Plug-in Based Debugging For Embedded Systems

Shahabeddin Farokhzad[1], Gokhan Tanyeri[2], Trish Messiter[2], Paul Beckett[1],
[1] *RMIT University,* [2]*Clarinox Technologies Pty. Ltd.*
s3175161@student.rmit.edu.au, gokhan@clarinox.com

Abstract

A flexible, plug-in based debugger is described. The debug system, built as a C++ interface class, is independent of the physical layer, which can be a network, a serial connection (e.g., RS232), or even a file on hard disk or flash memory. The plug-in mechanism is described and an example presented of how these are written to fit into the debugger environment.

1. Introduction

It is little surprise that the single most requested area of improvement in embedded design support is debugging tools [1]. Debugging has always been amongst the biggest concerns of designers and embedded systems are becoming increasingly complex and have unique constraints which make them hard to debug. As test and debug continues to consume the largest slice of the development and maintenance cycle [2], new approaches to debugging are required.

In contrast to the general purpose computer software design environment, a primary characteristic of embedded environments is the sheer number of different platforms available to the developers (CPU architectures, vendors, operating systems and their variants). Embedded systems are, by definition, not general-purpose designs: they are typically developed for a single task (or small range of tasks), and the platform is chosen specifically to optimize that application. Not only does this fact make life tough for embedded system developers, it also makes debugging and testing of these systems harder as well, since different debugging tools are needed in different platforms.

Put simply, embedded systems debuggers have two key requirements:

1. to identify and fix bugs in the system (e.g. logical or synchronization problems in the code, or a design error in the hardware);
2. to collect information about the operating states of the system that may then be used to analyze the system: to find ways to boost its performance or to optimize other important characteristics (e.g. energy consumption, reliability, real-time response etc.).

The second requirements plays a greater role in embedded system design than it does in the general purpose domain as the resources accessible by the embedded system developers are far more constrained. It is likely that the designer will have to balance a complex interrelationship between competing considerations such as low power, robustness, small size and weight, real-time requirements, long life cycle, cost and low (or zero) tolerance for malfunctions. As a result, it is often not straightforward to determine all of the resources needed at the planning stage or even at the design stage of the development procedure. Part of design improvement procedure is always done during the testing/debugging stage. It is important to take advantage of state-of-the-art tools to facilitate this procedure.

In this paper, we describe a flexible, plug-in based debugger developed to address these requirements. We will describe the plug-in mechanism and how these are written to fit into the debugger environment. The approach presented here was conceived and implemented by Clarinox Technologies [3] to overcome the major barriers to fast and efficient debugging of embedded systems and is based upon many years of experience in the embedded industry. The key features of the debugger are:

- it is built as a C++ interface class that is independent of the physical layer. The physical layer can be a network, a serial connection (e.g., RS232), or even a file on hard disk or flash memory;
- a robust DLL-based plug-in interface for the debugger enables developers to add their own debugging functionality to the base environment. A plug-in can receive and parse special messages sent by the debug target and can maintain its own window and thread context.

To date, the technique has been used to develop several blocks as part of the standard deployment of the debugger, including protocol monitor for Bluetooth, GPRS and RFID, a memory analyzer and symbol loader.

The remainder of this paper proceeds as follows. In Section 2, conventional approaches to embedded debuggers are examined, and in particular how these compare to our approach. Section 3 briefly describes the overall debugger environment, including its typical physical layer interface. In Section 4, we also introduce the concept of plug-in based debugging, and how plug-in modules are written for the system. We illustrate this using an example plug-in. Finally, Section 5 concluded the paper and comments briefly on future work.

2. Debugging Embedded Systems

A good debugger and profiler are essential in the embedded domain in which applications need to operate for long periods of time. Debug tools exist to make the embedded system controllable and observable at a number of levels, including assembly and source-level, syntax level and in-circuit. The typical debugging process starts on the host machine using emulation and operates at increasingly more realistic levels [4]. Traditionally the final level would involve special purpose test hardware incorporated into the target device. However, the increasing need for hardware/software co-design along with vastly shorter development cycles makes this In-circuit Emulation (ICE) approach less viable. This has led to the development of software-only approaches (e.g., [5]) in which platform-independent components of an application execute natively on the host, while the platform dependent sections are run using a tightly coupled instruction set simulator for the target processor. As the host typically exhibits significantly higher (raw) performance than an embedded target processor, the overheads of this approach can be relatively small.

