embenatics Interface Description

This white paper introduces the embenatics tool chain, specially the description of inter-process communication interfaces and data types based on a simple MP3 player example.

Introduction to the Sample Application

In this series of white papers, a simple MP3 player is taken 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 a simple MP3 player example. A short description of the blocks will make it easier to understand the basic function of the player. Each white paper will focus on different subjects of the sample application design to highlight the specific topic of the paper.

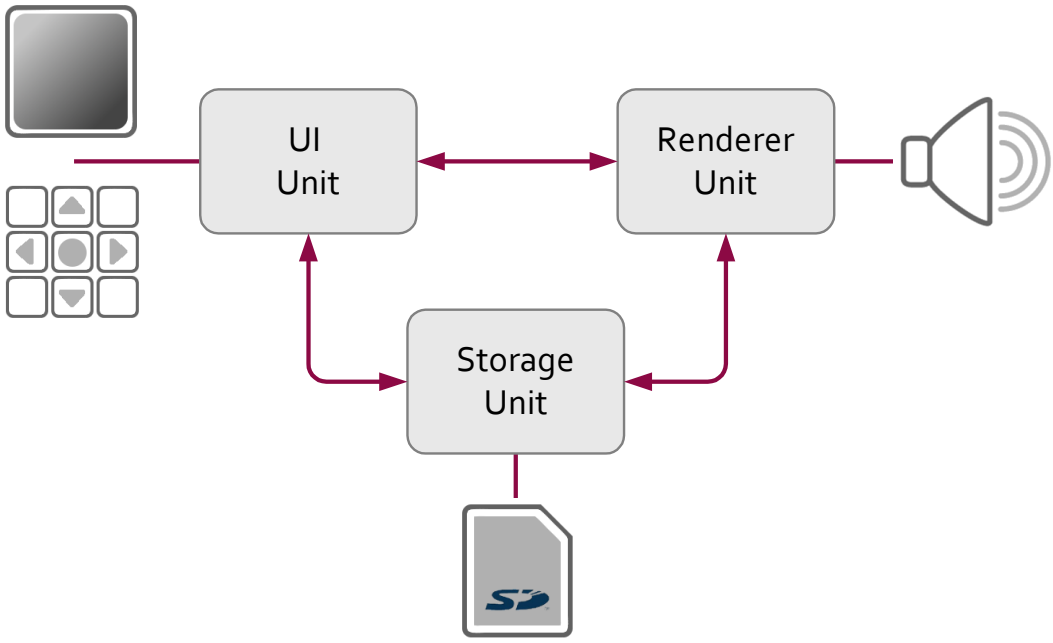

Figure 1 Block Diagram of the MP3 Sample Application

The UI Unit handles the user interaction, which comprises user commands as well as displays status information, like the current track that 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 get 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 five threads. Figure 2 illustrates the thread model of the MP3 player in a UML-like notation. Each thread is displayed showing its name and the interfaces implemented by each thread. The RPC based inter-process communication relationship among the threads is indicated by the arrows between the threads, e.g. the UI thread accesses services of the display_control interface of the DISPLAY thread.

