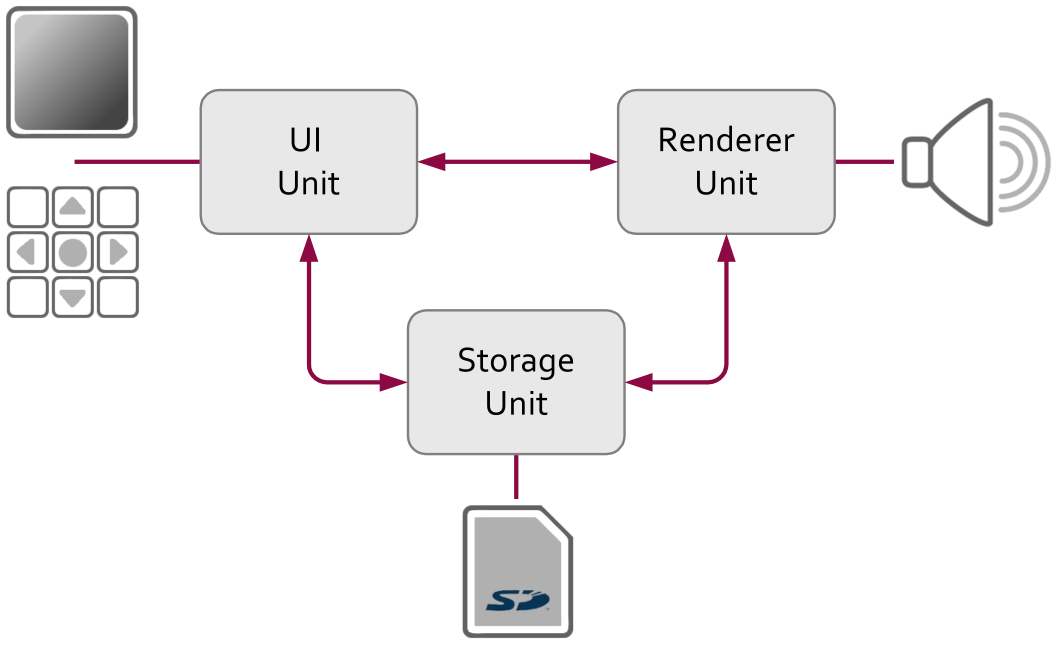
WHITE PAPER

mbLay - Inter-Process Communication Model

This white paper gives an introduction into the communication model which is a key feature of the embenatics foundation layer. It is explained by using a simple MP3 player example.

Introduction to the Sample Application

In this series of white papers a simple MP3 player is used as a sample application to show the various features and working steps of the embenatics tool suite. The figure below shows the building blocks for the MP3 player example. A short description of each block will facilitate the understanding of the basic functionality of the player. Each white paper will focus on different subjects of the sample application design to highlight the specific topic of the paper.



Block Diagram of the MP3 Sample Application

The UI Unit handles the user interaction, which comprises user commands as well as displaying status information, like which track is playing, the list of selected titles, etc. Access to the Storage Unit is used to provide all information about the available music tracks. The UI Unit interfaces with the Renderer Unit to pass on user commands, as well as to display the current status of the player.

The Storage Unit keeps track of the available music titles and provides access to the stored MP3 files. It interfaces with the UI Unit to provide track information. The interface with the Renderer Unit provides access to the coded music data that should be replayed.

The Renderer Unit is responsible for handling the music track to be played, managing a play list of titles, displaying information on the state of the player and converting the coded MP3 data into audible music. It receives track information and control commands via the UI Unit interface. Access to the MP3 data is obtained by interfacing with the Storage Unit.

In this white paper the MP3 player application is implemented by using five threads. The DISPLAY thread performs the graphic commands for drawing icons and text, and utilizes the LCD driver for visualization purposes. The KEYPAD thread receives information about key presses from the keypad driver. It forwards the key presses to the UI thread that controls the entire MP3 player. Additionally, there is the STORAGE thread managing a database containing the information about tracks, albums and artists stored on the memory card. A RENDERER thread is responsible for accessing the audio hardware and informing the UI about the audio renderer's state.

