

Table 1.1 Interpretation of the bracket letters

Bracket letters	Data types
B	BYTE
W	WORD
UW	UWORD
SH	SHORT
U	USHORT
S	STRING
BO	BOOLEAN
P	BITMAP FILE

Listing 1.1 Test.txt (Resource file)

```
#FORM
NAME=FORM_TEST
ID=0
~F0
(B)OBJECT.TYPE=FORM
~F1
(SH)BOUNDS.X=0
(SH)BOUNDS.Y=0
(SH)BOUNDS.WIDTH=160
(SH)BOUNDS.HEIGHT=160
~F2
(U)FOCUSED_OBJECT=1
~F3
(B)FORM_STYLE=NORMAL
~F4
(S)FORM_TITLE=Test Application
~F5
(SH)FORM_BITMAP.X=0
(SH)FORM_BITMAP.Y=0
(SH)FORM_BITMAP.WIDTH=11
(SH)FORM_BITMAP.HEIGHT=12
(P)FORM_BITMAP.FILE=Q_TWO_BIT,TEST.BMP
~F6
(U)NO_OF_OBJECTS=1
~F7
(U)OBJECT_ID=1
(B)OBJECT_TYPE=BITMAP
#END_FORM
#BITMAP
NAME=BITMAP_TEST
ID=1
```

```
~F0
(B)OBJECT.TYPE=BITMAP
~F1
(U)RELATED_TABLE_ID=65535
~F2
(SH)BOUNDS.X=10
(SH)BOUNDS.Y=10
(SH)BOUNDS.WIDTH=20
(SH)BOUNDS.HEIGHT=20
~F3
(B)BITMAP_STYLE=BITMAP_STYLE_0
~F4
(SH)BITMAP1_BITMAP.X=10
(SH)BITMAP1_BITMAP.Y=10
(SH)BITMAP1_BITMAP.WIDTH=20
(SH)BITMAP1_BITMAP.HEIGHT=20
(P)BITMAP1_BITMAP.FILE=Q_FOUR_BIT,TEST.BMP
~F5
(SH)BITMAP2_BITMAP.X=10
(SH)BITMAP2_BITMAP.Y=10
(SH)BITMAP2_BITMAP.WIDTH=20
(SH)BITMAP2_BITMAP.HEIGHT=20
(P)BITMAP2_BITMAP.FILE=Q_FOUR_BIT,TEST01.BMP
~F6
(BO)BITMAP_ATTR.BITMAP_ENABLE=TRUE
(BO)BITMAP_ATTR.BITMAP_ACTIVE=FALSE
(BO)BITMAP_ATTR.BITMAP_VISIBLE=TRUE
#END_BITMAP
```

Converting a resource file to a C file

In view of programming, only a resource text file is useless. So compiling a resource file to a C file is essential. Such process is composed of two intermediate steps. First, the resource file is compiled to generate a binary file. Second, this binary file is converted to a C file. Both intermediate steps operate with two specific executed files: `rcompile.exe` and `bin2hex.exe`. Conversion operates as follow:

- Switch to DOS prompt if you are using windows.
- Under the folder which consists of the two execute files and also the resource file, you can type:

  ```
  rcompile [resource file name]
  ```

  The name of the generated binary file is `resource.bin`.
- Then you can enter:

  ```
  bin2hex resource.bin [output C file name]
  ```

Developing Your Application

VT - OS is a single-task and event driven operation system. Only one application can run at a time. Each application is composed of several forms which contain certain UI objects to display. Before you start to write an application, you should realize the operation of Helio system and then you can structure your own application.

- Each application has a `Main` function which is inside `main.c` file. System sends a **launch code** to start an application. The concept of launch codes will be discussed thoroughly later in this chapter. Note that the purpose of `Main` function is to receive launch codes and call `AppLaunch` function to handle them. This function exists in `ApLaunch.c` file.
- Since VT - OS is event driven, an event loop exists in `main.c` file to get events and then pass to event handler. Each application contains several forms. Different forms should have their own event handler function. The Chapter "Application Event Loop" discusses in details.
- VT - OS application is stopped when it receives `EVT_APP_STOP` event. Later in this chapter, you will learn how to stop an application.