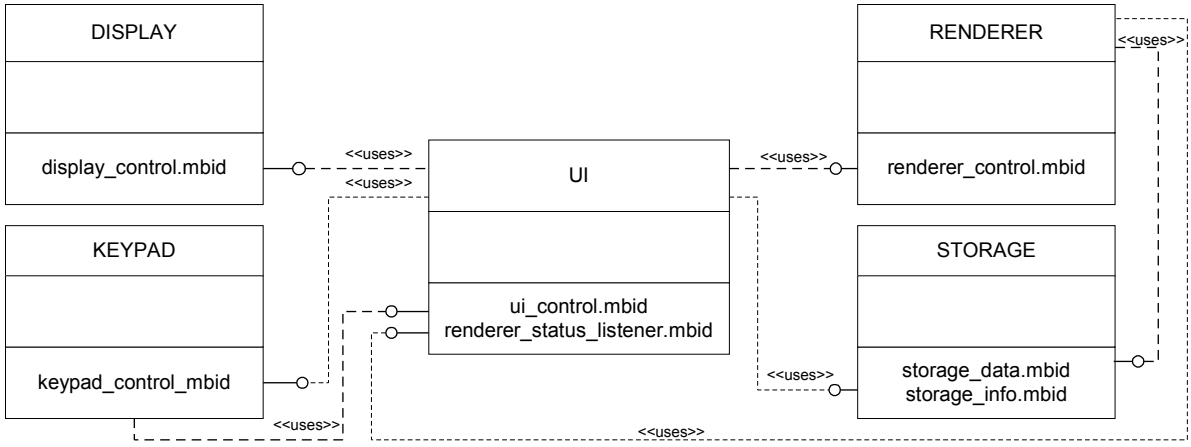

Figure 2 Thread model of the MP3 player

It has to be stated that the main intention of this example is to highlight the features of the embenatics foundation layer software as well as the capabilities of the tool chain. With this in mind, it should be acceptable that some design decisions are a bit off the most efficient and optimal solution software developers can think of.

Introducing embenatics Interface Descriptions - MBID

One of the key design principles using the embenatics tool suite is to describe all inter-process communication interfaces by using a dedicated interface description language (IDL). All building blocks that belong to the same interface are grouped by a so-called interface description document, called MBID. An MBID will hold the IDL which declares communication services, data types and definitions and will act as a single source input document from which development and diagnostic documents and files are derived. All generated documents are of great use throughout the development process of the target system. More details about the embenatics design methodology and tool chain can be found in another publication of this series of whitepapers [embenatics Design Methodology].

The embenatics tool suite not only defines the interface description language but is also linked to the easy to use accompanying editor mbEdit. This editor facilitates the creating and reusing of communication interfaces and assists in organizing these interfaces in different projects. The editor mbEdit is an Eclipse™-based tool which will seamlessly integrate and utilize the Eclipse™ workbench and other Eclipse™ plug-ins. The Eclipse™ platform is Java™-based and therefore runs on multiple PC operating systems like Windows™, Linux™ and MacOS™.

As we will see in greater detail in this document, mbEdit will be used to define inter-process communication services, data types and data definitions that are used to directly communicate between different threads of the system utilizing an RPC mechanism. More detailed information about the details of inter-process communication using embenatics foundation layer software can be found in another publication of this series of white papers [mbLay Inter-Process Communication Model].

In addition to services and data types, the user could also create supplemental diagnostic information, which will prove helpful for debugging and testing purposes when bringing the system to life. MBID documents can be organized in a similar fashion as header files that are commonly used in programming languages. This means that one MBID can reuse definitions of types and services of another MBID by referring to that element.

Encapsulating services and corresponding data types in MBID documents help streamline the interface design and will lay the foundation for the design of inter-process communication channels. Interfaces as well as the usage of system resources, i.e. threads, semaphores, memory, etc. are defined in description documents as part of the embenatics design methodology. This centralized approach will be used throughout the system design process. More information about the embenatics system description procedure and tools can be found in another publication of this series of white papers [embenatics System Description].

Once all MBID documents belonging to a system are created, they are fed to the embenatics generator tool mbGen. This tool will create all necessary artifacts that are needed for system setup, software development, as well as diagnostics and testing.

A Closer Look at Interface Descriptions for the MP3 Player

Based on our MP3 player example briefly described above, we will now take a closer look at how the interfaces are described in detail using some meaningful examples. As you can see in Figure 2, a set of MBIDs has been designed to fulfil the inter-process communication needs of the sample system. We will offer a quick glance at some of them to highlight some services and data types that are well suited to explain the details of using interface descriptions.

Defining Data Types

Basis for the definition of custom data types is a number of default definitions that are already declared by the embenatics tool suite. These default types comprise signed and unsigned types of various data sizes and are embedded in a dedicated MBID called `mb_types.mbid` which is a part of the embenatics products.