Communication Using Messages

A very common inter-process communication approach in multi-threaded or multi-core systems is the exchange of messages. For this purpose, most operating systems provide an API to send and receive messages containing data structures which consist of a message identifier and related parameters. In a synchronized communication, the sender needs to wait until the receiver has processed the message. This necessitates the receiver sending back a reply message after processing the request. Such a reply might be accompanied by some parameters reporting the status and/or giving back results to the initiating sender.

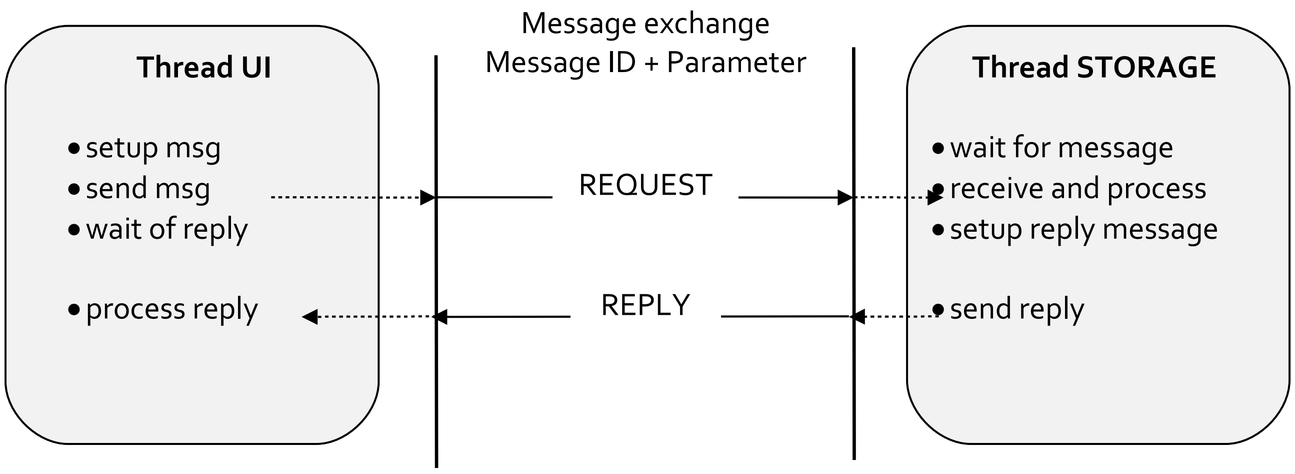


Figure 1 Traditional Message Based Communication

The usage of message-based communication implies that you know the partitioning of your software in subsystems. When sending a message, you need to know the thread to which you send the message; if this thread resides on a different core, it might require additional effort. If you want to change the partitioning at a later stage in your development cycle you have to modify your implementation to ensure that the message reaches the correct receiver.

Communication using Remote Procedure Calls

The communication model of the embenatics foundation layer, together with its tool chain, provide you with a further abstraction level. It is much more intuitive in terms of procedural programming and supports you in keeping your software independent and portable.

The embenatics foundation layer offers a remote procedure call model (RPC) in order to abstract from any CPU-core or thread partitioning during implementation. The RPC model is an established and standardized communication type and has been used in the UNIX network world for several years. The embenatics RPC model is optimized for the needs and limitations of an embedded system.
According to the RPC model, information is exchanged among subsystems by simply calling a C-function and passing the related parameters. If required, the function call returns its result to the caller.