This chapter discusses the following topics:

- Launching an application
- Handling Launch Codes
- Stopping an application
- Application programming concept
- Launch codes

Launching an application

Launch codes are communication tools between VT - OS and applications. Basically, each application can perform a normal launch which loads an application wholly and displays all its user interface. For example, when user selects an application icon from the main menu screen or presses any shortcut buttons on the Helio, system then generates the launch code `LAUNCH_CMD_NORMAL_LAUNCH`, which tells the application to have a complete and normal launch.

However, in some cases, applications do not need to have a complete launch. System may require the application to perform some actions rather than to display its user interface. A good example is the global find. System generates the launch code `LAUNCH_CMD_FIND`, which tells each application to search its database but no need to

show user interface. It is wasteful to have a complete launch of each application for just searching database. So other launch codes can avoid wasting resources.

Each launch code is composed of two types of information:

- Command codes (`cmd`), such as `LAUNCH_CMD_NORMAL_LAUNCH`, provide launching information for application to perform required action.
- Command pointer (cmd_ptr) provides extra information for the application. Such information is optional. For example, the global find action needs to provide the string text for application to search its database.

NOTE: Some launches require the command code only and do not need any further information. In the structure of these launch codes, their command pointer may be useless. Thus the application does not need to free the command pointer of these launch codes when handling them. In other words, if the command pointer of a launch code is used to provide extra information, the application requires to free this command pointer when handling it. In the following section, You will learn how to handle launch code. For the structure of the launch code, please refer to the chapter "Details of launch codes". Also a complete list of other launch codes is shown in the section "Launch codes summary".

Handling Launch Codes

When an application first receives a launch code in `Main` function, it passes the launch code to the `AppLaunch` function, which handle the launch code for this application. For instance, the Phonebook application has a `PhonebookAppLaunch` function for handling launch codes. Listing 2.1 shows the `Main` function of Phonebook application as an example.

Inside the `AppLaunch` function, it first checks whether it needs to handle it. For example, if the calculator application receives the launch code `LAUNCH_CMD_FIND`, it should exit without doing anything because no database exists in this application. Later in this chapter will teach:

- Handling Normal Launch
- Handling Other Launch Codes

Listing 2.1 Main function of Phonebook application

```
__main(WORD cmd, void *cmd_ptr)
{
    PhonebookAppLaunch(cmd, (void*)cmd_ptr);
}
```

Handling Normal Launch

An application performs a normal launch only when it receives the launch code LAUNCH_CMD_NORMAL_LAUNCH. After receiving this code, the application begins to initialize the UI parameters to memory, and also tries to restore the application status if necessary. Then it goes to an event loop. To stop an application, an EVT_APP_STOP event should be received to break the event loop. Finally the application saves the parameters of the application if any changes occur and then it unloads all its UI objects from memory before leaving. Note that the application does not need to free the command pointer of the launch code LAUNCH_CMD_NORMAL_LAUNCH before termination of the application. Listing 2.2 shows the codes to handle a normal launch of Phonebook Application.

Note that not all application requires to restore application status and to save application parameters before starting and stopping an application respectively. For example, if a game does not allow user to continue his game after he leaves, in this case both application restore and application save are not necessary. Note also that the Data Manager in VT - OS helps to restore and save application status. Please refer to the Chapter "Data Manager" for more details.

Listing 2.2 Codes for handling a normal launch in Phonebook Application

```
case LAUNCH_CMD_NORMAL_LAUNCH:
     UIApplicationInit();
     PhonebookAppRestore(FALSE);
     EventLoop();
     PhonebookAppSave();
     UIDeleteAllAppObjects();
     DataCloseDB(pb_dbid);
     return TRUE;
```

Handling Other Launch Codes

Launch codes other than LAUNCH_CMD_NORMAL_LAUNCH bring an application to launch differently. Basically, the application should decide whether it handles the launch code or not. As aforementioned, if a calculator application receives the launch code

LAUNCH_CMD_FIND, it should decide not to handle it and then free the command pointer before exit. This launch code provides extra information for the application and thus freeing the command pointer is necessary. If you want to know which launch codes can provide more information, please refer to the chapter "Details of launch codes". Listing 2.3 shows the source codes for handling launch codes of Calculator application. A complete list of launch codes is provided later in this chapter in the "Launch codes".

