e-SIM MMI Solution

Integration Guide

e-SIM MMI Solution 2.0

Integration Guide

e-SIM Ltd.
POB 45002
Jerusalem 91450
Israel

Tel: 972-2-5870770
Fax: 972-2-5870773

Written and produced by e-SIM Ltd.
MMI_Plat_Int-2.0

Contents

Chapter 1: Overview

The e-SIM MMI Reference Application (referred to as the "MMI Application") was developed to be easily integrated into any platform. This guide provides the information that you, the embedded system integrator, need to integrate the MMI Application into your product's platform.

In order to adapt the MMI Application to your product's platform, you must:

- Prepare your platform for the integration.
- Integrate the RapidPLUS state machine API.
- Integrate the basic MMI Application API (e.g., keypad and display).
- Integrate the MMI Application with the Protocol Stack.
- Integrate the MMI Application with the basic hardware services.
- Test the integrated application on the target.

Requirements

Before you begin the integration process, please ensure that you have the necessary knowledge and software, and that you have prepared the MMI Application task.

Knowledge

RapidPLUS: You should be familiar with RapidPLUS code generation and the embedded RapidPLUS APIs. For assistance, refer to the *Generating Code for Embedded Systems* manual and the online *MMI Component Reference Guide*.

Note: We recommend that you take the RapidPLUS Basics, RapidPLUS Code Generation, and e-SIM MMI Solution training courses before you begin.

Platform: You should be familiar with your platform environment, including API, and with compilation and debugging tools.

Software

- MMI Application in XML format.
- RapidPLUS CODE version that is appropriate for the MMI Application version.
- RapidPLUS embedded kernel and graphic libraries compiled for your platform (see Appendix A for the Porting order).

- e-SIM Customization Tools (i.e., the Multimedia Resource Manager, Menu and Text Generator, and Layout Editor) or their resulting data.
- Compilers, linkers, debugging tools, and other tools necessary for your platform.

Preparations

You must prepare a stand-alone task for the MMI Application. This task must have an incoming queue for messages from other tasks. The MMI task must include an initialization part and an infinite loop for message processing.

You must ensure that necessary information is sent from other tasks to the MMI task.

Integration Overview

The process of integrating the MMI Application into your platform can be divided into four main stages. At each integration stage, you should test and debug your work.

The four stages are:

STAGE	DESCRIPTION
1. Integrate the RapidPLUS state machine API	You will (i) implement the Timer request API, Dynamic Allocation API, Debug API, and user error callback function; and (ii) call the RapidPLUS initialization process.
2. Integrate the basic MMI Application API	You will (i) integrate the platform keypad with the MMI Application's keypad; and (ii) integrate the graphic display object (GDO) with the platform LCD or replace the RapidPLUS graphic services with those of the platform.
3. Integrate the MMI Application with the Protocol Stack	You will integrate the MMI Application by using AT commands or by modifying the MMI service layer to fit your API.
4. Integrate the MMI Application with the basic hardware services	You will integrate the MMI Application with services such as battery, file system, tones, melodies, and backlight.

After integration, you should test the integrated application on the target using your normal testing procedures.

Chapter 2: State Machine Integration

In order to integrate the RapidPLUS state machine API, you will need to prepare a set of callback functions, and then provide these callback functions to RapidPLUS in the initialization process.

Preparing the Callback Functions

For details about all of these functions, refer to "The Application Programming Interface" chapter in the *Generating Code for Embedded Systems* manual.

Timer Request API

RapidPLUS relies on the embedded system to receive timer services; therefore, you must implement two callback functions using the embedded system's timers.

- `usrTimerReqFunc`
  This function will be called by the RapidPLUS kernel each time a RapidPLUS timer object/timer tick object is started. The embedded system must call `rpd_TimerExpired` when the timer has elapsed.
- `usr_TimerStopFunc`
  This function will be called by the RapidPLUS kernel when a timer object/timer tick object is stopped before it has expired.

Dynamic Allocation API

The MMI Application dynamically allocates components in order to optimize RAM memory consumption and the RapidPLUS kernel dynamically allocates internal data structures; therefore, you must implement two callback functions using the embedded system's heap allocation mechanisms.

- `usr_MallocFunc`
  This function allocates a block of memory and returns the pointer to the block.
- `usr_FreeFunc`
  This function receives a pointer to a previously allocated block of memory annd releases it.

