A Development Environment for Embedded Software

Byeongdo Kang

Department of Computer and Information Technology,
Daegu University, Republic of Korea
{bdkang, yjkwon}@daegu.ac.kr

Abstract

In recent years, embedded systems have become so complex and the development time to market is required to be shorter than before. As embedded systems include more functions for new services, embedded software gradually grow in size, and development costs and time are increasing. In order to overcome this serious matter, we need a customized design and test technique for embedded software. In this paper, we present a software architecture style for embedded software. It facilitates the composition of reusable functions and helps developers to reduce development time. Because the costs associated with revealing errors of embedded software in applications are rising, we propose a test method and tools for target environments.

Keywords: *Embedded Software, Software Design, Software Test.*

1. Introduction

An embedded system is a special purpose computer that is used inside of a device. For example, a microwave oven contains an embedded system that accepts input from the panel, controls the LCD display, and turns on and off the heating elements that cook the food. Embedded systems generally use microprocessors that contain many functions of a computer on a single device. Motorola and Intel make some of the most popular microprocessors. Software engineers will often program directly to the microprocessor hardware without using a host operating system. As embedded systems have become more sophisticated and have started to include more general computer functions such as networking, using a host operating system has become more popular. Linux and Windows Embedded are two popular operating systems for implementing embedded systems.

In recent years, the functions added to embedded systems have grown various and complex in most parts of the applications. Therefore, the development costs and time are increasing. This problem makes developers want flexible implementation to consider changing requirements. Because hardware development for the evolving functions are less flexible than software-based implementation, embedded software becomes to include more functions and is getting bigger. It is getting difficult for software developers to keep the development time to market on time. Embedded software design costs are rising considerably because of the increased complexity of functions. This leads us to a reason to need an adequate design and test technique for embedded software.

In this paper, we propose a design method and an architecture style for embedded software. In addition, a test technique is also presented. In section 2, we introduce related works in brief. And then, we present a design and test technique for embedded software in section 3. Finally we will come to a conclusion with summary.

2. Related works

Embedded systems for reactive real-time applications are implemented as a mixture of software and hardware. Software is used for features and flexibility while hardware is used for performance. Design of embedded systems is subject to many different types of constraints, such as time, size, power consumption, reliability, and costs.

Traditional methods for designing embedded systems start from specifications. The specifications are written for hardware and software respectively. Hardware-software partition is decided according to developers' experiences. Software components are integrated into a complete system with hardware later. The problems with traditional methods are the lack of a unified hardware-software representation and the immaturity of well-defined co-design of hardware and software.

2.1 POLIS

The POLIS[1] is a methodology for specification with a unified hardware-software representation, automatic synthesis, and validation of embedded systems. In POLIS, designers write their specifications in a high

level language that can be directly translated into co-design finite state machine(CFSM). The CFSM, like a classical finite state machine(FSM), transforms a set of inputs into a set of outputs with only a finite amount of internal state. The difference between the two models is that the synchronous communication model of classical concurrent FSMs is replaced in the CFSM model by a finite, non-zero, unbounded reaction time. This model of computation can also be described as global asynchronous and local synchronous. Each element of a network of CFSMs describes a component of the system to be modelled. The CFSM specification is a unbiased towards hardware or software implementation.

The formal specification and synthesis methodology embedded within POLIS makes it possible to interface directly with existing formal verification algorithms that are based on FSMs. POLIS includes a translator from the CFSM to the FSM formalism which can be fed directly to verification systems.

System level hardware-software co-simulation is a way to give designers feedback on their design choices. These design choices include hardware-software partitioning, CPU selection, and scheduler selection. These decisions are based on design experience.

A CFSM sub-network chosen for hardware implementation is implemented and optimised using logic synthesis techniques. A CFSM sub-network chosen for software implementation is mapped into a software structure that includes a procedure for each CFSM. Interfaces between different implementation are automatically synthesized within POLIS. These interfaces come in the form of cooperating circuits and software procedures embedded in the synthesized implementation.