With this in mind, we are ready to create custom data types needed for the MP3 player example. In the following sections, we will examine the interface that resides between the UI and the Storage unit called `storage_info.mbid`, also shown in Figure 2. The main purpose of this interface is to provide access to the detailed information of the different music tracks that are managed by the Storage unit. One of those pieces of information is a bitmap picture that represents the cover art which belongs to a certain music track. In our example exist three different sizes of that bitmap picture, all squares, for different display situations. To keep the data transfer chunks a reasonable sizes, we have scaled the pictures into a set of rows for the different sizes. The following example shows the definition of a data structure holding the pixels for one row for the smallest bitmap size.

```
// data type to hold a row for a small cover size
STRUCTURE SML_COVER_ROW {
      // row data
      ITEM U16 >mb_types.mbid< data[SML_COVER_SIZE];
};
```

As we can see the IDL representation of that data type is quite straight forward. A keyword declares the type of the data as a structure followed by the type name `SML_COVER_ROW`. The structure itself consists of an array of type `U16` representing the pixel data itself. The type `U16` – 16 bit unsigned - for the pixel data has been imported from the default type interface description file `mb_types.mbid`, as explained above. For the size of the array we used a dedicated definition which stands for the number of pixels for a row. See the above IDL statement for that definition.

```
// small cover size
DEFINITION SML_COVER_SIZE 32;
```

All declarations can be preceded by explanatory comments as an option. We see later in this document, examples of the data logs and how these comments are very helpful during the diagnosis phase.
When the above IDL declaration will be fed into the embenatics generator mbGen, the following C constructs will be found in the corresponding header file.

```
#define STORAGE_INFO_SML_COVER_SIZE     32            // small cover size

/*----------
| data type to hold a row for a small cover size
/----------*/
typedef struct
{
      U16 data[STORAGE_INFO_SML_COVER_SIZE]; // row data
} STORAGE_INFO_SML_COVER_ROW_type;
```

Pre- and Postfixes

We can see in the previous example code snippet that the generated header file content differs from the IDL definition because the type name of the structure and the definition name have been extended. MBID files allow the definition of so called prefixes and postfixes that are valid for all relevant elements for the whole document. Prefixes are put in front of all element names and help to avoid duplicate declarations. In our MP3 Player MBID files we have set the prefix value always to the respective interface name; i.e. `STORAGE_INFO` for the example above. Postfixes are only applicable for new type declarations and are appended to the given element name. Our default postfix value for the MP3 Player MBIDs is set to `type`.
Pre- and postfixes help the user to organize the design of independent interfaces in multiple MBID documents. They can be used to create a type of namespace for MBID documents and therefore avoid compilation problems due to duplicate declarations.

Variable Arrays

To form a full cover picture, more than one pixel row is needed. Therefore the next data type that has been defined collects a set of pointers for all rows of an album cover picture in one array. The corresponding IDL for that data type is shown below.

```
// data type to hold small cover size, or a fraction of it
STRUCTURE SML_COVER {
      // rows
      ITEM SML_COVER_ROW * rows[~SML_COVER_SIZE];
};
```

At first glance that definition looks like any other, but a closer look shows you this key character ~ (tilde) , that is used for defining the array's size. This is a special feature of the embenatics tool suite that we call a "variable array" meaning that the number of actual filled array elements can vary between zero and the maximum array size. To support that behaviour, the generator adds an additional counter element to the data structure that will hold the number of present array elements. In our example this feature is used to read fractions of the album cover data to keep memory allocation at a lower level. See below the C constructs that have been generated by mbGen to reflect that data structure.

```
/*----------
| data type to hold small cover size, or a fraction of it
/----------*/
typedef struct
{
      STORAGE_INFO_SML_COVER_ROW_type * rows[STORAGE_INFO_SML_COVER_SIZE]; // rows
      /* variable array control starts here */
      U8 rows_c_;
} STORAGE_INFO_SML_COVER_type;
```

In addition to the array element, the counter `rows_c_` has been generated. For such elements a simple macro is provided to support convenient access for the programmer. Of course, when allocating memory for such a data type, the full size of the array has to be taken into account; but, when transferring data of that type between two different platforms, only the required amount of data is transported. That saves time and bandwidth in case of low data volume situations.