User Error Callback Function

RapidPLUS checks error conditions (such as array and string overflow) and internal errors that can be caused by memory corruption. RapidPLUS will call this callback function any time RapidPLUS detects these error conditions. Therefore, you **MUST** implement this callback function in such a way that you will receive a clear notification each time an error occurs.

- `usr_ErrorFunc`
  The parameters received by this function are the error type and error severity.

Debug API

This function is optional. If you register it, you will be able to follow the execution of the MMI Application. We highly recommend using it in order to follow the transitions among modes.

- `usr_DebugFunc`
  This function has many options that can be activated; the most useful option is `fDBGTTransitionsDetail`.

Initializing the RapidPLUS State Machine

In order to initialize the state machine and the MMI Application, you must call the following API in the initialization part of the MMI task. **Be sure to pass the corresponding callback functions that were previously implemented.**

➡ **To initialize the state machine:**

1. Register the timer callback functions by calling:
   ```
   rpd_setTimerRequest(usrTimerReqFunc, usr_TimerStopFunc, 0);
   ```
2. Initialize the dynamic allocation and the error callback functions by calling:
   ```
   rpd_PrivInitMallocTask(usr_ErrorFunc, usr_MallocFunc, usr_FreeFunc);
   ```
3. Initialize the debug API by calling:
   ```
   rpd_SetUserDebug(usr_DebugFunc,debugBuff,DEBUG_BUFF_SIZE,
   fDBGTTransitions);
   ```
   Be sure to statically allocate `debugBuff as an array of integers with DEBUG_BUFF_SIZE` size. Refer to "Using the Debug API" in the *Generating Code of Embedded Systems* manual.
4. Initialize the GDL and GDO. See "Integrating the Graphic Display with the Platform LCD" on p. 8.
5. Call `rpd_PrivStart();`. This function will execute the first cycle of the state machine, executing the entry activities for the root mode of all allocated user objects (UDOs).
6. Call `runRapidCycles(100);`. This function will start the initialization proces of the MMI Application. See "Executing State Machine Cycles" on p. 5.

Executing State Machine Cycles

The execution of the MMI Application requires that the RapidPLUS state machine perform cycles. The basic mechanism includes feeding the state machine with events or data (using the generated API) and processing the data by calling `rpd_PrivRunIdle`. (Information about feeding the state machine is discussed in the other integration chapters; here we are concerned with executing the state machine cycles.)

In order to improve the performance of the state machine and the MMI Application, you should implement the following mechanism:

```
void runRapidCycles(int maxCycles)
{
        int rpd_moreToDo, i=0;
        do{
      rpd_moreToDo = rpd_PrivRunIdle();
         }while(rpd_moreToDo && i++<MAXCYCLES);

  if (rpd_moreToDo)
  {
     /* Normally most actions are dealt with within a few state machine
cycles. However the occasional situation arises when we have run the
maximum number of cycles but there is still more to do. In this case we
will return to the task message loop to see if there are any messages to
process. This is important because we do not want the task message
queue to overflow. If there aren't any genuine messages we want to
continue executing as soon as possible. In order to ensure this, we put
a special message in the queue. This message should be processed by
calling again this function.
      */
            esim_send_msg(RPD_RUN_CYCLES, (U8)0, NULL);
       }
}
```

1. `MoreToDo` value

   The function `rpd_PrivRunIdle` returns a value composed of 3 possible flags (described in the "Bit Values Table" in the section "The State Machine and the 'More To Do' Return Value" in the *Generating Code for Embedded Systems* manual). If this return value is not equal to zero, the MMI Application is not yet in a stable state and additional cycles must be executed. The number of continuous cycles is limited by the constant `MAXCYCLES`. The recommended `MAXCYCLES` value is 20.

2. Exceeding the `MAXCYCLES` value

   In case the application requires more than `MAXCYCLES` cycles, a mechanism for sending a message to the RapidPLUS task is implemented in order to prevent the RapidPLUS task's queue from overflowing.

Chapter 3: Basic MMI Application Integration

In order to integrate the MMI Application API, you will need to (i) integrate the platform keypad with the MMI Application's keypad; and (ii) integrate the graphic display with the platform LCD.