2.2 UML-based design methodology

There is a system level design method for embedded real-time systems combining the informal strengths of UML[2] with the formal strengths of SDL[3]. The key aspects of this method are the use of object-oriented concepts to specify hardware and software components. The informal strengths of UML are married with the formal strengths of SDL in this specification capturing flow[4].

The complete specification flow consists of four phases. Each phase of the system level design flow is characterized by a set of products. During the first formulation phase of system design and the specification capture, UML and its process are used. At the subsequent formalization phase, SDL concepts defined in UML 2.0 is used with additional patterns. The static and dynamic views of UML and SDL are used for the specification of the system communication and coordination. Then, the detailed design phase addresses the internal specification of the modules and other elements. Full functional specifications are to be described with detailed SDL. The fourth phase is the deployment specifying the target architecture where the functionality of a system is mapped.

3. Design and test for embedded software

Embedded software is a little bit different from the applications software or systems software that is running on a computer such as personal computers, workstations, or mainframe computers in the way of developing them. Applications software or systems software is usually developed on a host computer that has high computing power and practical software development environments with graphic user interfaces. The target processor of an embedded system is typically minimal in function and size because its goal is to minimize cost and space. So, the target hardware of embedded systems will not support any software development tools complying with a well-defined methodology. Therefore, the program running on the target is developed on a host computer at the start, and then cross-compilers and linkers are used to generate target code, and finally it is downloaded in the target processor. Each of the host environment and the target environment has different functionalities and interfaces to users. Design and test tools that run on the host system are various and provide users with high level interfaces and information about the execution state of program. But, few tools exist on the target system.

3.1 Structure of embedded systems

An embedded system is a specialized computer system that is part of a larger system or machine. Typically, an embedded system is housed on a single microprocessor board with the programs stored in ROM. Some examples of applications of embedded systems are consumer electronics, telecommunications, automobiles, and plant control. Although the application domains are different from each other, they have common structure in functional configuration. Figure 1 shows a layered structure including hardware platform, hardware-dependent software, application software, and application programming interfaces[5][6]. Application program interfaces are necessary for communication between the hardware-dependent software and the application layer of software. The hardware-dependent software is connected with the physical hardware and network. Real-time operating system and device drivers are closely coupled with hardware platform. From the viewpoint of an application domain, performance and

size are the constraints that usually determine the hardware platform. For a particular application, a processor must meet a minimum speed, and the memory system must meet a minimum size.

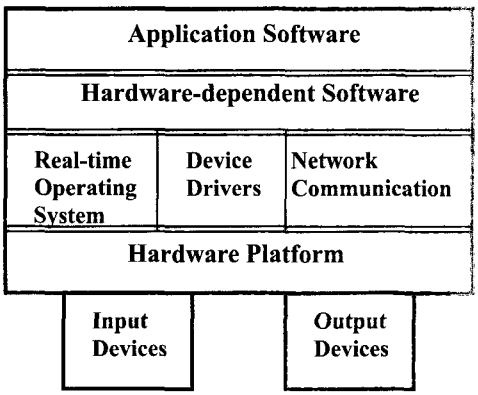

Figure 1. Functional structure of embedded systems

3.2 Designing embedded software

Designing an embedded system includes main four steps. The basic steps are the following:

- Requirements specification
- Hardware and software partitioning
- Software design
- Hardware design
- Interface design
- System integration and test

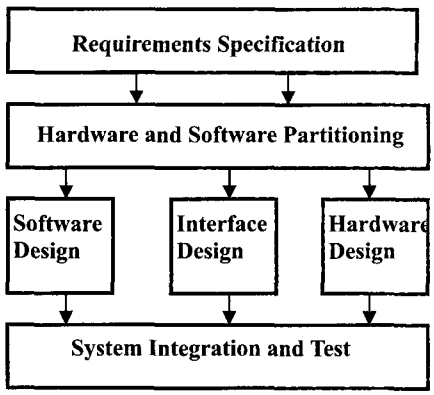

Figure 2. System-level design processes

At the first step, user's requirements are analysed and described with a specification language. Developers can derive required functions after analysing requirements. These functions are allocated to hardware or software. Hardware and software are co-developed. At the same time, interfaces between hardware and software are developed. After all parts of hardware and software are developed, they are integrated into a system. Figure 2 represents the system level design steps.