Additional difficulties arise when developing heterogeneous, distributed and resource constrained systems. Unpredictable and obscure errors can arise due to the complex interactions between concurrent software (i.e., the application, operating system and any middleware support) and the hardware platform and its interconnecting networks. In [6], this problem is addressed by porting the operating system to a virtual machine monitor (VMM), thereby presenting a number of useful the hardware and software abstractions to the debug system. Running each component of the application in its own virtual machine offers the designed greater control over the aspects of the application and its environment.

However, it is unclear what direct effect this type of virtualization process has on the interactions it is trying to reveal, given that embedded systems tend to operate very close to their maximum resource limits. This is especially true in highly distributed systems such as wireless sensor networks. The multi-hop, resource-constrained, and timing dependent nature of these networks, mandates an approach that minimizes its IO latency memory overhead. Various approaches have been proposed. For example, the *Emstar* environment [7] uses a multi-process service model that sacrifices some level of performance in return for increased robustness in debugging heterogeneous embedded sensor-actuator networks. The *Clairvoyant* tool [8] tries to maintain a high level of performance by embedding debugging commands into the target binary (called dynamic binary instrumentation). Although this approach succeeds in running the program at native speed directly on the hardware without using extra hardware or making changes to the program's source code, issues such as limited memory and flash lifetime severely limit the resources available for debugging on the actual sensor nodes. The profiling system proposed in [9] uses a hybrid hardware and software approach based on dynamic instruction stream editing [10] to sample the executing instruction stream.

While the details of the actual debug mechanism will vary widely across the range of specific embedded targets, the process almost invariably involves remote tracing of the program execution, watching variables and dumping data to a host computer console. The JavaES framework [11] consists of a set of tools targeting JAVA applications that allow the internal state of the JRE to be monitored in real-time including digital and analog input-output values, free memory, number of active threads, timers, etc. In addition, the framework supports the remote IO values modifications and the redirection of the standard outputs for remote reading. Similarly, the Java based Avrora framework [12] includes a model that emulates each target device and supports interaction with the application code. All of these software emulation approaches rely on the speed of the host processor to preserve the timing and synchronization behavior of the target application.

3. Plug-in Based Debugging:

Although all of the tools identified in §2, above, are found in common use, they exhibit various drawbacks that can make them less useful in particular contexts. For example, debug tools tend to be platform specific and can require significant effort to port the design to a new platform, with its dedicated environment and user interface. Some of these tools require specific debug-related code to be inserted into the application code, further complicating the porting process.

Further, all debug tools necessarily produce a huge amount of data that is difficult and error prone to analyze by hand. However, any attempt to filter and categorize the data may serve to hide valuable information, especially when the analysis is spread across a number of separate tools. Tools that operate separately and independently may not effectively capture important time-related and synchronization information. Finally, developers may be forced to write their own debugging tools that suite their well-established design and development environment (e.g. a proprietary event-based debugger, or protocol analyzer etc.). However, apart from the obvious difficulties with writing a new tool from scratch, the time needed to develop a new debugging tool may be not justifiable if the tool is disposable in nature (i.e., when it is needed for one design only). The temptation is strong in this case to stay with existing design techniques and environment, even if it is marginally appropriate and can increase debug costs in the longer term.

