

mbLay Logging

This white paper gives an introduction into the logging capabilities of the mbLay foundation layer and shows how the logging events are presented by the logging tool mbLog.

During development and test of software for embedded systems it is helpful to monitor the activities of the running system. Debuggers, that can be connected via JTAG to the target CPU, are quite useful for special debugging purposes, but they are not always capable of showing the program flow. In addition, they are expensive and not always independent of the used CPU type. The mbLay foundation layer offers a debugging interface which can be connected to a PC running the logging tool mbLog. The connection between target and PC might be a simple serial line but it can also be based on TCP/IP, which permits greater distance between the logging PC and the target system.

The logging tool on the PC is called mbLog and is described in another white paper [[mbLog Logging and Diagnosis Tool](#)]. In brief, it allows to collect, decode, filter and display logs from the foundation layer based software components in the target system.

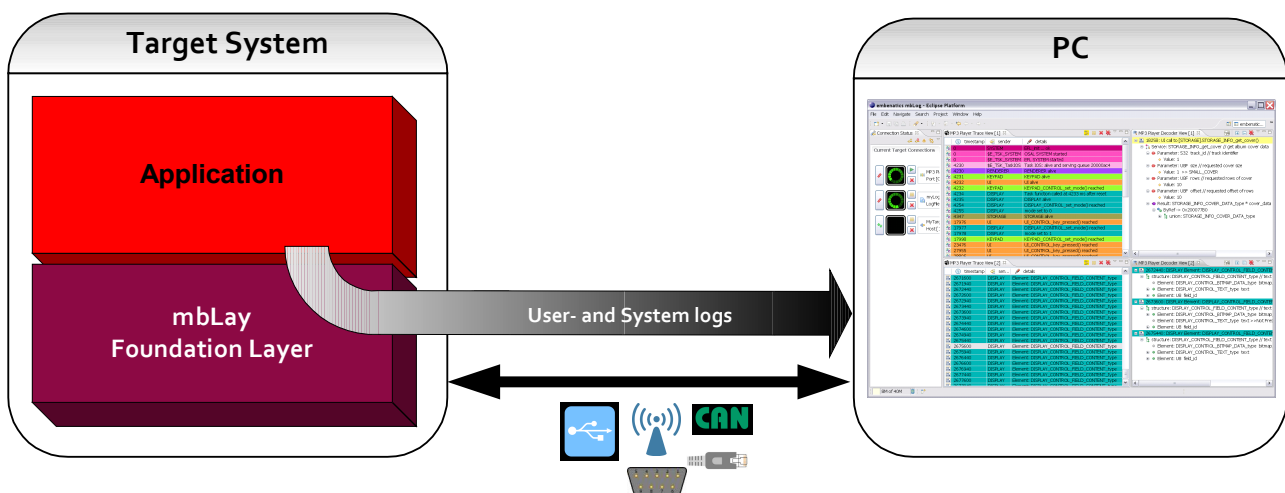


Figure 1 Logging of the Target System using mbLog

The communication activity and statistical data concerning the system resources are automatically logged by the foundation layer itself. In addition, application developers can instrument the code thanks to the logging API of mbLay. This allows to explicitly add logging events to be sent to the mbLog logging tool wherever useful in the application. These events can be utilized to help analyzing the program flow, data contents and the inter-process communication. The different kinds of logs can be distinguished by the logging tool because of different classifications. Filters can

be applied to the logging classes at runtime. An already recorded trace can be filtered by mbLog as well in order to streamline the trace for an efficient analysis.

Due to the physical limitation of the transport media bandwidth and a limited storage capacity in the target, logging might hinder threads from execution in real time. Therefore, the logging behaviour can be configured on a per thread basis. Either the logging amount of a thread can be reduced by disabling certain logging classes or the logging behaviour can be configured to abandon logs which cannot be sent in time. In the latter case, the logging tool will inform the user about the number of dropped logging events.

The blocking behaviour and the logging class setting can be dynamically changed at run-time by commands sent from the logging tool mbLog.

Instrumentation of the Code

The following section introduces the different logging event classes which can be used by the application developers in their code.