Listing 2.3 CalculatorAppLaunch in Calculator application

```
BOOLEAN CalculatorAppLaunch(WORD cmd, void *cmd_ptr)
{
     switch(cmd)
     {
     case LAUNCH_CMD_NORMAL_LAUNCH:
          UIApplicationInit();
          MathsAppRestore(FALSE);
          EventLoop();
          MathsAppSave();
          UIDeleteAllAppObjects();
          return TRUE;

     // Calculator does not handle global find
     case LAUNCH_CMD_FIND:
          pfree(cmd_ptr);
          return TRUE;
     case LAUNCH_CMD_GOTO_REC:
          if (((GotoRec*)cmd_ptr)->find_string)
          pfree(((GotoRec*)cmd_ptr)->find_string);
          pfree(cmd_ptr);
          return TRUE;
     case LAUNCH_CMD_ALARM_HIT:
          pfree(cmd_ptr);
          return TRUE;
     default:
          return FALSE;
     }
     return FALSE;
}
```

Stopping an application

An application stops when it receives an `EVT_APP_STOP` event. The event loop detects this and then terminates the application. You will know more about the event loop and events in next chapter.

System cleans up when an application stops. The process includes closing the database and saving application parameters for restoring through Data Manager. Application first saves application parameters if necessary for restoring next time. Then it deletes all UI objects of the application from UI links. Finally system closes the database of the application if any. Note that each application can have its own application save function.

Application programming concept

The previous sections bring you the idea of how to develop an application. This section will discuss about how to write an application programmatically. Each application may consist of several forms and each form is handled individually by its own event handler function. You can learn more in the section "ApplicationHandleEvent" of next chapter. All these application event handlers are inside `App.c` file.

Each application should have an `AppLaunch` function to handle launch codes. For example, Phonebook application has its `PhonebookAppLaunch` function while Calculator application has its `CalculatorAppLaunch` function. No matter which application, its `AppLaunch` function is inside `AppLaunch.c` file. Moreover, `main` function, which passes the received launch code to the `AppLaunch` function, is essential to each application. The main function is inside `main.c` file.

If an application needs to store its status before it terminates, the pair functions, `AppSave` and `AppRestore`, should help. The former is called after the event loop. Its function is to save the data in the application when it is called to stop. The latter is called before the event loop. Its function is to restore the previously saved data of the application before the application starts. Note that not all applications need these pair functions, such as Calculator application. Listing 2.2 and Listing 2.3 show the source codes of Phonebook application and Calculator application respectively. If you compare the codes of these two applications which handle normal launch, you will find `PhonebookAppRestore` and `PhonebookAppSave` are called before and after the event loop in Phonebook application but no these pair functions exist in Calculator application. The pair functions are inside AppSave.c file. Note that when you define anything in an application, you can define them inside `App.h` file. A typical example is to define the object ID of an application.

Launch codes summary

Table 3.1 displays all the launch codes in VT - OS. These launch codes are defined inside `ALaunch.h` file.

Table 3.1 VT - OS launch codes

Launch code	
`LAUNCH_CMD_NORMAL_LAUNCH`	Launch application normally.
`LAUNCH_CMD_FIND`	Use in global find. Find a text string.
`LAUNCH_CMD_GOTO_REC`	Go to a particular record and display it with string highlighted or without string highlighted optionally.
`LAUNCH_CMD_ALARM_HIT`	`AlarmManager` sends this launch code to the application to tell the alarm is hit.
`LAUNCH_CMD_SYNC_START`	Launch VSync application. Use when the button in cradle is pressed.
`LAUNCH_CMD_VOXMEMO_LAUNCH`	Launch VoiceMemo application. Use when the button at the back of the Helio is pressed.
`LAUNCH_CMD_CAL_PEN`	Launch to calibrate pen in System setup application. Use when the Helio is hard reset.
`LAUNCH_CMD_MODEM_CONNECT`	Launch Connect application in email application.

NOTE: Application may not need to handle all the launch codes because some launch codes are so specific for only one application. Obviously, the last four launch codes in Table 3.1, they are used to launch a specific application. For the launch code `LAUNCH_CMD_ALARM_HIT`, it is used for the application which has alarm setting.