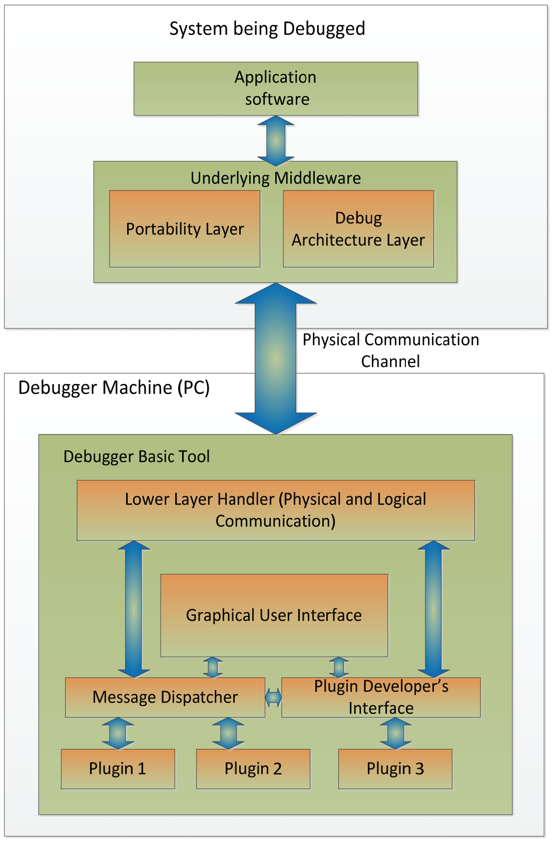

Figure 1. **Plug-in based Debugging Framework**

In order to overcome these issues, we propose a debugging architecture based on plug-in modules. The overall structure (Figure 1) is a unified debugging tool which provides the fundamental functionality to which plug-in modules are attached to provide specific capabilities. The architecture provides a channel that handles all of the underlying communications required between the debugger and system being debugged, including any signal or protocol translation. As the debugger typically runs on a PC, these two are generally located in different environments.

The channel infrastructure supports any form of physical communication (e.g. UART, wired or wireless network, embedded busses, etc.) and any form of logical communication (e.g. full-duplex, binary data transmission, packet-based data transmission). The architecture also provides for both online and offline debugging. In the offline debugging scenario, data are stored locally (e.g. in hard disk or flash memory), and then transferred to the debugger machine for analysis. This facility is particularly useful for small embedded systems with no means of external communication.

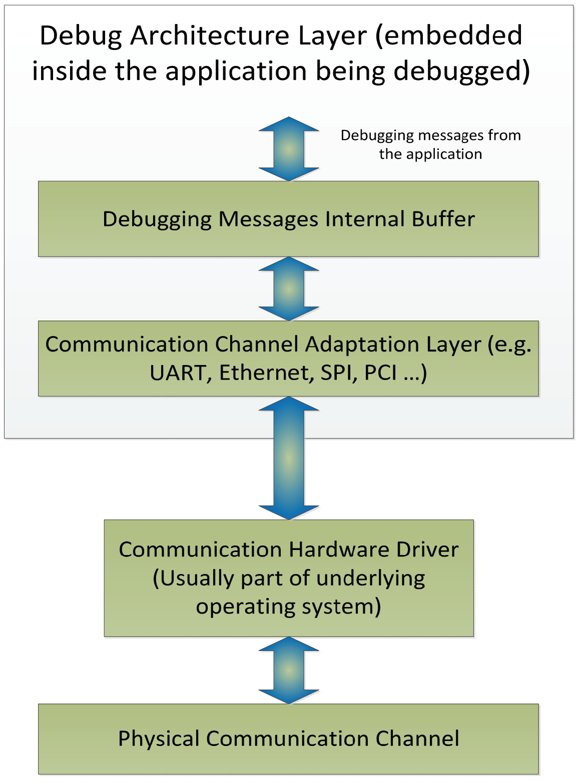
Debug Architecture Layer (embedded inside the application being debugged)
Debugging messages from the application
Debugging Messages Internal Buffer
Communication Channel Adaptation Layer (e.g. UART, Ethernet, SPI, PCI …)
Communication Hardware Driver (Usually part of underlying operating system)
Physical Communication Channel

Figure 2. **Debug Architecture Layer**

The overall structure is based on middleware that encapsulates and hides the underlying platform differences, and provides a common interface to the higher-layer application developers. This approach ensures that the architecture can operate in different platforms without a need for modification. The debug architecture receives and analyzes debug data from the target and dispatches them to the appropriate plug-in tool. The plug-in developer does not need to know anything about how the underlying architecture works. The Plug-in modules install handlers which are called when the related data are received. They have to analyze the data, and (if necessary) return a report to the main tool.

The architecture provides a simple and common interface to the plug-in modules. Modules are able to communicate with the central platform, with other modules or with the embedded system being debugged, thereby avoiding the need to provide special interfaces between proprietary debug tools. The interface must be very simple to the extent that it will justify any attempt to develop a special plug-in tool, even a disposable one.