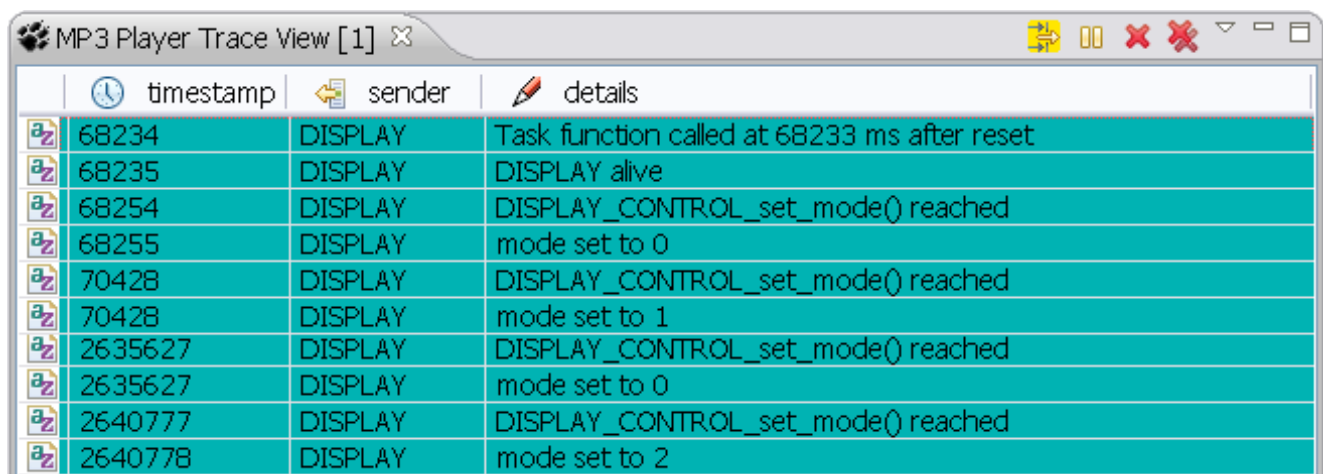
Logging classes POI (Point of Interest) and DEBUG

In order to allow filtering, these two levels of importance are available for logging. The log is added by just applying these set of macros which can be used in analogy to standard `printf` function. The macro postfix of `'_Pn'` is used to support a variable parameter list. If needed, additional classes can be added.

```
EFL_LOG_POI_Pn ("Task function called at %d ms after reset", EFL_get_time());
EFL_LOG_POI ("DISPLAY alive");
```

```
EFL_LOG_DEBUG ("DISPLAY_CONTROL_set_mode() reached");
EFL_LOG_DEBUG_Pn ("mode set to %d", mode);
```

Calling these macros with the shown parameters results in the following output of mbLog:



	timestamp	sender	details
	68234	DISPLAY	Task function called at 68233 ms after reset
	68235	DISPLAY	DISPLAY alive
	68254	DISPLAY	DISPLAY_CONTROL_set_mode() reached
	68255	DISPLAY	mode set to 0
	70428	DISPLAY	DISPLAY_CONTROL_set_mode() reached
	70428	DISPLAY	mode set to 1
	2635627	DISPLAY	DISPLAY_CONTROL_set_mode() reached
	2635627	DISPLAY	mode set to 0
	2640777	DISPLAY	DISPLAY_CONTROL_set_mode() reached
	2640778	DISPLAY	mode set to 2

Figure 2 Using DEBUG and POI Logging

Logging class ELEMENT

Sometimes it becomes necessary to log the content of a variable. This could be a basic-type variable or even a complex data structure containing pointers to other data structures in memory. With this logging class you have an elegant and efficient way to log these data structures and to visualize them in mbLog. For this purpose you just need to specify the targeted data structures in a description file using a C-like notation. During this task you are fully supported by the embenatics editor tool mbEdit. You will find a description of this tool in a separate white paper [[embenatics Interface Description](#)]. In addition, you can trace all data structures, which have already been described in the context of inter-process communication, by simply using the following macro in your code:

```
EFL_LOG_ELEMENT(TID_DISPLAY_CONTROL_FIELD_CONTENT_TYPE, display_content);
```

The first parameter `TID_DISPLAY_CONTROL_FIELD_CONTENT_TYPE` identifies the data type of the variable passed as second parameter `display_content`. The identifier and the data type definition is automatically created out of the interface description by the generator mbGen, which is part of the embenatics tool chain.

```
#define TID_DISPLAY_CONTROL_FIELD_CONTENT_TYPE 12

/*-----
| text and/or bitmap to be displayed in specified field of display
|-----*/
typedef struct
{
    DISPLAY_CONTROL_BITMAP_DATA_type bitmap_data; //
    DISPLAY_CONTROL_TEXT_type text;
    U8 field_id;
    /* optional element control starts here */
    U8 bitmap_data_v_ :1;
    U8 text_v_ :1;
} DISPLAY_CONTROL_FIELD_CONTENT_type;
```