On the other hand, the basic phases for developing embedded software are as follows:

- Software architecture design
- Software subsystem definition
- Functional block design
- Feature and task implementation

(1) Software architecture design

The first phase of designing embedded software is the software architecture design. Software architecture is the overall system structure as described by the components and connections among components[7]. Software architectural styles categorize architectures based on characteristics specific to a structural composition, such as shared data, abstract data type, implicit invocation, and pipe and filter.

The software team understands the target system at this phase and reviews it with the proposed hardware architecture. To compose target system architecture, the user selects the optional features desired for the target system. Once all the features have been selected for a target system, the target system architecture is composed from the corresponding architecture patterns[8]. The architecture design will be further refined in software subsystem definition.

(2) Software subsystem definition

A system can be divided into one or more subsystems. Each subsystem is a logical or a physical unit to be developed. A subsystem contains one type of features that the system needs to support. The features are implemented by tasks that are executable units. Those tasks are defined in the subsystems.

(3) Functional block design

Each subsystem is divided into one or more functional blocks based on characteristics and manageable size. A block contains various tasks that accomplish the functions of features on a processor. Block design specifies the message interactions between tasks. All the inter-processor communication takes place through messages. Tasks and message interfaces are defined in detail in this phase. All the data and their possible values are identified.

(4) Feature and task implementation

Designing a task means that all the interfaces

between tasks should be well defined. All the message parameters and timer values should be defined in this phase. Tasks are implemented by a programming language, and then the source programs are stored in computer files.

Figure 3 shows the hierarchical structure of software. One system consists of one or more subsystems. Subsystems are implemented and loaded on processors. A processor is a executable unit for one or more subsystems. All the blocks are also implemented and then become executable modules running on a processor.

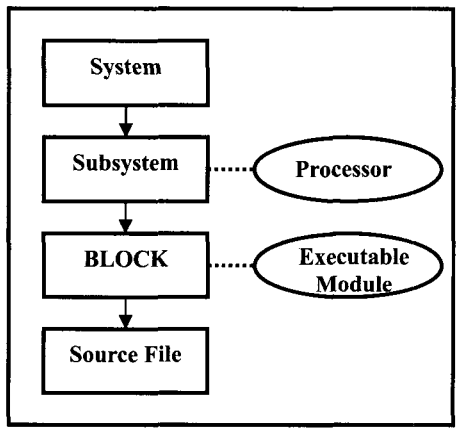

Figure 3. The software hierarchy

3.3 Software testing

Embedded systems have difficulty in testing and debugging software. Because embedded systems are usually developed on custom hardware configurations, tools and techniques that can apply to one are not applicable to another[9]. We present an integration test strategy regarding software architecture styles and tools for target debugging in this section.

(1) Software architectures and test strategies

Integration test insure the consistency of component interfaces. It supports the systematic composition of blocks or subsystems. There are six traditional test strategies for integration test, such as critical module, sandwich, top-down, bottom-up, thread, and big-bang.

Figure 4 shows the positive relationships of integration test strategies and four software architectures. For example, the thread integration test strategy would be the best choice if a pipe and filter architectural style were chosen[10].

	Type			
	Shared Data	Abstract Data	Implicit Invocation	Pipe& Filter
Critical Module		O		
Sandwich		O	O	
Top-down			O	
Bottom-up		O		
Thread			O	O
Big Bang	O			

Figure 4. Integration test and architectural styles

(2) Target debugger

A debugging system has at least two processes. One is for running test program and the other is used to have the debugger execute. Figure 5 shows an environment of a target debugger. Developers perform their test activities on a host computer through interface. By the target debugger, developers can use commands for debugging, such as memory handling function, register management, breakpoint, and execution control.