Let's get back to the example of the MP3 Player. The UI Unit is in charge of executing the user input-triggered state machine of the player and building up the display objects for the LCD.
For the "Now Playing" view of the player, the UI Unit needs to get the status from the Renderer Unit first in order to determine the reference and the status of the song which is currently played. With that information, the UI Unit will obtain text strings about the artist, album and title together with the cover picture of this song from the Storage Unit. The Storage Unit has access to the storage media which contains the available MP3 files. The "Now Playing" screen looks like this:

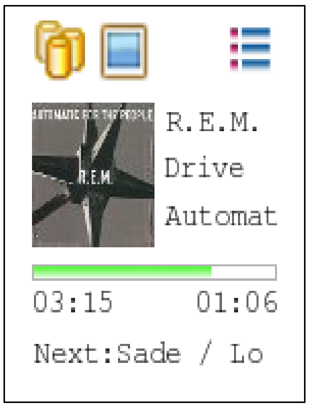


Figure 2 Screen Shot of the MP3 Player's "Now-Playing" View

When the UI, Storage and Renderer units are running in the same thread, you would most likely implement a sequence of function calls. First you would call a function of the Renderer in order to get the status of the actual played song. Then you would obtain the song-related information together with the cover bitmap from the Storage Unit by a subsequent call. Finally you would update the display with the formatted text and graphics components.

The code snippet taken from our implementation shows how the data from the renderer and the storage is obtained. Error handling is stripped off in order to keep the example simple:

```
....
/*
 * Used data structures
 */
RENDERER_CONTROL_RENDER_STATUS_type * renderer_status;
STORAGE_INFO_TRACK_INFO_type * track_info;
STORAGE_INFO_COVER_DATA_type * cover;

/*
 * Obtain the status of the actual song
 */
renderer_status = RENDERER_CONTROL_get_status();

/*
 * Get the track information Artist, Album, Title etc. from the storage manager
 */
track_info = STORAGE_INFO_get_track_info(renderer_status->track_id);

/*
 * Get the cover bitmap medium size in chunks (rows_to_read, row_offset)
 */
cover = STORAGE_INFO_get_cover(
             renderer_status->track_id,
             STORAGE_INFO_MEDIUM_COVER,
             rows_to_read,
             row_offset
        )
/*
 * Display information on screen and deallocate renderer_status, trac_info and cover
 */
....
```

This was easy! Let’s imagine you have a system design where these three units run in separate threads. A function call is unsafe and not good programming practice because of the lack of thread synchronization. You might tend towards using a message-based communication but this would mess up the code above by introducing routines for sending messages and receiving the reply. Furthermore, your code will become dependent on the chosen thread architecture.

The RPC communication model of the embenatics foundation layer solves this problem.
Believe it or not - the code from the single-threaded example above won’t change in a multi-threaded environment!

How is this possible? The foundation layer automatically converts the function calls together with its parameters into messages and sends them to the thread that has implemented this function. It also ensures that the destination function is called only when the receiving thread is in a state where it is capable of performing the operation. This will keep the thread’s global data consistent. The transport of the return value to the caller is handled by the foundation layer as well.

In order to use this RPC functionality, you need to tell the foundation layer which functions you are calling via RPC and what the data types of the parameters and the return value will look like. For this purpose you are well-supported by the tool chain of the embenatics product portfolio.