Integrating the Platform Keypad with the MMI Application's Keypad

The EMB_KPD component is the interface of the MMI Application to the real keypad. To integrate the real keypad with EMB_KPD, the Keypad task must send a message to the MMI task each time a pushbutton is pressed or released.

Normal Pushbutton Presses

After receiving notification from the Keypad task, the integration layer of the MMI task must translate it into a RapidPLUS message of type KEY_INFO in the KEY_MSG union in EMB_KPD's interface. In order to perform this translation, the integration layer must allocate a structure of type KEY_INFO (see emb_kpd.h) and assign the fields according to the data received.

The following table presents the expected sequence of messages, type KEY_INFO, in a press on the "2abc" key:

press_state	**state**	**virtual_key_id***
1. FIRST_PRESS	PRESSED	TWO (2)
2. INSIGNIFICANT	RELEASED	TWO (2)

* The expected value for the virtual_key_id field is one of the items in KEY_VALUE_Cs in the RPD_WINDOW interface.

To send the KEY_IFO message, use the following macro from mainapp.h:

```
R10597_EMB__KPD_send_KEY__MSG_KEY__INFO(embStructPtr,size)
```

Long Presses and Repeating Presses

The MMI Application expects the embedded system to analyze the timing of presses and releases in order to supply information about long presses and repeating presses. The MMI Application reacts differently to these types of presses.

The following table presents the expected sequence of messages, type KEY_INFO, in a long press of the left soft key:

press_state	state	virtual_key_id*
FIRST_PRESS	PRESSED	SK_LEFT (501)
LONG_PRESS	PRESSED	SK_LEFT (501)
REPEAT_PRESS	PRESSED	SK_LEFT (501)
REPEAT_PRESS	PRESSED	SK_LEFT (501)
...	PRESSED	SK_LEFT (501)
INSIGNIFICANT	RELEASED	SK_LEFT (501)

* The expected value for the virtual_key_id field is one of the items in KEY_VALUE_Cs in the RPD_WINDOW interface.

Refer to the section "Sending a Structure form the Embedded System to RapidPLUS" in the chapter "Interfacing with Generated User Objects" in the *Generating Code for Embedded Systems* manual.

Integrating the Graphic Display with the Platform LCD

The MMI Application's graphic display can be integrated with the platform's LCD in one of two ways: (i) using the MMI Application's true color graphic display object; or (ii) replacing the RapidPLUS graphic services with those of the platform. In some cases you may want to use the true color graphic display object with external bitmaps and/or fonts.

Using the RapidPLUS graphic services makes the integration of the MMI Application with the platform simple and allows testing of the exact graphic behavior in the simulation environment. In case you need to reuse specific parts of the embedded graphic services (e.g., fonts or font engine), you can combine RapidPLUS graphic services with the platform's graphics.

In the MMI Application, the components that manage the graphic services are:

- **EMB_BITMAPS** contains all of the MMI Application's bitmaps. The bitmaps are automatically imported into this component by the Multimedia Resource Manager tool. EMB_DISPLAY also contains the MMI Application's fonts. The RapidPLUS font objects are bitmap fonts based on fonts that are compatible with Microsoft® Windows.
- **EMB_DISPLAY** contains the true color graphic display object (GDO). This component links the application code to the display device.

- **SRV_DISPLAY** integrates all other graphic display services and is the front-end of the graphic display services to the MMI Application.

Using the MMI Application's True Color Graphic Display Object

The heart of the graphic services is the RapidPLUS true color graphic display object. The embedded graphic display object (GDO) interacts with the graphic devices via the RapidPLUS graphic display library (GDL). This libary is a stand-alone library of functions that exists outside of the MMI task.

➡ **Integrate the GDO using the RapidPLUS GDL:**

1. Generate SRV_DISPLAY, EMB_DISPLAY, and EMB_BITMAPS as full objects (UDOs).
2. Implement the GDO initialization and the `bitblt` function (as described in the chapter, "Integrating Graphic Displays" in the *Generating Code for Embedded Systems* manual). Follow the example in App_Api.c, which is located in the platform folder.
3. When linking, include the ugxxx.lib graphic library that you received from e-SIM (see "Software" on p. 1).

Using the Platform's Graphic Services

The MMI Application's native graphic display services can be adapted or replaced with the platform's graphic services. When deciding on whether to adapt or replace RapidPLUS graphic elements, you should consider font and bitmap issues.