Unions

As stated above, our MP3 Player example has to deal with three different sizes of album cover art pictures. Therefore, similar to the data types for a small cover defined above, data types for medium and large covers exist. To show another feature of the embenatics tool suite, we group those data types in a union as a container. See the following IDL for that data type.

```
UNION  COVER_DATA {
      // small cover
      ITEM TAG[TAG_SMALL_COVER,1]  SML_COVER * sml_cover_bmp;
      // medium cover
      ITEM TAG[TAG_MEDIUM_COVER,2] MED_COVER * med_cover_bmp;
      // large cover
      ITEM TAG[TAG_LARGE_COVER,3]  LRG_COVER * lrg_cover_bmp;
};
```

What we see in this example is a tag declaration for each item of the union. It consists of a name and a value. When using that data type during programming, the tag helps to identify the different data types collected in that union. The user is free to choose tag name and value. Nevertheless, the tool chain can do that in an automated fashion. As a result, the generator will add a tag element to the data structure that can be accessed by a simple macro for convenience. See below the outcome of the generator.

```
typedef struct
{
      union
      {
            STORAGE_INFO_SML_COVER_type * sml_cover_bmp; // small cover
            STORAGE_INFO_MED_COVER_type * med_cover_bmp; // medium cover
            STORAGE_INFO_LRG_COVER_type * lrg_cover_bmp; // large cover
      } COVER_DATA_union;
      /* union control tag */
      U8 COVER_DATA_t_;
} STORAGE_INFO_COVER_DATA_type;
```

```
/* union tags defined for STORAGE_INFO_COVER_DATA_type starts here */
#define TAG_SMALL_COVER       1
#define TAG_MEDIUM_COVER      2
#define TAG_LARGE_COVER       3
```

Optional Elements and Value Mappings

As a last example for data types being used in the MP3 Player, we take a closer look at the music track information and how it is represented by the corresponding data type. Track information is extracted from the MP3 meta data fields, which describe e.g. artist name, song title, etc. What information in particular is present in the MP3 file can not be anticipated and solely depends on the MP3 file itself. To reflect that situation in the data structure we declare such elements as optional by using the keyword OPT. See below the example data type for the track information, represented by the IDL.

```
// 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;
};
```

Every item in a data structure that is declared as optional has a dedicated valid flag that represents whether the data for that item is valid and present or not. The generator takes care of that and creates such flags at the end of the data structure. Again, simple macros are available for easy access for the programmer.
Another small detail in the above data structure can be seen by looking at the string type, represented by the keyword STRING. The embenatics tool suite supports a dedicated type for zero terminated strings. This allows for the efficient handling of the transfer of such elements across system boundaries, making it possible that only the required number of string characters are exchanged instead of the maximum reserved memory space. For diagnostic and display purposes, string types will be given special treatment to be presented in a user friendly and readable format. Next we take a look at the generated C constructs that have been created for that data structure.

```
/*----------
| data type to hold music track information
/----------*/
typedef struct
{
      C8 album[STORAGE_INFO_MAX_STR_LEN+1];  // optional album name
      C8 artist[STORAGE_INFO_MAX_STR_LEN+1]; // optional artist name
      C8 title[STORAGE_INFO_MAX_STR_LEN+1];  // optional title name
      U16 track_no;                          // optional track number
      U32 time;                              // title playing time
      U8 has_cover;                          // cover availability status
      /* optional element control starts here */
      U8 album_v_         :1;
      U8 artist_v_        :1;
      U8 title_v_         :1;
      U8 track_no_v_      :1;
      U8 time_v_          :1;
} STORAGE_INFO_TRACK_INFO_type;
```

The other main topic in this chapter is value mapping. Value mappings can be assigned to data elements for an increased readability during the test and diagnostic phase. They are separately defined and assigned where needed by using the @ key character. In the above example we demonstrated that for the `time` and `has_cover` elements of the data structure. Both mapping declarations are shown below.

```
// track time mapping
MAPPING TRACK_TIME DEC {
      // short title
      RANGE 0 120;
      // normal title
      RANGE 121 360;
      // long title
      RANGE 361 1800;
};

// cover status mapping
MAPPING COVER_STATUS HEX {
      // has cover
      DEFINITION HAS_COVER 1;
      // has no cover
      DEFINITION NO_COVER 2;
};
```