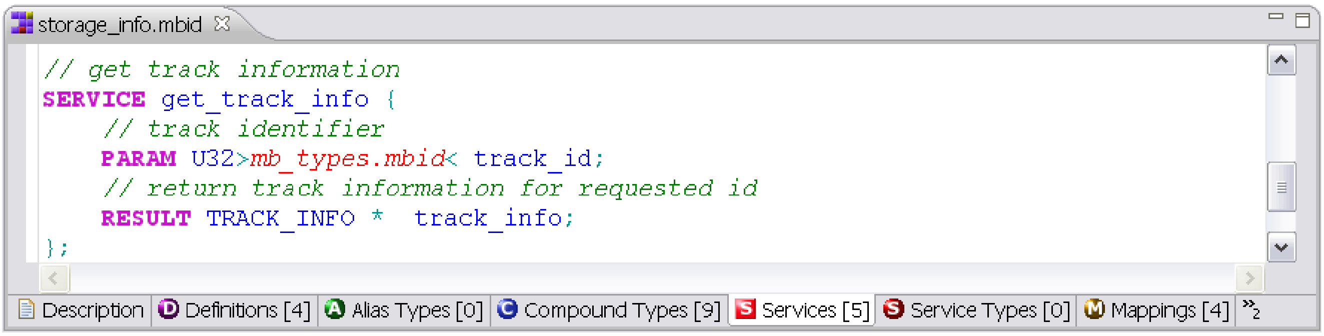
Let’s focus on the function: **`STORAGE_INFO_get_track_info`**

```
...
RENDERER_CONTROL_RENDER_STATUS_type * renderer_status;
STORAGE_INFO_TRACK_INFO_type * track_info;
STORAGE_INFO_COVER_DATA_type * cover;

/*
 * Obtain the status of the actual song
 */
renderer_status = RENDERER_CONTROL_get_status();

/*
 * Get the track information Artist, Album, Title etc. from the storage manager
 */
track_info = STORAGE_INFO_get_track_info(renderer_status->track_id);
...
```

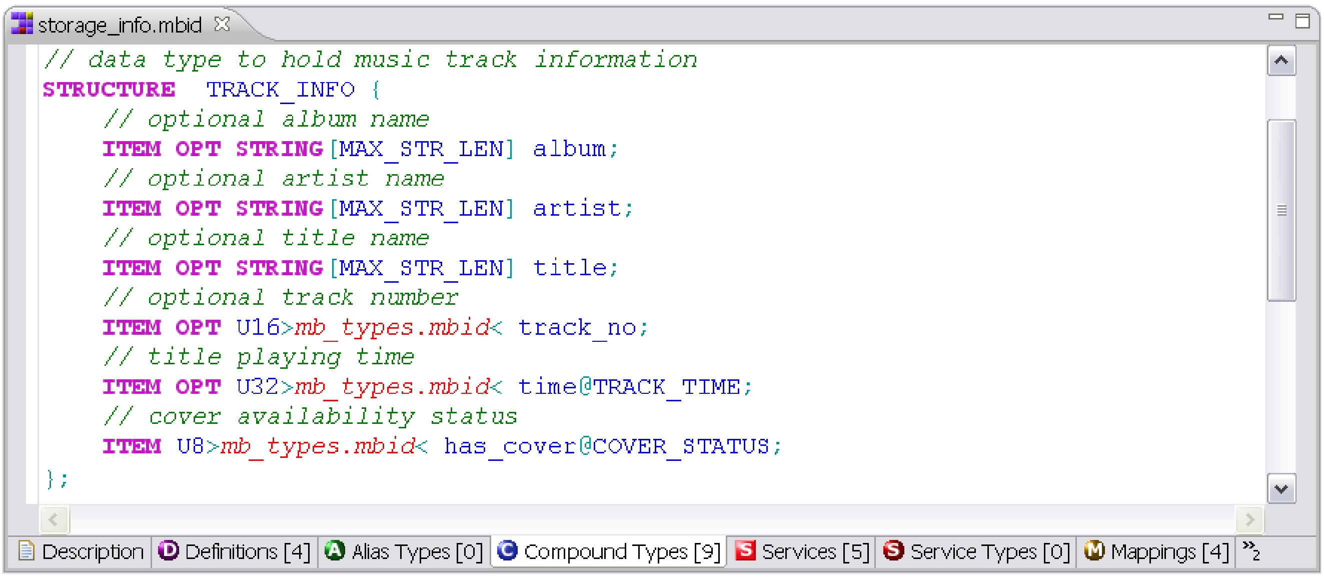
In our embenatics terminology we call an RPC function a "service". Prefixes like STORAGE_INFO_ and RENDERER_CONTROL_ are used in our example to indicate to which interface a service belongs. In a so-called interface description document (MBID), you specify the function together with its parameters. The language of this description is very similar to C-code and we provide an editor which supports you in this task. A detailed description of the language and the tool can be found in another publication of this series of white papers [embenatics Interface Description].



```
// get track information
SERVICE get_track_info {
    // track identifier
    PARAM U32>mb_types.mbid< track_id;
    // return track information for requested id
    RESULT TRACK_INFO *  track_info;
};
```

Figure 3 Definition of the get_track_info Service

The service *get_track_info* expects a parameter *track_id* which is a 32bit value and returns a pointer *(*track_info)* to a data structure of the type *TRACK_INFO*. As you can see in the next figure, this data structure is defined with the help of the same editor.