Application Event Loop

Brief Overview

All VT OS applications are event-driven and single-tasking i.e. only one Form is active each time. As described in the section Launching an application, an application will perform a complete launch and display its user interface when it receives the `LAUNCH_CMD_NORMAL_LAUNCH`. It starts with a startup routine, goes into an event loop (which will be discussed later in this chapter) and then exits with a stop routine finally. Figure 3.1 shows the control flow in an application.

Events are commands generated by the system in respond to certain user input action. They are handled by the appropriate event handlers. Event handler functions are arranged in hierarchy. When an event handler handles an event, all the other event handler functions, which are arranged below it, are skipped. Event Manager is the main interface between the OS software and the application. It creates the queue and provides functions for appending events onto the queue. It is written in the file `EVENTMGR.cpp`. For further details of the event manager, please refer to the Chapter "Event Manager".

The source code of the Event Loop is shown in Listing 3.1. In the event loop, the application keeps on checking the events in the event queue. If there are no events in the event queue, the application remains in the EvtGetEvt until there is event and it quits the event loop when it receives `EVT_APP_STOP` sent by the system through the event queue. If there are events in the event queue, the application fetches the first event from the queue and dispatches it. The event is passed onto the system by calling the SystemHandleEvent function. If the event is not a system-handled event, it will not be handled by SystemHandleEvent, the application will call MenuHandleEvent to handle it. If it is not handled by MenuHandleEvent, the application will call the ApplicationHandleEvent to handle it. If it is not handled by ApplicationHandleEvent, the event will be handled by the FormDispatchEvent which will point to the event handler of a specific Form of the application. In this Chapter, we will discuss each of these one by one:

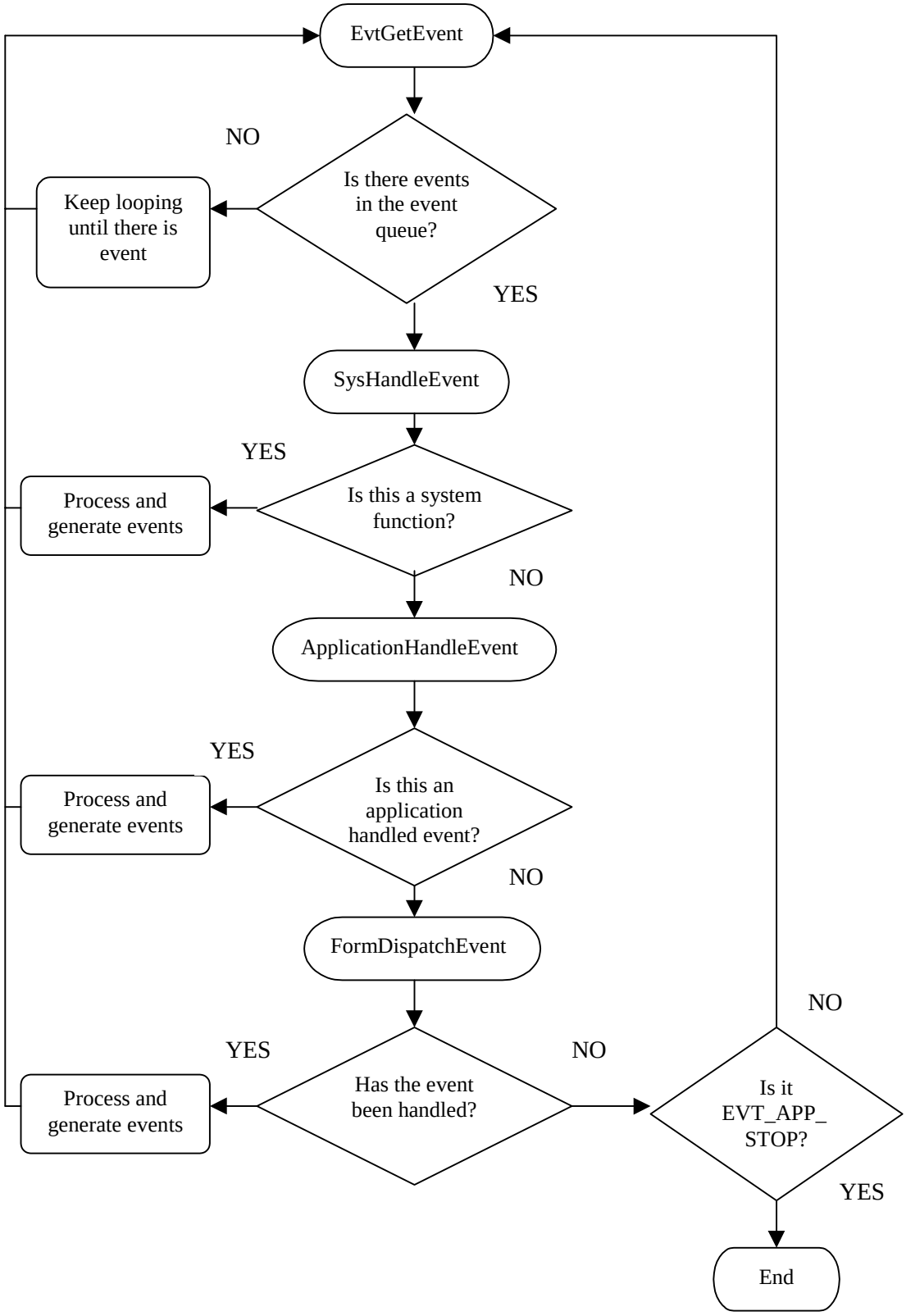
Listing 3.1 The Event Loop

```
void EventLoop(void)
{
     EvtType   event;

     do
     {
       EvtGetEvent (&event);
```

```
          if (!SystemHandleEvent(&event))
            if (!MenuHandleEvent(&event))
               if (!ApplicationHandleEvent(&event))
                    FormDispatchEvent(&event);
      }
      while (event.eventType != EVT_APP_STOP);
}
```

