Extended Abstract

EMBEDDED LOGGING FRAMEWORK FOR SPACECRAFTS

Frank Dannemann
Institute of Space Systems
German Aerospace Center
frank.dannemann@dlr.de

Prof. Dr. Sergio Montenegro
Aerospace Information Technology
University of Würzburg
montenegro@informatik.uni-wuerzburg.de

Abstract

During the different development phases of a spacecraft there exist various kind of information worlds, which often hold the same kind of content. In this paper we present an approach that can be seen as a first step to combine these worlds. An embedded software framework is introduced, which takes over the functionality of the internal debugging features of the system on the one hand, and on the other hand is a possible candidate for substituting the system's telemetry application, which is responsible for sending spacecraft status information to the ground station. Taking a minimal implementation of the framework, it's advantages by using it in the context of a satellite's console-based debug statements mainly needed for integration and testing purposes are shown in a practical use case.

1 INTRODUCTION

Taking it to the extreme, there are at least two information worlds on board of a satellite which coexist with no interchange of information at all, although there's great intersection of the content that has to be provided. On the one hand there the debugging world of the embedded software developer, who's in charge of developing the software for the boot image of the satellite, while on the other hand there's the "official" telemetry world, holding the satellite status information to be send to the groundstation each time the satellite is in contact with the antennas on earth.
This situation has several disadvantages, some of them are quite crucial:

EFFORT OVERHEAD The same kind of information has to be provided to both information worlds which is an unnecessary overhead.

ERROR PRONE In case of a change both information worlds have to be adapted.

MEMORY CONSUMPTION By holding redundant information not only the the size of the source code and boot image will be larger than necessary, also the memory consumption at execution time will be bigger then in case of having only one single framework for information processing.

PROCESSOR CONSUMPTION The execution and formatting of the unnecessary `PRINTF` statements on board of the satellite consumes processor time and load, even though they could safely be deactivated after launch, because none of the message to be printed on a console can be displayed anymore.

The solution of getting rid of the unnecessary `PRINTF` statements would be their removal in the final boot image of the OBC. But nobody dares to do this, because of three major reasons:

1. it is too much effort
2. the behaviour after removal is not predictable
3. the tested and maybe certified boot image will loose this status (additional testing and certification may be necessary)

Our approach to this problem is to combine these two information worlds and introduce a framework which is based on the idea of using information sources and sinks and decoupling them, meaning the separation of the functionalities (1) logging of information, (2) sending them to a sink and (3) displaying them. Like displayed in figure 1, this replacement of the former PRINTF-debugging and logging functionality is achieved by introducing new components like *Logger*, *Appender* and *Layout*. Providing a minimal

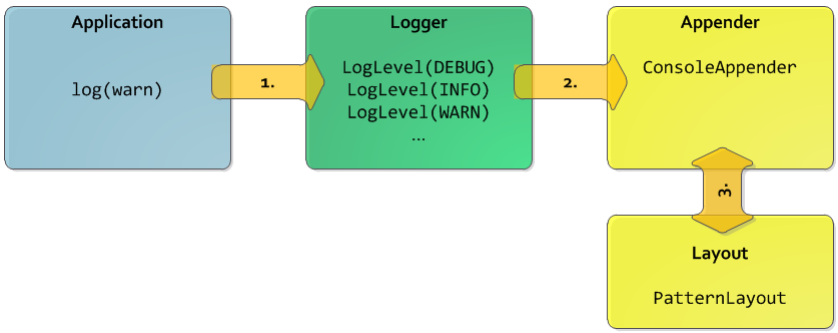

Figure 1: Separation of data flow

working example, an exemplary *ConsoleAppender* has been combined with a *PatternLayout* providing the engineer with a similar output to PRINTF, yet being much more flexible and getting all the additional advantages the logging framework has to offer. Within this paper we present our first results, using the real-time operating system RODOS as a testbed for development and testing. The advantages of using the framework firstly as a replacement of the console based debug messages are explained and shown in a real use case.

2 RODOS

RODOS is the acronym for Realtime Object-Oriented Distributed Operating System. As an embedded operating system it is specially designed for space applications, but fits perfectly to all applications demanding high dependability. The RODOS real-time kernel and middleware provide an integrated object-oriented framework to multitasking resource management and to network-based communication infrastructure. Although targeting minimal complexity, no fundamental functionality is missing, as its microkernel provides support for resource management, thread synchronization and communication, input/output, and interrupts management. The system is fully preemptive and uses priority-based scheduling and round robin for threads sharing the same priority level. RODOS is written mainly in the C++ programming language, some hardware dependent parts are written in C and target specific assembly language. Despite RODOS is intended for stand alone use in embedded systems, the user can also run it on-top of Linux as guest OS. It is built as a static library, so that a user can link his code against this library and run the resulting binary.
The overview of the RODOS architecture is shown in picture 2. Besides the HDL, core and management layers RODOS also comes with an (optional) middleware. The users main interface is therefore the application module, which encloses one or more threads. In each application module, the programmer can create an application object that defines an application name