`TRACK_TIME` is a simple example for a range mapping. A range is defined by a minimum and maximum value. That means if the data value for that element falls between the defined range settings, the associated comment will be assigned to that value.
Another example for a discrete value mapping can be seen for `COVER_STATUS`. Here the value has to exactly match the defined value, should the comment be assigned to it. Discrete value mapping declarations result also in definitions contained in the generated C-header files, which can be used by the programmer to assign these values to variables in the code.
In addition, the numerical representation of the target data item can be adjusted using mappings. In the above example a decimal and a hexadecimal representation has been set. There are other mapping strategies available, like pattern and limiter matching, which are not utilized in this

example. The outcome of this matching attribute can be seen when logging such elements and displaying them using the embenatics tool suite target logger mbLog.

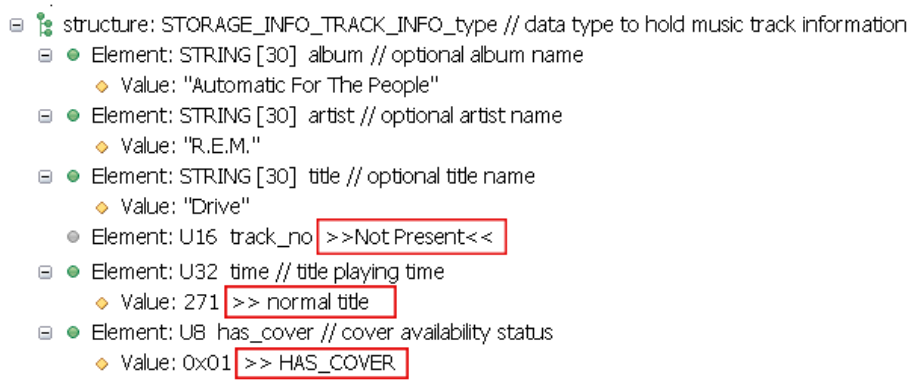

Figure 3 Decoded Data Type Example

In Figure 3 we can see the `STORAGE_INFO_TRACK_INFO_type` data structure decoded by mbLog. In this example the item `track_no` is not available and therefore it is set to not to present. In addition, the above explained mappings for `time` and `has_cover` have been highlighted to illustrate their usage. A more detailed introduction to the capabilities of mbLog can be found in another publication of this series of white papers [mbLog Logging and Diagnosis Tool].

Definition of Services

The embenatics foundation layer software uses services as communication objects and transport medium of above defined data type elements. Services are declared and grouped in MBID documents to form interface definitions and act as the functional access points for inter-process communication among threads that implement these interfaces. More information on inter-process communication can be found in another publication of this series of white papers [mbLay Inter-Process Communication Model].

Services are declared in a similar fashion as already briefly explained for data types. There are, however, some particular differences that are highlighted in this section of the document. We take a closer look at the MBID dedicated to the Renderer Unit that has been implemented by the RENDERER thread as shown in Figure 2, the `renderer_control.mbid` interface. Its main purpose is to control and supervise the music to be played by the MP3 device. Simple services like start, stop, pause and resume a playing track have been defined as shown below.

```
// pause current track
SERVICE pause NON_BLOCKING {};

// resume playing current track
SERVICE resume NON_BLOCKING {};

// start playing track from playlist
SERVICE start NON_BLOCKING {
      // track id to be played
      PARAM TRACK_ID>common_types.mbid< track_id @TRACK_ID>common_types.mbid<;
};
```

```
// stop playing current track
SERVICE stop NON_BLOCKING {};
```