Figure 3.1 The Control Flow in an Application



SystemHandleEvent

The system handles events like tapping inlay icons, tapping the popup keyboard on the screen and tapping objects in a Form such as buttons. It mainly takes care of user input and generates other events for further action. It returns true if these system events are handled. Listing 3.2 shows the SystemHandleEvent function. When an event is passed to it, only one of the four functions will receive it, for example:

- Inlay Events include activating Main Menu, Inlay Keyboard, Inlay Menu, OK, Exit, shortcut to Global Find and Calculator.

- KeyboardHandleEvent handles the event that has occurred when tapping the keys on the popup keyboard.

- FormHandleEvent handles the events that have occurred in the Form. These include any pen events, selecting objects in the Form like Menu, Scrollbar, Table, Schedule Line, Bitmap…etc. The subsequent events are handled by their corresponding Event Handlers like MenuHandleEvent, ScrollbarHandleEvent…etc.

- JotHandleEvent handles events for handwriting recognition.

They are arranged in a hierarchy, when an event is handled by a function, it will not be handled by the other which is lower in the hierarchy.

Listing 3.2 The SystemHandleEvent function

```
BOOLEAN SystemHandleEvent(EvtType *Event)
{
     if (InlayHandleEvent(Event)) return TRUE;
     if (KeyboardHandleEvent(Event)) return TRUE;
     if (FormHandleEvent(Event)) return TRUE;
     if (JotHandleEvent(Event)) return TRUE;
     return FALSE;
}
```

MenuHandleEvent

MenuHandleEvent allows the system to handle the events that has occurred on the menu object. These include any pen events, menu enter event, menu select event and menu exit event.

ApplicationHandleEvent

If MenuHandleEvent did not handle the event, the application will call the ApplicationHandleEvent to handle it. ApplicationHandleEvent only handles the `EVT_FORM_LOAD` event. It performs the function of loading a form into the memory and switching the event handlers of the application to a particular form event handler. This is done by FormInitForm, FormSetEventHandler and FormSetActiveForm functions. Listing 3.3 shows the template of the ApplicationHandleEvent of the Phonebook application. As the event is not handled by the system, it is passed to FormDispatchEvent and according to the form to which the application switch, the corresponding procedures are then taken. Therefore in Listing 3.3, only the switch cases need to be modified when writing an application.

- In the FormInitForm function, the resources of the form object defined in the resource file is loaded to a specific location in the memory. The application can then access the form object by pointing to the corresponding location in the memory.

- There are many forms in one application. The developer needs to clarify which Form will receive the user's action. The FormSetEventHandler is called to set the event handler routine for a specified form. It maps the event handler functions for that form to the function pointer FormDispatchEvent. Therefore all events not handled by the system will go to FormDispatchEvent.