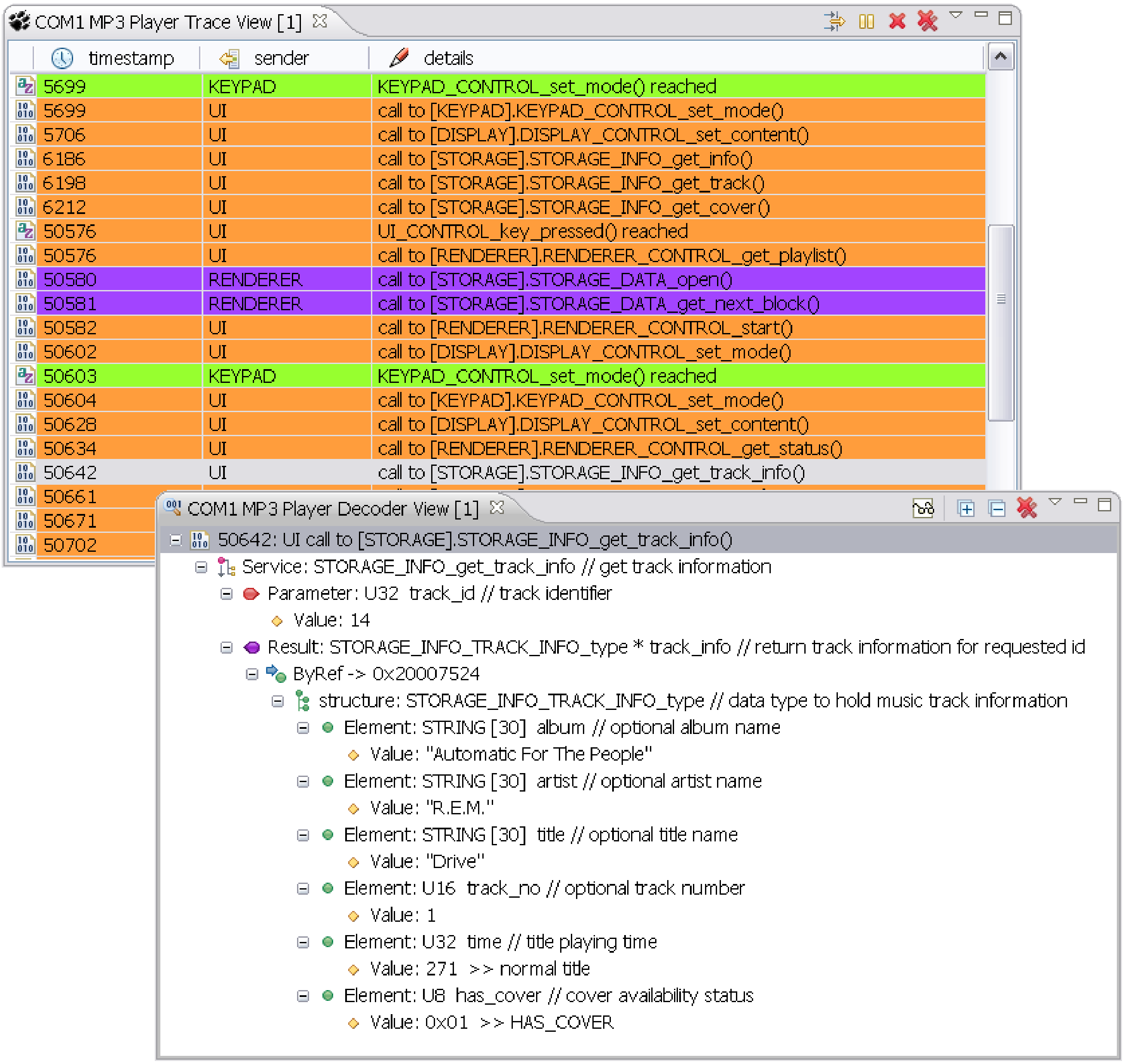
```
// data type to hold music track information
STRUCTURE  TRACK_INFO {
    // optional album name
    ITEM OPT STRING[MAX_STR_LEN] album;
    // optional artist name
    ITEM OPT STRING[MAX_STR_LEN] artist;
    // optional title name
    ITEM OPT STRING[MAX_STR_LEN] title;
    // optional track number
    ITEM OPT U16>mb_types.mbid< track_no;
    // title playing time
    ITEM OPT U32>mb_types.mbid< time@TRACK_TIME;
    // cover availability status
    ITEM U8>mb_types.mbid< has_cover@COVER_STATUS;
};
```