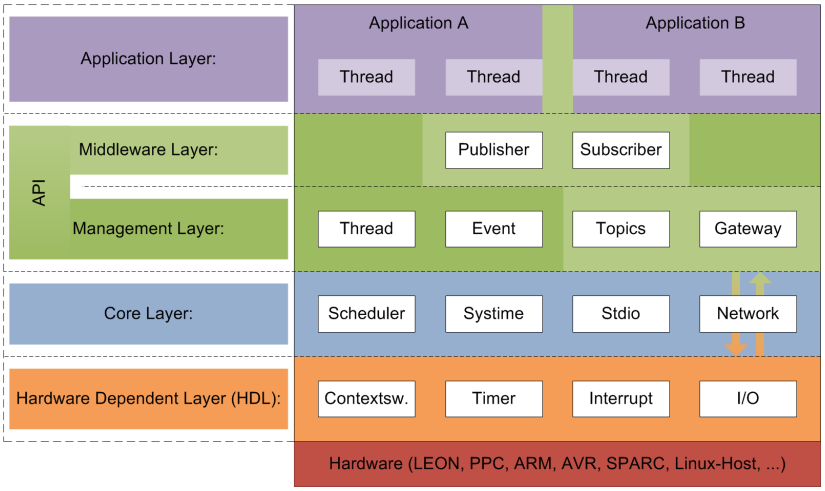

Figure 2: RODOS Layers

and an identification number. By using the publish/subscribe mechanism from the middleware it is easily possible not only to establish an inter-process communication between applications, but also to communicate with several RODOS computing nodes over the network (Ethernet (TCP/IP), CAN communication) using gateways.
A detailed design description of RODOS can be found in [2]. RODOS is Open-Source under the BSD-license and can be obtained through it's SourceForge[1] or DLR-websites[2].

3 RODOS-DEBUGGING WITH PRINTF

Referring to figure 2, there are two kinds of users who are using the debugging-technique using print-statements for the console. First, there is the RODOS user, who wants to use the RTOS (RealTime Operating System) in order to operate the onboard computer of his specific mission. He is only working in the application layer, dealing with the RODOS API from the middleware- and management-layers. Debugging with `PRINTF` will help him finding errors during the time of developing and testing his applications and will finally end up with an error-free boot image for the onboard computer. Second, the RODOS developer itself makes extensive usage of the `PRINTF`-debugging capabilities while working on all layers that are located underneath the application layer. Though both types of developers gain benefit from using the monitoring framework presented here, within this paper we focus on the needs of the application developer, meaning the user of the RTOS.

Debugging with print-Statements inside the source code of a software program is often referred to as *printf-debugging* [1]. The name arises from the `printf`-statement in the C programming language, but similar statements can be found in all major high-level programming languages used for embedded systems. As RODOS is developed mainly in C++, it's internal debugging statement is also called `PRINTF`[3]. The main intention behind the usage of these kind of statements is to track the control and data flow during the actual execution of the code. This is done by adding `PRINTF`-statements directly in the code of the applications. For this purpose RODOS offers debug methods like `PRINTF` and `ERROR`. A shown in picture 3, these methods can be used by application developers by simply including debug.h in their application. In this typical debugging scenario, the developer starts a console

[1] http://sourceforge.net/projects/rodos
[2] http://www.dlr.de/rodos
[3] `cerr` or `cout` together with their operators << and >> would have been also a possible solution

on his host computer, establishes a serial connection to the target platform (here: the onboard computer of a satellite) and is afterwards able to watch the PRINTF-output on the terminal using a standard UART-interface.

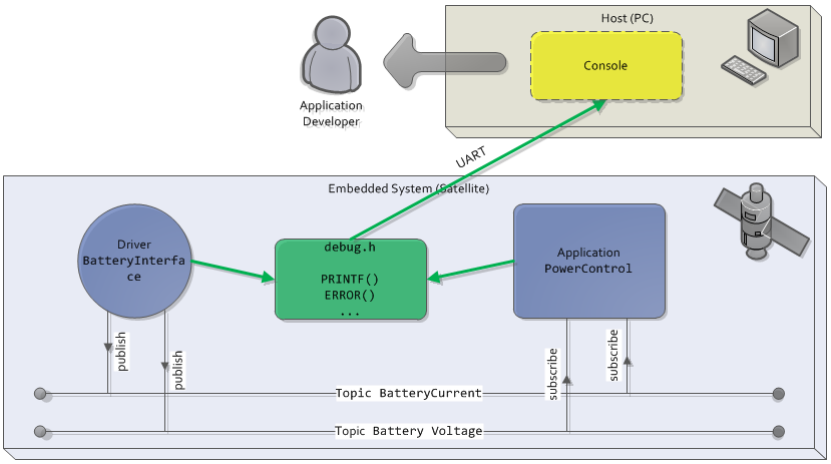

Figure 3: RODOS debugging using PRINTF statements

4 THE NEW MONITORING FRAMEWORK

When designing the new monitoring framework, there where several requirements which had to be taken into account. Some of the most important ones are listed here:

SIMPLE & FAMILIAR USAGE The new framework should be as simple to use as the familiar PRINTF- and ERROR-methods.