What we can see in this example is that all services are declared by the keyword SERVICE followed by the name of the service itself. The services `pause`, `resume` and `stop` have been reduced to the simplest command without parameter or result. This is in contrast with the `start` service which has a parameter that passes the `track_id` that should be played by the Renderer Unit. In addition, the type TRACK_ID has been imported from another MBID called common_types.mbid and has been mapped to TRACK_ID of the same imported MBID.
As stated above, these services are used for inter-process communication among threads that implement the respective interfaces. To control the thread synchronization when accessing the services of an interface, some keywords are provided to categorize the service mode. In the example above, the keyword NON_BLOCKING is used which implies that a call to this service will return immediately without waiting for completion by the hosting process. These kinds of services can be seen as shot and forget triggers.
Another example for a service of this MBID is `get_volume`, which provides the current volume setting.

```
// get volume
SERVICE get_volume BLOCKING {
      // returns actual volume level
      RESULT U8F >mb_types.mbid< volume@VOLUME_MAP;
};
```

Here we see the definition of a service return value that is defined by the keyword RESULT. Due to the nature of this service, we have to wait for the output of this request. The service mode is set to BLOCKING, which means that the caller has to wait until the result can be provided by the called process. The service modes can be declared globally for a complete MBID or individually for each service as shown in the examples above.
Finally, we take a look at a more complex service that has parameters and a result and that will also utilize the cover data types that we have defined earlier in this document. The example service has been taken from the `storage_info.mbid` that is implemented by the Storage Unit. Here we have declared a service to get the cover for a music track, so that it can be displayed by the UI Unit. As we have seen, when defining the different data types that hold the cover data, there are different sizes of covers and it is possible to request parts of the cover data for bandwidth optimization. Therefore our service needs a set of parameters that help to customize the request for cover data of a music track. As a result we will get a pointer to the requested cover data contained in a `STORAGE_INFO_COVER_DATA_type`.

```
// get album cover data
SERVICE get_cover {
      // track identifier
      PARAM TRACK_ID>common_types.mbid< track_id;
      // requested cover size
      PARAM U8F>mb_types.mbid< size @COVER_SIZE;
      // requested rows of cover
      PARAM U8F>mb_types.mbid< rows;
      // requested offset of rows
      PARAM U8F>mb_types.mbid< offset;
      // return the cover data for the requested size
      RESULT COVER_DATA * cover_data;
```

```
};
```

The following figure shows a decoded example log for a call of the `get_cover` service from the UI Unit to the Storage Unit. This will also show the `STORAGE_INFO_COVER_DATA_type` union in use that was previously explained in more detail.