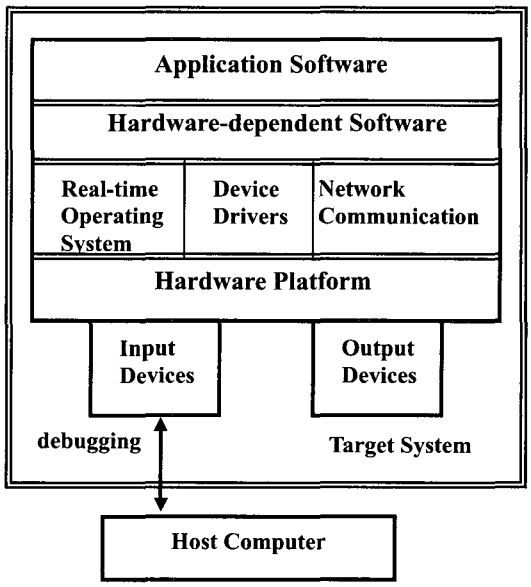

Figure 5. The environment of target debugger

(3) Testing and debugging processes for embedded software

We suggest three steps for testing the application software in the embedded system. The first step is the functional test. The functional test consists of the unit test and the integration test. The second step is the host static testing. The last step is the target test that includes the test for interfaces between software and hardware.

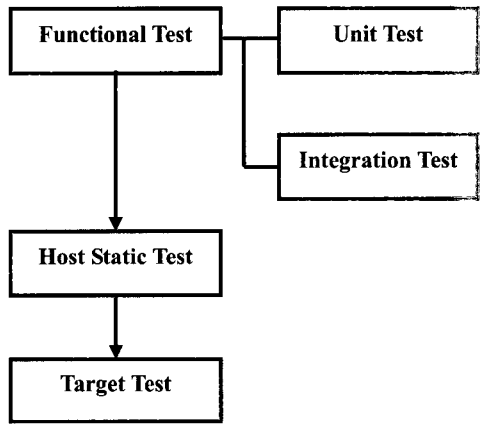

Figure 6. Three test steps for embedded software

- Functional test: this test is to verify whether functions implemented perform in compliance with the user's requirements correctly or not. At first, all developers perform the unit test that is to make certain that their blocks implemented work completely. And then, all the blocks are integrated and tested to confirm whether they work together according to the functional requirements.
- Host static test: the software subsystem that is composed of blocks is tested on the host machine. The values of variables, control flow, and the sequence of message communication are identified by using a host debugger.
- Target test: after accomplishing the host static test, the software is loaded into the target system. The software subsystem is integrated with the hardware subsystem. In this test step, developers use a target debugger. The target debugger provides developers or testers with the functions of handling memory, managing register, manipulating breakpoint, and execution control. Test engineers can debug target system during running the target program through the target debugger.

3.4 Target testing for embedded software

Software engineers develop their programs on a host machine. The host machine has fast processors, plenty of hardware resources, and powerful software development tools. In contrast, the target system doesn't have them. So, program developers load the object modules in the target board after developing and cross-compiling their source files. Software developers can't usually debug or test their programs on a target system because it doesn't have enough software or hardware resources such as software testing tools and memory. So it is hard for them to trace the states of executing programs on the target board. Figure 7 shows a host machine and a target board for an embedded system.

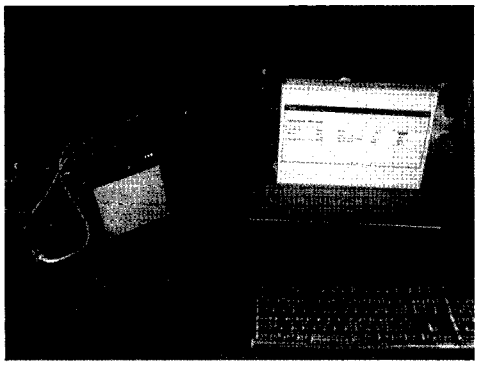

Figure 7. A host machine and a target board

As one of possible solutions of target testing, we propose a method for tracing the execution states during running the target system. Our method can analyse the execution flows of programs without being a heavy burden on the target system. This method extracts the information about conditions, input, actions, output, and related devices while target system is running. Conditions represent the branches of execution flows. The combination of input-actions-output shows the preconditions and post-conditions of activities at a given condition. Devices are related with performing the actions. Figure 8 shows the screen layout for execution flows.