When Using the RapidPLUS True Color Graphic Display Object

The information is this section applies if you will be adapting the graphic display object to work with your own fonts and/or graphic display libary.

Fonts

If your fonts are Microsoft Windows compatible, then replace the currently used fonts in the RapidPLUS font objects (defined in EMB_DISPLAY).

If your fonts are not Windows compatible, you must provide a solution for the simulation environment. This solution can either use similar fonts available in Windows or you should create a DLL with the fonts in their native formats. This DLL must provide a bitmap in Windows format for each character. You will need to integrate this DLL as a DLL object in EMB_DISPLAY.

Note: Changing font sizes may require changing the UI definitions.

Bitmaps

In the e-SIM MMI Solution, bitmaps are managed by the Multimedia Resource Manager tool (MultimediaResourceManager.xls). They are automatically imported into EMB_BITMAPS in the simulation environment, and are then generated in an RapidPLUS-specific format suitable for the embedded graphic display library (GDL).

In case you use your own graphic display library, you can use the Multimedia Resource Manager's code generation capabilities, adapting its macro to your own format. Refer to the *Customization Tools Guide* for details.

In case different bitmap formats need to be supported in the embedded libary (GDL), you can enhance the format support in the format driver's C files: fd_gen.c and fd_gen.h.

When Using an External Graphic Display

The information is this section applies if you will be replacing the MMI Application's services with the platform's graphic services, including fonts and bitmaps.

➨ **Integrate the graphic services using an external library:**

1. Generate SRV_DISPLAY as a full object (UDO) and EMB_DISPLAY as an interface-only object (UDI).
2. After code generation, implement (in C) the internal functionality of EMB_DISPLAY (so that it fits its interface) using the external library (for information about the object's interface and functionality, refer to the *MMI Component Reference Guide*).

Chapter 4: Protocol Stack Integration

The MMI Application can be integrated with the protocol stack in two ways: (i) using string-based AT commands; or (ii) using an abstract function-based service layer.

Integrating with an AT Command-Based Protocol Stack API

The interface in the component EMB_AT provides a set of basic functions for sending and receiving string-based AT commands. For details about these functions, refer to the *MMI Component Reference Guide*.

There may be some small differences between how protocol stacks implement the AT command protocol, for example, different timing or sequencing. AT commands used by the MMI Application follow the GSM 07.07 standard. In order to verify the match between the MMI Application and your protocol stack, test each functionality (for example, incoming call, outgoing call, etc.) tracing the sequence of the AT commands.

The MMI Application uses the following extensions to the GSM 07.07:

- %CPI- Call progress indication
- %NRG - network registration including limited service mode
- %CSQ- RSSI signal quality level indication

IIntegrating with a Function-Based Protocol Stack API

In cases where the protocol stack API does not support AT commands, you can use the following approaches:

- Generate the relevant low-level MMI Application service components as Interface Only (UDIs) and then implement these interfaces using the protocol stack API.
- Adapt these service components for the protocol stack API and then generate them as UDIs. Adapt the related high-level components accordingly.

Protocol stack integration can be achieved using either or both of these approaches. You should examine the interface of each service component to determine which approach is best for it.

Generate First, Then Implement

This way is recommended when the interfaces of the low-level service components are similar to the API of the protocol stack.

1. Generate the relevant low-level services (SRV_TAPI, SRV_SMS_LOW, and SM_PBOOK) as UDIs.
2. Implement these interfaces using the protocol stack API. For details about the interfaces of these functions, refer to the MMI *Component Reference Guide*.

Note: Changing the UDI interface will require that you adapt the upper layers.

If you keep the interface of the UDIs "as is" you will be able to reuse the "simulation only" components even though they are implemented using AT commands.

Adapt First, Then Generate

This way is recommended when the interfaces of the low-level service components do not match the API of the protocol stack.

1. Adapt the relevant low-level services API (SRV_TAPI, SRV_SMS_LOW, and SM_PBOOK) for the protocol stack API.
2. Adapt the related upper-layer components (services and HMI modules) for the changes made to the low-level services API.
3. Generate the low-level services as UDIs.