- The FormSetActiveForm function sets the specific form to be active. This is important because there is more than one form in an application, event handler may be mismatched to an unwanted form and the events in the correct form will not be handled correctly. By the FormSetActiveForm, the system knows which form is active and all the functions of the event handler will be addressed to that form. The event will be handled by the event handler of that active form instead.

Listing 3.3 The ApplicationHandleEvent function

```
static BOOLEAN ApplicationHandleEvent(EvtType *Event)
{
     Form *form_ptr;
     ObjectID form_id;
     BYTE object_type;
     Err Error;

if (Event->eventType == EVT_FORM_LOAD)
     {
```

```
form_id = (ObjectID)Event->eventID;
Error=UISearchForAddress(form_id,&object_type,(void**)&form
_ptr);
if (Event->para1 == 1 || Error != TRUE)
{
     FormInitForm(form_id);
}
if(!UISearchForAddress(form_id,&object_type,(void**)&form_p
tr)) return FALSE;

switch (form_id)
{
case FORM_TEL_LIST:
FormSetEventHandler(FORM_TEL_LIST,(void**)&FormDispatchEven
t, (void*)PhonebookTelList);
break;

case FORM_EDIT_CATE:
FormSetEventHandler(FORM_EDIT_CATE,(void**)&FormDispatchEve
nt, (void*)PhonebookEditCate);
break;

case FORM_INPUT_CATE:
FormSetEventHandler(FORM_INPUT_CATE,(void**)&FormDispatchEv
ent, (void*)PhonebookInputCate);
break;
          :
          :

}
FormSetActiveForm(form_id);
return TRUE;
     }
     return FALSE;
}
```

Form ID of the form

Event handler for the form with the form ID FORM_TEL_LIST

FormDispatchEvent

If an event is not handled by the SystemHandleEvent, the MenuHandleEvent and the ApplicationHandleEvent, the event will go to FormDispatchEvent finally. This is a boolean function which consists of one line:

```
BOOLEAN (*FormDispatchEvent)(EvtType *Event);
```

The system maps the event handler of a form to the function pointer FormDispatchEvent. After mapping, the unhandled event can be passed to the mapped FormDispatchEvent and the corresponding active form can process the event. In the example in Listing 3.4, the `EVT_MENU_SELECT_ITEM` is handled by the event handler PhonebookInputCate after the event handler has been mapped to the FormDispatchEvent pointer using the FormSetEventHandler function. Then

Let's take one of the Phonebook event handlers in Listing 3.4 as an example. If the user taps on the Exit button on an Input Categories form, the system appends the `EVT_INLAY_SELECT` event to the event queue. As the event is not handled by the system, it is passed to FormDispatchEvent and according to the switch case, the corresponding procedures are taken.

Listing 3.4 The event handler for the Input Categories form in Phonebook

```
BOOLEAN PhonebookInputCate(EvtType *Event)
{
     ObjectID tempID;

switch (Event->eventType)
{
case EVT_MENU_SELECT_ITEM:
MenuItemSelectedAction((USHORT)(Event->para1));
return TRUE;

case EVT_INLAY_SELECT:
     switch (Event->para1)
     {
     case INLAY_OK:
     if (InputCatCheckCharIn() == TRUE) return TRUE;
     break;

     case INLAY_EXIT:
     FormPopupForm(FORM_EDIT_CATE);
     return TRUE;

     default: return FALSE;
     }
     break;

case EVT_KEY:
     if (Event->eventID == SOFT_KEY)
     {
```

```
     FormGetActiveObject(FORM_INPUT_CATE,&tempID);
     if (tempID == TEXTBOX_INPUT_CATE)
     {
     if ((BYTE)(Event->para1) == 13 &&
     (InputCatCheckCharIn() == TRUE)) return TRUE;
     else TextboxAddKeyInChar(TEXTBOX_INPUT_CATE,
     (BYTE)(Event->para1));
     return TRUE;
     }
     }
     break;

case EVT_BITMAP_SELECT:
PhonebookEditCateSetScrollbar();
FormPopupForm(FORM_EDIT_CATE);
return TRUE;

case EVT_FORM_OPEN:
FormDrawForm(FORM_INPUT_CATE);
return TRUE;
default:
return FALSE;
     }
return FALSE;
}
```