Because all data come from the same communication channel and are then dispatched to the different plug-in modules, the data generation order is preserved. This order provides extremely important information on the effectiveness of parallelism in the code and reveals synchronization issues. The underlying architecture provides means for the user to take advantage of information hidden in the order. Furthermore, the architecture provides categorization and filtering facilities to the plug-in developers, allowing developers to be able to filter the data based on order.

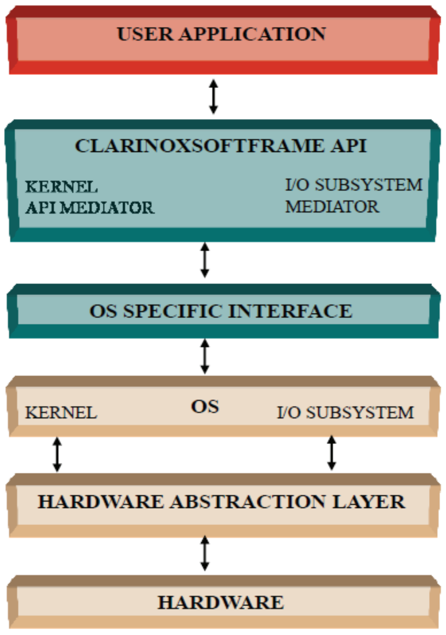

Figure 3. **The Softframe® API**

A substantial part of the Graphical User Interface is completely handled by the architecture, obviating the need fort the plug-in developer to become involved with GUI design tasks. On the other hand, the architecture permits developers to develop their own user interfaces and connect these to the tools interface, providing a very robust and powerful means of analysis report demonstration to the user while decreasing overall debugging time.

The debug code automatically compiles into the application as part of the middleware used. Although this frees the developer from having to insert it manually there may still be a need to define the debugging configuration parameters (e.g., what communication hardware to use, and its related parameters). Thus, the debugging technique is impractical for source-level and assembly level debugging (these debuggers must reside outside of and running in parallel with the application, so they can completely control the execution path of the code). But, given the fact that source-level debuggers are tightly coupled with the CPU and operating system on which the application runs and even the compiler used to generate the code, there is no sense in including source-level debugging as part of an architecture which means to be independent from the platform and be portable to any embedded environment. However, this architecture is suitable for all other types of debug tools.

The Debug Architecture Layer depicted in Figure 2 comprises the Communication Channel Adaptation Layer (CCAL) and an associated message buffer. The CCAL is a layer of software which provides a common logic communication interface to the upper layer and communicates with the related hardware driver on behalf of the application. The interface must be flexible enough to encapsulate and integrate different type of communication channels (e.g., there is a concept of "connection" and "disconnection" in some communication technologies, while absent in others technologies).

3.1 The Debugger Environment

This work is based on the debugging framework implemented as part of Clarinox *SoftFrame*® Middleware (Figure 3), which includes a GUI tool used to remotely debug SoftFrame® based embedded applications. The OS Wrapper includes functions such as threading, timers, semaphores, mutexes, dynamic memory management, inter-process message passing, event/message handling, finite state machine templates, serial device driver encapsulation, USB device driver and TCP/UDP Socket encapsulation. The framework extends the debugging tools and Board Support Package or Hardware Adaptation Layer provided by RTOS manufactures (e.g., the Intel SA-110, SA-1100, SA-1110, SA-120, SA-1500 evaluation boards) and offers debugging tools that can handle multi threaded applications that are not specific to only one environment.

3.2 Application/Debugger Message Passing

In the debugging framework as implemented, the basic tool is a Windows GUI application which is able to dynamically attach to plug-in DLLs, eliminating a need to recompile the tool for each new plug-in. From now on, this tool will be referred to as *the debugger*. The debugger is able to receive messages sent by an application and to analyze and format the messages in real time. The user can also send command messages to the application to control the debug process. Messages are categorized as follows:

1. *Thread related messages*, including thread control;
2. *Profiling and stack trace messages* describing the call and return history and profiling information;
3. *Informative (text) messages*, including logs, warning and fatal events.

4. *Memory messages* providing information on allocations, and de-allocations in the code;
5. *Protocol Stack Monitoring messages*, including monitors for Bluetooth, RFID, and GPRS. These messages identify all stack activity, revealing all low-level protocol data and messages in real time.