With this approach, the protocol stack simulation—which is based on AT commands—will need to be modified. The components involved in protocol stack simulation are (i) the low-level services and SRV_AT; and (ii) EMB_AT and SIMUL_AT.

Chapter 5: Basic Hardware Services Integration

You will need to integrate the MMI Application with services such as battery, file system, tones, melodies, and backlight.

The components that handle the interface to hardware services are embedded interface components. These components are usually generated as interface-only objects (UDI).

Important: Be sure that you integrate ALL the interfaces. Lack of integration of some of the services may prevent the system from starting.

If the interfaces of UDIs do not match the interface of the platform, you can:

- Change the interfaces in the UDIs—making sure to modify other components that use them—and then generate code. Be very careful changing the interface because the code generator might not keep the user code (the signature of the functions includes the parameters).
- Enhance the adaptation layer (that is, the manually written user code) to fit the original interfaces of the UDI components.

This chapter briefly describes the relevant interface components. You should refer to the *MMI Component Reference Guide* for details about each component's interface. You should also refer to example code in the Platform subfolder.

Integrating the File System

The MMI Application expects file system functionality in the platform. The interface is located in EMB_FILEMNG. The interface is C-like. The simulation environment uses an ActiveX object for this functionality (RpdSimFFS.ocx).

Integrating the Platform Services

The integration points for most of the services in the platform are located in EMB_HSRV. Some of these services are provided to the MMI Application directly by EMB_HSRV and others are abstracted by SRV_UTIL.

Integrating the Smart Editor Engine (T9)

Chinese support, multi-tap support, and smart editing support are implemented in WMI_EDITOR and WDG_TEXTBOX using a T9® engine supplied by Tegic Communications, Inc. This interface is presented by EMB_T9. In order to use the MMI

Application with the T9 engine, you must obtain a license from Tegic. Additionally, the files rpteal.c and rpteal.h contain integration code that must be used.

Integrating the Camera

The camera application is provided as a reference implementation based on a software and hardware environment provided by TransChip, Inc. This interface is presented by EMB_CAMERA. In the reference implementation, much functionality is implemented in external code provided by TransChip. You may need to enhance the RapidPLUS implementation to fit other camera environments.

Integrating Graphic Codecs

EMB_ALBUM provides interface for manipulating images. It expects software that (i) expands compressed images to a format that allows drawing; and (ii) can resize these images. The codecs are not provided by e-SIM.

Calculator Functionality

The calculator application (HMI_CALCULATOR) implements the basic functions inside EMB_CALC using native C arithmetic functionality. You do not need to change the implementation of EMB_CALC unless the functionality used is not supported by your environment.

Interface to External Applications

Currently, integration with external applications is based on Start and Stop functionality. HMI_EXT_APP is the representative of the external applications in the main MMI. Each external application has its own interface component: EMB_JAVA, EMB_MMS, and EMB_WAP. The interfaces of these components can be enhanced to match the real requirements.

Conditional Code

Through EMB_CONFIG, you can control the inclusion/exclusion of components in the embedded MMI Application. EMB_CONFIG has a constant set (HMI_Cs) that is used in a condition to include or exclude a component. In the simulation environment, the value of the constant set items is 1 (i.e., include).

If you want to exclude a component from the embedded MMI Application, set the value to zero BEFORE compilation. The best way to do this is to keep a version of the generated h file (emb_config.h) with the appropriate values and overwrite the generated code before compilation by copying this file instead of the newly generated one. You can have a batch file execute automatically after the code generation process bu using a Run command set in the “Command to Run After Code Generation” option in the Code Generation Preferences dialog box, General tab.

Additional Hardware Services Integration

The following components are described elsewhere:

EMB_ANIM_DATA: refer to Chapter 3: "Multimedia Resource Manager" in the *Customization Tools Guide*.

EMB_AT: see "Integrating with an AT Command-Based Protocol Stack API" on p. 11.

EMB_BITMAPS and EMB_DISPLAY: see "Integrating the Graphic Display with the Platform LCD" on p. 8.

EMB_KPD: see "Integrating the Platform Keypad with the MMI Application's Keypad" on p. 7.

EMB_LM_DATA: refer to Chapter 5: "The Layout Editor" in the *Customization Tools Guide*.

EMB_TEXTRES: refer to Chapter 4: "Menu and Text Generator" in the *Customization Tools Guide*.