Calling the macro with the variable of this type results in the following output in mbLog:

	timestamp	sender	details
10 010	126949	DISPLAY	Element: DISPLAY_CONTROL_FIELD_CONTENT_type
10 010	126980	DISPLAY	Element: DISPLAY_CONTROL_FIELD_CONTENT_type
10 010	128000	DISPLAY	Element: DISPLAY_CONTROL_FIELD_CONTENT_type
10 010	128049	DISPLAY	Element: DISPLAY_CONTROL_FIELD_CONTENT_type
10 010	128098	DISPLAY	Element: DISPLAY_CONTROL_FIELD_CONTENT_type
10 010	128226	DISPLAY	Element: DISPLAY_CONTROL_FIELD_CONTENT_type
10 010	128410	DISPLAY	Element: DISPLAY_CONTROL_FIELD_CONTENT_type

Figure 3 Element log appears in the Trace View

Each of these logging events can be expanded in order to see the contents of the data elements:

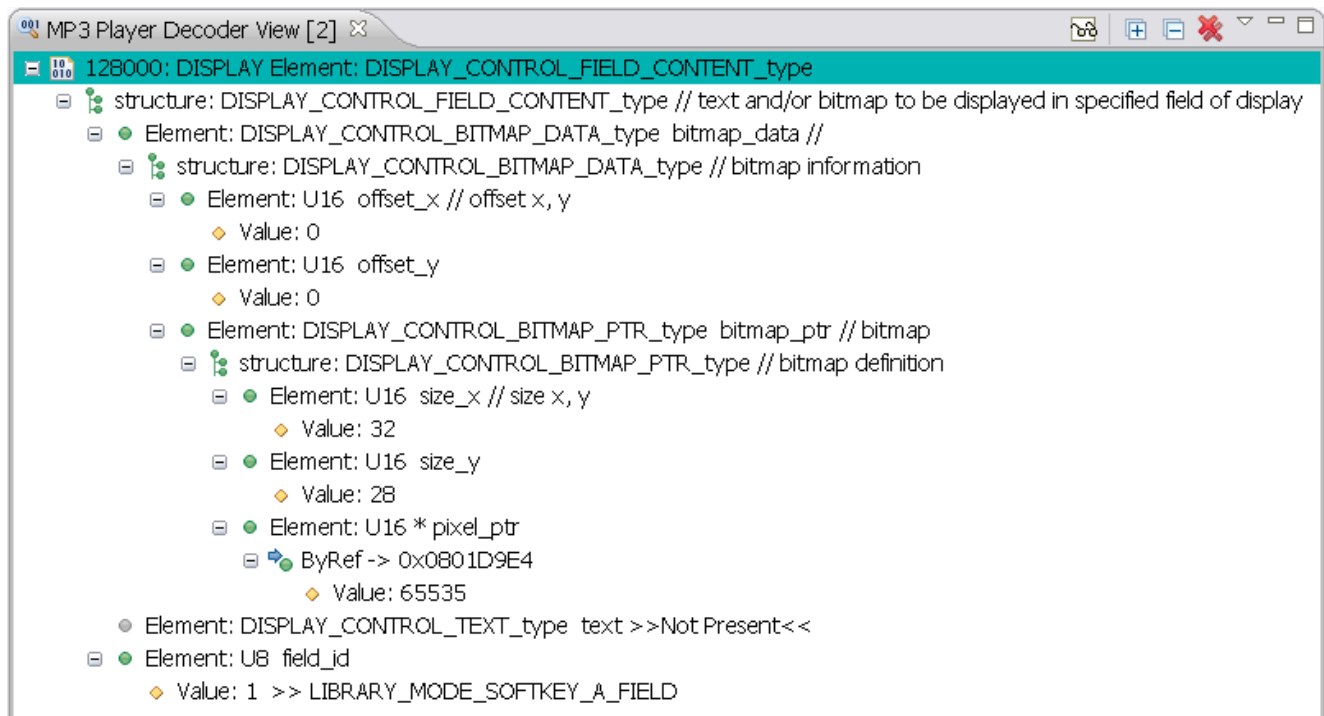


Figure 4 Decoded Element log of `display_content`

In the Decoder View in Figure 4 you see the tree representation of the logged data structure. Besides the values, all comments and mappings from the interface description are displayed in order to support you in analysing the data. Compared to a simple hex-dump, you might imagine how helpful this will be.