Figure 4 Definition of the TRACK_INFO Data Type

Once a function is defined to be an RPC service, a call to this function will automatically result in a remote procedure call. The foundation layer will automatically route to the service-implementing thread, even if it is located on a different CPU core.
The implementation of the service itself is quite straight forward. Besides a special macro it is normal C-code:

```
STORAGE_INFO_TRACK_INFO_type *  RPC_IMPL(STORAGE_INFO_get_track_info)( U32 track_id )
{
  STORAGE_INFO_TRACK_INFO_type * track_info;
  /*
   * allocate and fill the track_info structure with the MP3 related data for track_id
   */
  ...
  return track_info;
}
```

The development tool chain will also generate the corresponding C-header files which consist of function prototypes and data type definitions for the described service.
Furthermore, it allows you to log each of the specified service calls with all the information about the actual parameters and the return values. The following picture shows the output of the embenatics logging tool [mbLog Logging and Diagnosis Tool]. In the top window you see a snapshot of the communication in the "Now Playing" dialog. The bottom window shows the content of the RPC parameter together with the return value (Result) of the call.



timestamp	sender	details
5699	KEYPAD	KEYPAD_CONTROL_set_mode() reached
5699	UI	call to [KEYPAD].KEYPAD_CONTROL_set_mode()
5706	UI	call to [DISPLAY].DISPLAY_CONTROL_set_content()
6186	UI	call to [STORAGE].STORAGE_INFO_get_info()
6198	UI	call to [STORAGE].STORAGE_INFO_get_track()
6212	UI	call to [STORAGE].STORAGE_INFO_get_cover()
50576	UI	UI_CONTROL_key_pressed() reached
50576	UI	call to [RENDERER].RENDERER_CONTROL_get_playlist()
50580	RENDERER	call to [STORAGE].STORAGE_DATA_open()
50581	RENDERER	call to [STORAGE].STORAGE_DATA_get_next_block()
50582	UI	call to [RENDERER].RENDERER_CONTROL_start()
50602	UI	call to [DISPLAY].DISPLAY_CONTROL_set_mode()
50603	KEYPAD	KEYPAD_CONTROL_set_mode() reached
50604	UI	call to [KEYPAD].KEYPAD_CONTROL_set_mode()
50628	UI	call to [DISPLAY].DISPLAY_CONTROL_set_content()
50634	UI	call to [RENDERER].RENDERER_CONTROL_get_status()
50642	UI	call to [STORAGE].STORAGE_INFO_get_track_info()
50661		
50671		
50702		

```
50642: UI call to [STORAGE].STORAGE_INFO_get_track_info()
  Service: STORAGE_INFO_get_track_info // get track information
    Parameter: U32  track_id // track identifier
      Value: 14
    Result: STORAGE_INFO_TRACK_INFO_type * track_info // return track information for requested id
      ByRef -> 0x20007524
        structure: STORAGE_INFO_TRACK_INFO_type // data type to hold music track information
          Element: STRING [30]  album // optional album name
            Value: "Automatic For The People"
          Element: STRING [30]  artist // optional artist name
            Value: "R.E.M."
          Element: STRING [30]  title // optional title name
            Value: "Drive"
          Element: U16  track_no // optional track number
            Value: 1
          Element: U32  time // title playing time
            Value: 271  >> normal title
          Element: U8  has_cover // cover availability status
            Value: 0x01  >> HAS_COVER
```