3.3 Physical Layer Interface

This section examines the architectural layer interface of the debugging framework. Any code which is written with the aid of the SoftFrame and compiled in Debug Mode automatically includes this layer which manages the debug buffers, sending them to debugger via an arbitrary physical layer (Ethernet, RS232, Bluetooth, File etc.).

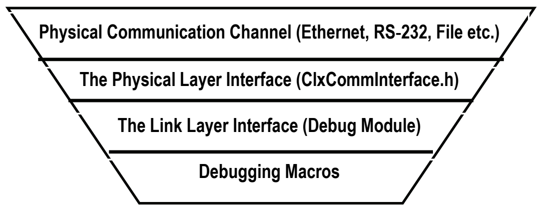
Physical Communication Channel (Ethernet, RS-232, File etc.)
The Physical Layer Interface (ClxCommInterface.h)
The Link Layer Interface (Debug Module)
Debugging Macros

Figure 4. **Basic Hierarchy of Debug Architecture Layer**

The debugging architecture (Figure 4) has the following structure:

- A number of C/C++ macros which obtain the debugging messages/data from the developer, and format them into the debugging buffer.
- A thread (created by the class DebugModule) manages the link layer of the debugging system. It waits for control messages from the debugger, and also sends the buffer contents to the debugger, either when the timeout expires or when the buffer is full, ensuring a maximum delay between when the debug message is generated and when it is parsed and shown on the debugger main screen.

4. Developing Plug-in Debugging Modules

This section explains how to develop a plug-in DLL for the debugger software.

4.1 Writing a Plug-in

The debugger receives text, and binary debugging messages sent by any program written using SoftFrame (in debug configuration). Text messages simply provide logging information, or specify errors which take place during the program execution. In contrast, binary messages cannot be directly displayed inside the Clarinox Debugger environment. They need to be analyzed first. Analysis is performed by plug-ins attached to the debugger software.

One or more DLL files (called plug-ins) may be attached to debugger framework to provide further services. A plug-in can do any or all of the following:

- analyze a user-defined message type;
- provide services to other installed plug-in modules;
- provide one or more items in the context popup menu (that is opened using a right click function on the debuggers main screen).

A plug-in DLL comprises only one exportable function. This function creates an instance of the main C++ class, and passes a pointer to the instance, to the debugger. Therefore, the main part of a plug-in is a C++ class. This class must inherit from the basic CDPluginInterface class, an abstract class with one compulsory method and several optional methods. The exportable function will identify the object methods to the debugger by passing the address of an instance of the plug-in class to the debugger. The user needs to write a C++ class (inherited from the debugger base class) and overload the appropriate methods. The overloaded function must pass some general information about the plug-in (i.e., the plug-in name, general description, etc.) and register the relevant message types with the framework.

It is then necessary to write a parser function to receive and analyze the messages of those registered types. This method is called every time a message of one of registered types is received from the target application, each time the user triggers the message representation on the main screen and when there is a search procedure in progress. The function would then typically create a human readable representation of the message to display, store or return to the debugger interface.

Delimiter	Length of Tag ID	Tag ID	Reader ID	RSSI
1 Byte	1 Byte	Variable	2 Byte	2 Bytes

Figure 5. **A simple RS232 packet structure**

4.2 An Example Plug-in

In this section, we will write a simple plug-in, which is a protocol monitor for a typical RSSI tag reader. These readers read proprietary active tags, and measure the power level of the signal coming from the tags. Then, they produce a one-byte value called RSSI (which is approximately inversely proportional to the distance of the tag from the reader). RSSIs will be encoded into a packet and will be sent to a computer or an embedded system via RS232 (Figure 5). This example depicts how a protocol monitor for a proprietary RFID protocol can be quickly designed and deployed.

These packets are captured by the target software and also moved to the debugger machine for processing. Using a simple pre-defined macro, a header containing information such as an arbitrary message ID is inserted to the packets before being sent to the debugger. On the debugger machine, a small plug-in is used to monitor and analyze the packets of the chosen message ID and show the results on the debugger main screen.