RODOS INTEGRATION The technique behind the new logging mechanism shall utilize the RODOS middleware, which is the tool of choice for all kinds of inter-process communication inside the RODOS framework.

SIMPLE & APPROVED DESIGN The framework design shall be oriented towards the architecture of the *log4j* logging framework[4] , which is widely used for java-based desktop applications. The design of log4j is very simple and similar to what is depicted in figure 1, nevertheless it's a powerful and mature logging framework, offering features like classifying messages by using categories and filter them accordingly at runtime.

LOW MEMORY & POWER CONSUMPTION The integration of the logging framework into RODOS shall result in a minimum increase of resource consumption.

DIVIDING INPUT FROM OUTPUT Due to the resource requirement mentioned above, the user has to be able to deactivate the message output (sink) at runtime.

VARIABLE OUTPUT Output shall be send to various message sinks: console, file, network, ...

Using the RODOS middleware it shall be possible to send logging information over the network to any other RODOS instance using gateways, like depicted in figure 4. In this example, formatting of the received messages on the host computer is done by the Appender- and Layout-modules. They can be configured in a way, that – if requested – the console output is similar to what is produced by the PRINTF-methods. But any other output is also possible. The first minimal implementation of the framework was developed and tested within the scope of a diploma thesis [3].

[4] http://logging.apache.org/log4j

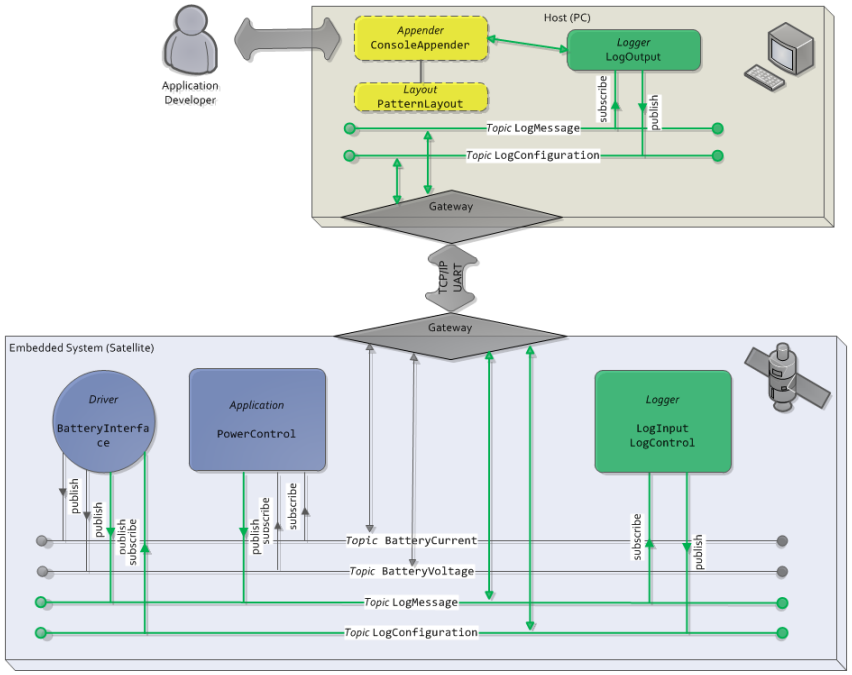

Figure 4: RODOS debugging using the Logging Framework

5 CONCLUSIONS & OUTLOOK

Though further testing on target hardware platforms (especially LEON-processor based boards) is necessary in order to evaluate the performance of the logging framework in an embedded system, the results achieved so far are very promising. Using the middleware communication capabilities of the RODOS RTOS, the framework integrates nicely into the embedded system software. The application developer has a simple-to-use interface at hand, which can be configured in way that e.g. debug messages are displayed just as the familiar print-statements. On the other hand, powerful features are added to system: message transport over network boundaries, flexible output configuration even at runtime, deactivation at compile- and runtime – just to mention a few.
The further development will focus – on the one hand – on expanding the current functionality of the framework, like adding additional layouts for other user groups (e.g. integration or mission control engineers) or creating new appenders (e.g. file appender). On the other hand, we are also thinking about new architectural changes. The most important ones are the usage of the logging framework as a replacement of the housekeeping functionality of a spacecraft, and the integration of the framework into a RTOS-independent middleware.

REFERENCES

[1] J.H.M. Dassen and I.G. Sprinkhuizen-Kuyper. *Debugging C and C++ Code in a Unix environment*, chapter Debugging techniques. 1999.

[2] Sergio Montenegro and Frank Dannemann. RODOS - Real Time Kernel Design for Dependability. In *DASIA (DAta Systems In Aerospace)*, 2009.

[3] Sven Müller. Entwicklung eines Rahmenwerks zur Nachrichtenprotokollierung für das eingebettete Echzeitbetriebssystem RODOS. Master's thesis, Universität Oldenburg, 2012.