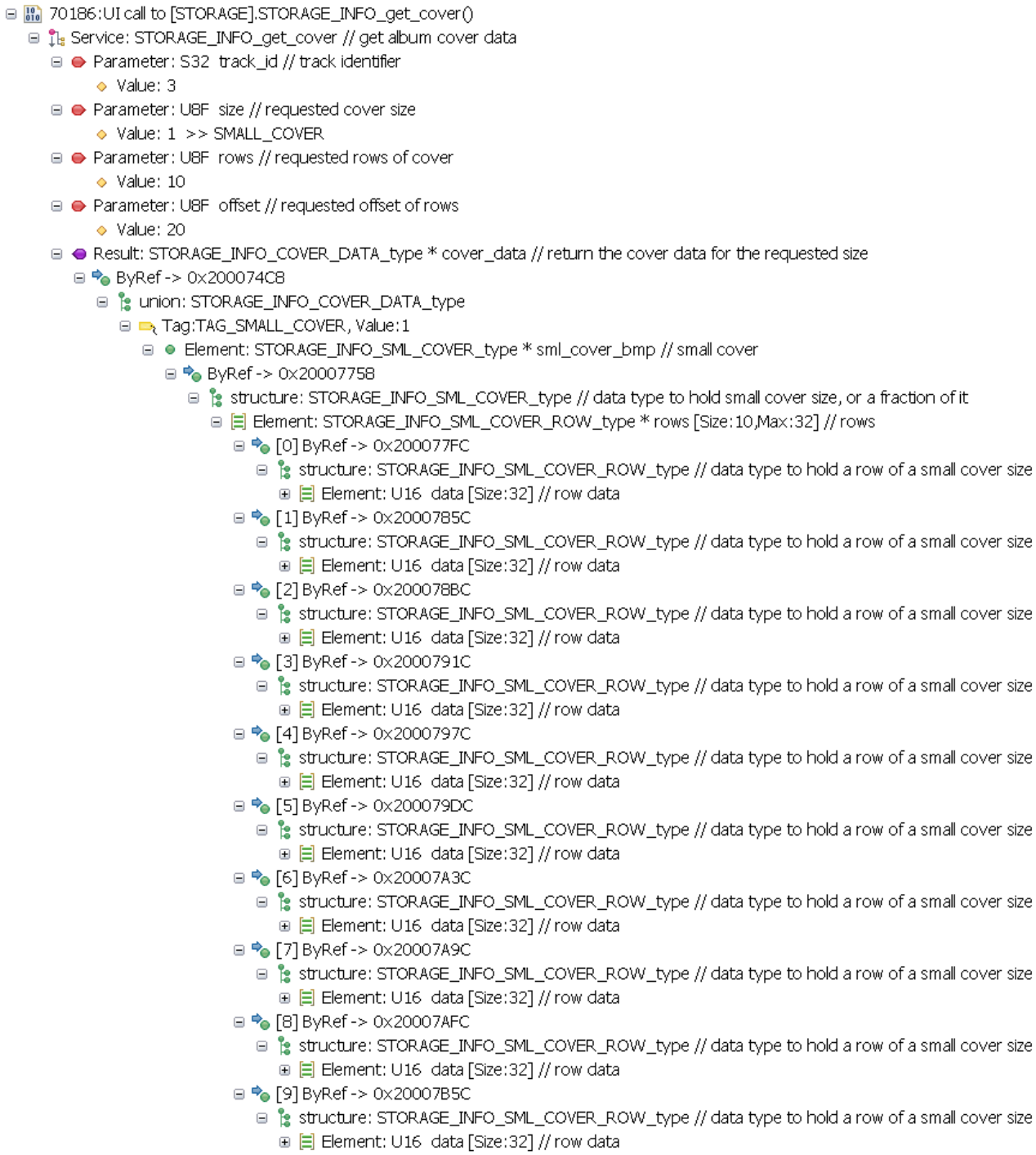

Figure 4 Decoded Service Call Example

The generator mbGen will generate all necessary tables and structures that are needed to support the inter-process communication via service calls. In addition, the programmer will get a framework

of template files that he/she could use to fill in the actual code to bring these services to life for a running system. From there on inter-process communication is as simple as a standard function call.

Using the IDL Editor mbEdit

The embenatics tool suite supports the developer of interface descriptions by providing a dedicated editor for that purpose called mbEdit. With mbEdit projects can be created to structure the use and visibility of MBID files in a system. Syntax highlighting as well as context-sensitive editing support and text completion assist the user to define data structures, types and services. All items of an MBID are grouped in different tabs for a better overview and structure of the document. MBIDs that belong to the same project provide their declared types and definitions for reuse while defining new interface descriptions.
Below is a screenshot of mbEdit showing the editing process for the MP3 Player example. In the upper left corner is the project explorer section, showing all projects in the workspace and the files that are part of that project. To the right is the editing space with the numerous tabs for the different elements of an MBID file. The setting shows the context-sensitive text completion dialog, where the user can select one of several provided suggestions. The lower left shows an overview of all elements contained in the currently active MBID file.