Figure 5 Logging the Remote Procedure Call of the Service get_track_info

Even if you decide to use the traditional message-passing method, the embenatics tool chain supports you in defining message identifiers together with the appropriate parameter structures. A detailed logging of messages is fully supported, too.

Communicating Across Subsystem Boundaries

Let’s imagine you want to re-use the implementation of the three units of the MP3 player in a media player system. The system consists of a handheld device with the LCD and the keypad acting as a remote control unit for browsing and selecting songs from your MP3 library. The remote control is connected to a media server using a wireless network. The media server hosts your MP3 library and contains an audio renderer in order to decode the MP3 and play music via your speakers in the living room.

As shown in the MP3 design model at the beginning of this document, the UI Unit will run on the remote control device while the Renderer and the Storage units are located on the media server device.

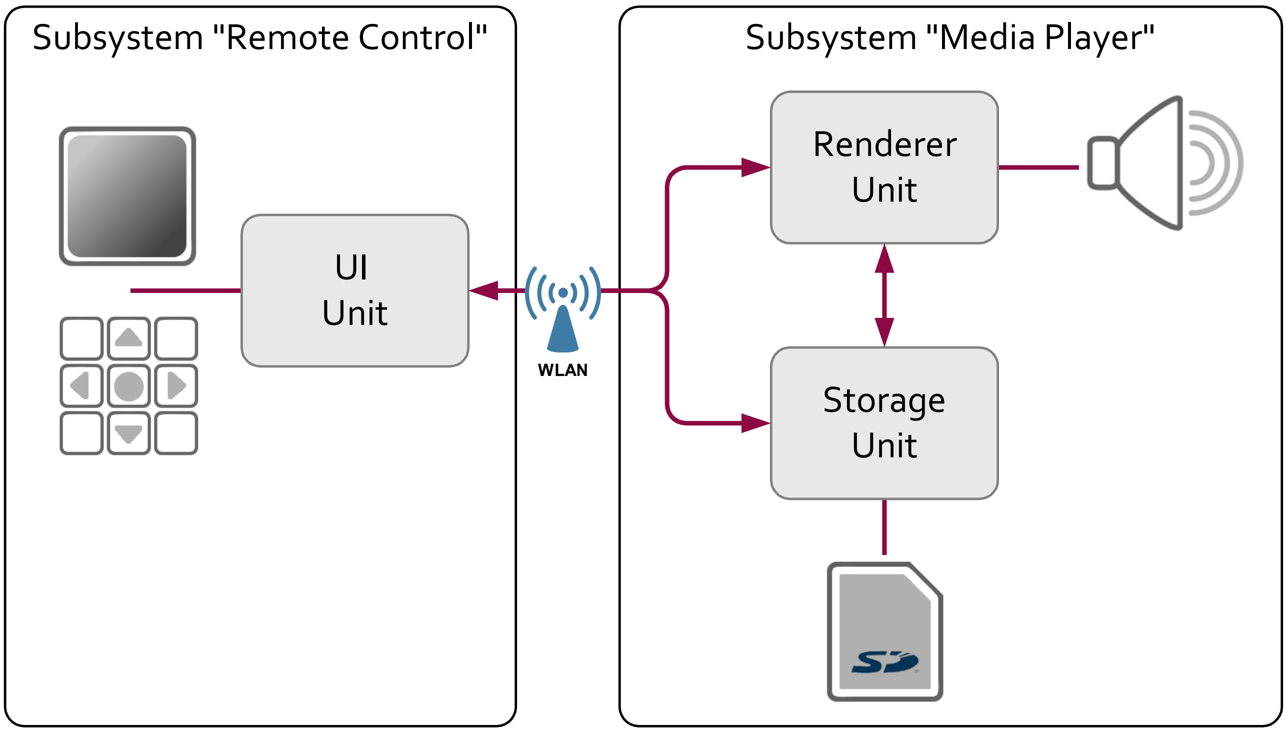


Figure 6 Partitioning the MP3 Player in Different Subsystems

In this case, the remote procedure call is performed from the runtime environment of one CPU (remote control unit) and transmitted to the environment of the media server CPU. This means that the calling thread and the called thread run in different memory environments. Further complexity is added by having the CPUs of both systems differ in terms of endianness, memory alignment, and by their ability to use a different operating system. Will this work?

Yes! In fact, the foundation layer is capable of detecting system boundaries and copy memory regions from one system to another by converting between different memory data layouts. You are also allowed to use pointers in the exchanged data as long as you describe the data type of the pointer. If the service call is performed via system boundaries, the memory content to which the pointer is pointing to will be cloned by the foundation layer at the destination CPU. For this purpose, it is required to use the embenatics foundation layer on all interacting systems. This is possible because the foundation layer is available for several CPU types and operating systems. The definition of the thread location and the communication relationship is done via the system description editor for which you can find more detailed information in another publication of this series of white paper [embenatics System Description].

Even in such a distributed multi-core environment, the implementation of the UI example shown above stays unchanged. You can see that the embenatics RPC communication model supports you in coding the program flow by using simple function calls independent of the thread-model, thread location and communication media. Even in a distributed system, the communication among the threads can be logged in the same way as already briefly described for the single chip solution.