Communication Logs

In the last sections, examples are shown which demonstrate how logging commands can be used to instrument the application code. In order to show the built-in logging capabilities of the foundation layer mbLay, an example of the automatic communication log is given here. The communication model of the foundation layer is described in the [[mbLay – Inter-Process Communication Model](#)] white paper. The following example is derived from the MP3 player application which is used in this series of white papers.

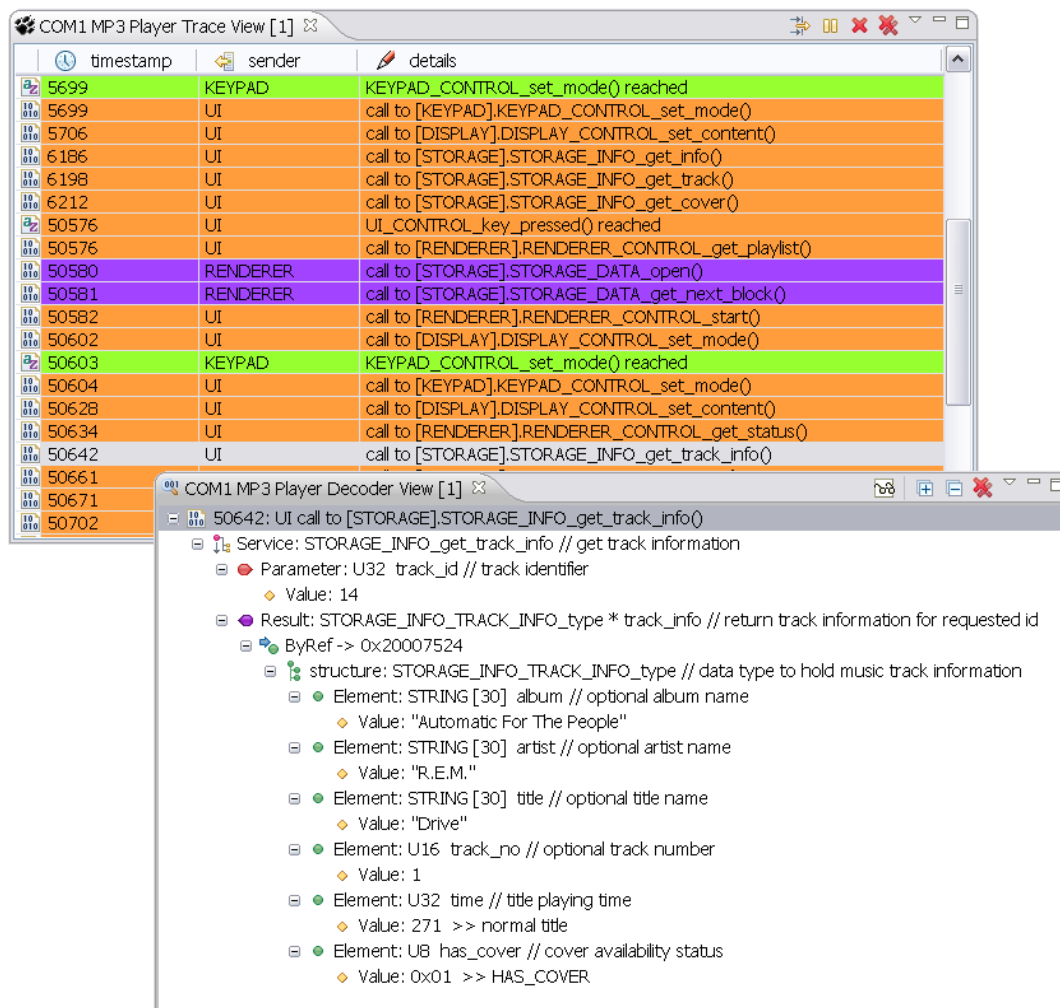


Figure 5 Logging the Remote Procedure Call of the Service `get_track_info`

The above displays a snapshot of the communication in the “Now Playing” dialog. If enabled in the filter settings, this trace is automatically generated each time a RPC communication between threads is completed. The information given by the Decoder View in the bottom window is very similar to the element decoding in Figure 4. In addition, the name of the called service, the result and all defined parameters which belong to this service are shown.

Conclusion

The foundation layer mbLay provides a debug interface that allows the connection of the logging tool mbLog. Via this interface, mbLay is able to provide logging information automatically about the inter-process communication and the system resource status. In addition, an API is available that allow to instrument the code with additional logging events. These events can be utilized to log simple printouts but also to dump out complex data structures. Logs can be filtered in order to reduce the bandwidth of the debug interface media. The powerful PC based tool mbLog is available to collect and display these control and data flow logs of the running system for diagnostic and test purposes.

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