In order to trace the execution flows, programs for a target system should be compiled with our testing driver. The testing driver includes software routines that extract information about execution flows. The object module after compilation is loaded on the target system board.

When the programs are running on the target system, test engineers can show the execution flows through the screen. Figure 9 represents the relationship between programs for target system and testing driver.

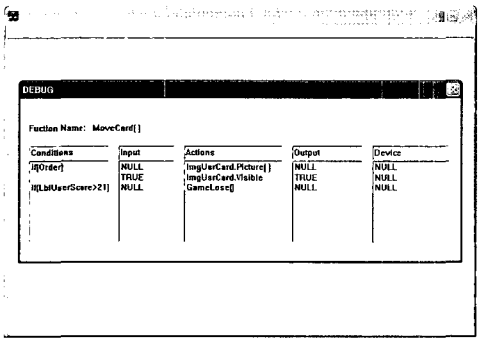

Figure 8. Screen layout for extracting execution flows

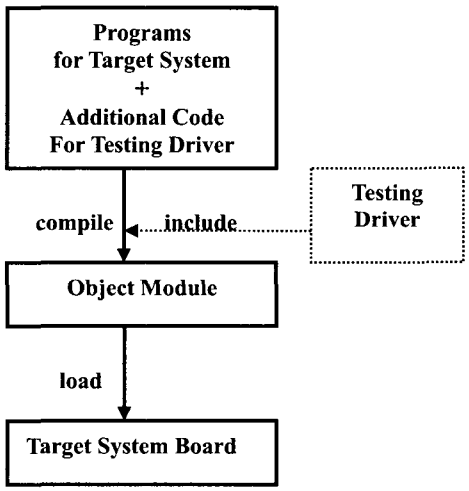

Figure 9. Embedded programs and testing driver

4. Conclusion

The functions added to embedded systems have grown various and complex in most parts of the applications. Therefore, the development costs and time are increasing. This leads us to a reason to need an adequate design and test technique for embedded software. The basic phases for developing embedded software are four steps: software architecture design, software subsystem definition, functional block design, and the feature and task implementation. Embedded systems have difficulty in testing and debugging software. Because embedded systems are usually developed on custom hardware configurations, tools and techniques that can apply to one are not applicable to another. We present an integration test strategy regarding software architecture styles and tools for target testing.

References

[1] The POLIS System, Release 4.0, http://www-cad.eecs.berkeley.edu/Respep/Research/hsc/abstract.html

[2] B. P. Douglass, Real-Time UML: Developing Efficient Objects for Embedded Systems, 2nd Edition, Addison Wesley, 2000.

[3] R. Saracco, J. Smith, and R. Reed, Telecommunications Systems Engineering Using SDL, NewYork: North-Holland, 1989.

[4] Gjalt de Jong, "A UML-Based Design Methodology for Real-Time and Embedded Systems," Proceedings of the 2002 Design, Automation, and Test in Europe Conference and Exhibition, 2002.

[5] Alberto Sangiovanni-Vincentelli and Grant Martin, "Platform-Based Design and Software Design Methodology for Embedded Systems," IEEE Design & Test of Computers, November-December, 2001, pp.23-33.

[6] M. Sgroi, L. Lavagno, and A. Sangiovanni-Vincentelli, "Formal Models for Embedded Systems Design," IEEE Design & Test of Computers, April-June, 2000, pp.2-15.

[7] Mary Shaw and David Garlan, Software Architecture: Perspectives on an Emerging Discipline, Prentice Hall, 1996.

[8] H. Gomaa and G. A. Farrukh, "Composition of Software Architectures from Reusable Architecture Pattern," Proceedings of the ISAW3, Orlando, USA, 1998, pp.45-48.

[9] Harry Koehnemann and Timothy Lindquist, "Towards Target-Level Testing and Debugging tools for Embedded Software," Proceedings of the Conference on TRI-Ada'93, Seattle, USA, October, 1993, pp.288-298.

[10] Nancy S. Eickelmann and Debra J. Richardson, "What Makes One Software Architecture More Testable Than Another?" ACM SIGSOFT 96 Workshop, San Francisco, USA, 1996, pp.65-67.