Since not all communication relations among subsystems are required to be synchronous, meaning that the caller of a service is blocked until the called service has terminated, it is also possible to define a service as non-blocking.
Example: The keyboard thread does not have to wait until the UI has processed the key event. In this case, services can be marked as "non-blocking" in the respective interface description.

It is allowed to call another service from a service implementation. The foundation layer maintains the correct call order. Therefore cascading call hierarchies are therefore no problem in the RPC context.

Support of Distributed Development

The RPC model allows you to move software units from an embedded system to a PC for development and debugging purposes and seamlessly move them back to the target system once they are working. This makes it possible that each function which is specified as an RPC service can be tested from outside the target system. In the MP3 example all services like e.g. the Storage Unit are accessible by a test system which runs outside of the target system.

Conclusion

In addition to the traditional message-based communication, the embenatics foundation layer provides an advanced RPC based communication model. This model allows you to implement the exchange of information among threads in a procedural way, independent of the underlying distribution of the subsystems. RPC is the basis for a machine-independent exchange of information and lays the foundation for re-use, distribution and testing of your implementation.

About Us

embenatics is a new company that entered the market in 2010. Our focus is on embedded software development; as such we offer a software foundation layer and tool suite that supports your development team in designing embedded software in an efficient, portable and maintainable way. Based on our wide and varied experience in embedded systems design and development, we know that future product requirements are hard to predict. Our goal is, therefore, to provide you with our technology to make the design of your products as flexible and adaptable as possible. Our approach allows your company to concentrate on the core competencies that differentiate your valuable product from those of your competitors.

Before embenatics was founded, we worked with well-known international companies over two decades and gained valuable experience in the embedded software business. While working as software developers and architects, we encountered the various challenges of the embedded software development life cycle. This wide range of experiences is the backbone of the software foundation products that are offered by embenatics.

Our business philosophy is to establish a close and trustful relationship with our customers in order to successfully promote and support projects over a long time period. For further information please contact

Joachim Pilz
Beerenstraße 29
14163 Berlin

info@embenatics.com
www.embenatics.com

Phone +49 30 26 34 75 28
Mobile +49 176 96 98 46 07