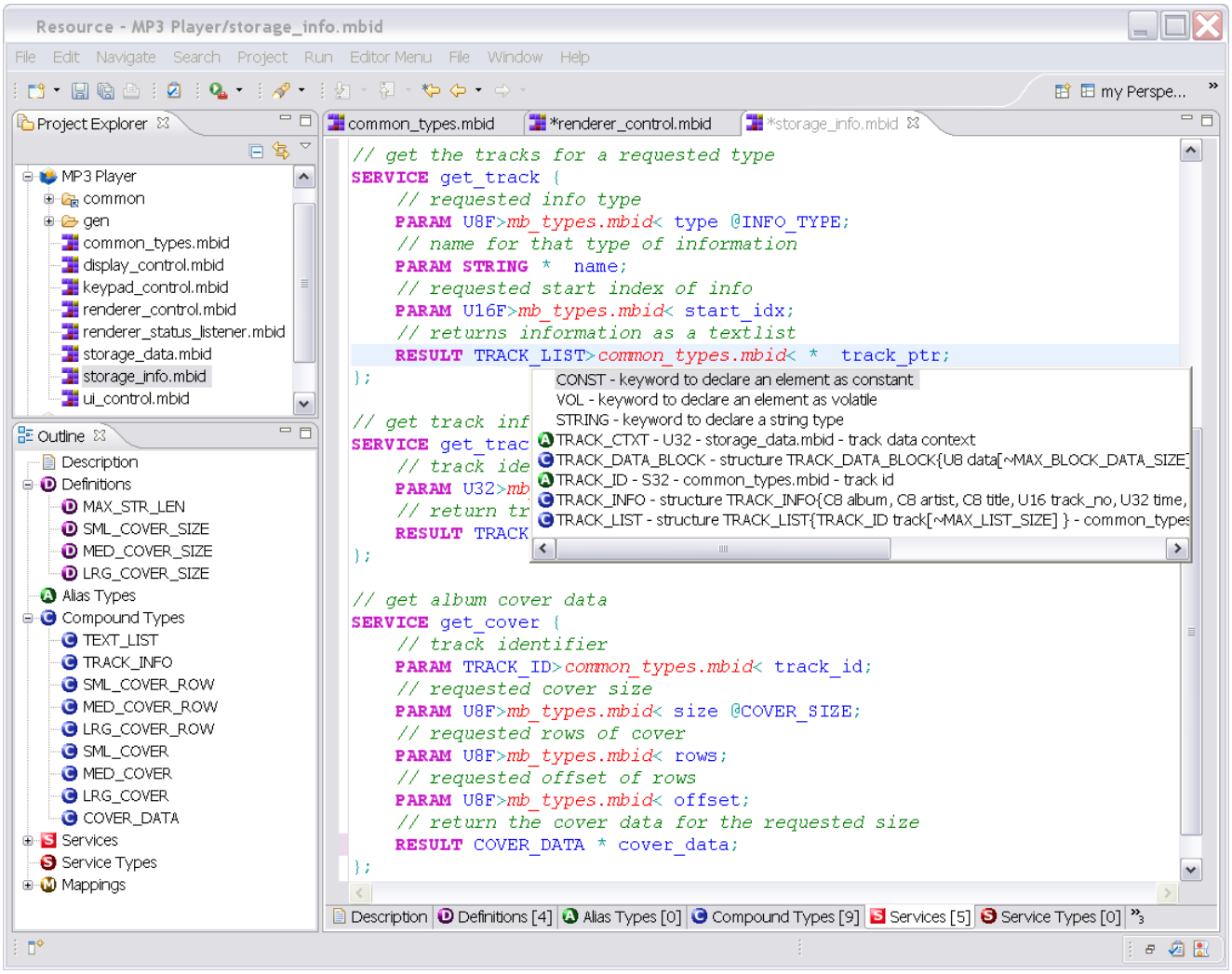

Figure 5 mbEdit Screenshot

Besides the interface description editing mbEdit also supports the system description editing, which is described in more details in another publication of this series of whitepapers [embenatics System Description].

Conclusion

Interface descriptions are the basis for inter-process communication when using the embenatics foundation layer software. The documents, so called MBIDs, are used to declare definitions, data types and services, by using a dedicated interface description language IDL. Based on the set of MBID documents that define the communication interfaces of a system or subsystem, the generator tool mbGen will generate all required output documents, like header and stub files, as well as tables and system code, that is needed to fulfil the needs of seamless inter-process communications. Besides the communication aspect, data types and structures can also be defined for internal data hosting objectives with the benefit to tracing and accessing such types from outside the system.

The developer will be supported by the embenatics tool suite that provides a powerful editor for interface and system description documents, as well as the target logger tool mbLog for diagnostic and test purposes of the running system.

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