The item to be handled by the developer is to write a class which derives from *CDPluginInterface*, and to overload some of its methods. The method *exportPluginInfo*. provides general information about the plug-in such as its name, a general description, etc. A second method, *exportMessageID* registers the message ID(s) in which the plug-in is interested. In this example, only one arbitrarily chosen message ID is used (assuming only one type of packets exists). The main method to overload is a message handler called *messageParser*. All messages of the registered message ID are delivered to this function by the debugger. The method is written to parse the reader packets (encoded in the format mentioned in Figure 5) and then to return a human-readable text about the packet, to be shown at the debugger interface.

All other details (e.g. handling physical communication, dispatching the messages to the appropriate plug-in, handling user interface details, loading and initializing the DLL plug-ins etc.) are handled by the *Softframe* debugging framework. Since the packets are redirected by the target system (rather than being captured directly from the reader), the plug-in demonstrates whether or not the target system is able to receive and properly separate the incoming data stream into independent packets. Furthermore, the result can be used effectively to find the state of the target system when other errors happen and are captured by the debugger (e.g. what packet was being parsed by the remote system at the time the error happened).

5. Conclusions

We have described a plug-in based debugger that is built as a C++ interface class and is independent of the physical layer, which can be a network, a serial connection (e.g., RS232), or even a file on hard disk or flash memory. The overall objective of the debug system is to reduce risks associated with commonly required code and therefore to reduce overall debug times. It also offers pre-built common architectural blocks (finite state machines, inter-process communication, timers, integrated debugging), along with wireless and wired protocols and I/O interfaces.

6. References

[1] Nass, R., Annual study uncovers the embedded market, Sept 2, 2007, [online]: http://www.eetimes.com/design/other/4007166/Annual-study-uncovers-the-embedded-market.

[2] Cravotta, R., Shedding light on embedded debugging, 2008, [online]: http://www.edn.com/article/472158-Shedding_light_on_embedded_debugging.php.

[3] Clarinox Technologies Pty. Ltd., The Clarinox Softframe, 2010, [online]: www.clarinox.com.

[4] Stan Schneider and Lori Fraleigh, The Ten Secrets of Embedded Debugging, September 2004, [online]: http://www.eetimes.com/design/other/4025015/The-ten-secrets-of-embedded-debugging.

[5] Kraemer, S., et al., "HySim: a Fast Simulation Framework for Embedded Software Development," in *Proceedings of the 5th IEEE/ACM international conference on Hardware/software codesign and system synthesis* Salzburg, Austria: ACM, 2007, pp. 75-80.

[6] Ho, A., Hand, S., and Harris, T., "PDB: Pervasive Debugging with Xen," in *Proceedings of the Fifth IEEE/ACM International Workshop on Grid Computing*, 2004, pp. 260-265.

[7] Girod, L., et al., "Emstar: A Software Environment for Developing and Deploying Heterogeneous Sensor-Actuator Networks," *ACM Trans. Sen. Netw.,* vol. 3, p. 13, 2007.

[8] Yang, J., et al., "Clairvoyant: a Comprehensive Source-Level Debugger for Wireless Sensor Networks," in *Proceedings of the 5th international conference on Embedded networked sensor systems* Sydney, Australia: ACM, 2007, pp. 189-203.

[9] Nagpurkar, P., et al., "Efficient Remote Profiling for Resource-Constrained Devices," *ACM Trans. Archit. Code Optim.,* vol. 3, pp. 35-66, 2006.

[10] Corliss, M. L., Lewis, E. C., and Roth, A., "DISE: a Programmable Macro Engine for Customizing Applications," *International Symposium on Computer Architecture (ISCA),* vol. 31, pp. 362-373, 2003.

[11] Holgado-Terriza, J. A. and Viudez-Aivar, J., "A Flexible Java Framework for Embedded Systems," in *Proceedings of the 7th International Workshop on Java Technologies for Real-Time and Embedded Systems* Madrid, Spain: ACM, 2009, pp. 21-30.

[12] Titzer, B. L. and Palsberg, J., "Nonintrusive Precision Instrumentation of Microcontroller Software," in *Proceedings of the 2005 ACM SIGPLAN/SIGBED conference on Languages, compilers, and tools for embedded systems* Chicago, Illinois, USA: ACM, 2005, pp. 59-68.