임베디드 소프트웨어 개발을 위한 JTAG 기반의 디버깅 도구

김병철,° 강문혜, 전용기, 임채덕*
경상대학교, *한국전자통신연구원
{kim,°turtle,jun}@race.gsnu.ac.kr, cdlim@etri.re.kr

A JTAG-Based Debugging Tool for Developing Embedded Softwares

Byung-Chul Kim,° Moon-Hye Kang, Yong-Kee Jun, and Chea-Deok Lim*
Gyeongsang National University, *Electronics and Telecommunication Research Institute

요 약

임베디드 소프트웨어는 타겟 시스템의 자원과 타이밍에 민감하므로 실제 타겟 시스템과 동일한 환경에서 디버깅해야한다. 이를 위한 기존의 기법들은 타겟 시스템의 자원에 직접적으로 접근하여 시스템 상태를 조사하거나 제어한다. 그러나 이러한 기법들은 내부 신호나 자원에 대한 접근이 제한되어 있는 SoC (System-On-a-Chip) 프로그램을 디버깅하기는 부적합하다. 본 논문에서는 산업 표준화된 JTAG을 기반으로 공개 소프트웨어인 gdb를 연동하여 SoC 소프트웨어를 디버깅하는 도구를 제안한다. 따라서 본 도구는 타겟 시스템에 영향을 주지 않고 경제적으로 디버깅할 수 있는 환경을 제공한다.

1. 서 론

임베디드 소프트웨어[KiKL03, Lee01, Yagh03, RuCo01]는 처음에는 간단한 제어 프로그램만으로 산업용 기기를 제어하는 데 그쳤지만 최근에는 군사용 제어기기, 디지털 정보, 가전기기, 자동 센서 장비 등의 폭넓은 분야에 사용되고 있다. 이러한 임베디드 시스템의 기능을 결정하는 임베디드 소프트웨어를 안정되게 개발하기 위해서는 이들을 효과적으로 디버깅하기 위한 도구가 필요하다. 그러나 임베디드 소프트웨어는 타겟 시스템의 자원과 타이밍에 민감하여 실제 타겟 시스템과 동일한 환경에서 디버깅해야 하는 어려움이 있다. 특히 메모리나 입출력 장치등과 같은 모듈들을 하나의 프로세서에 집적한 SoC (System-On-a-Chip) [JoPu02, Pete99]는 내부 신호나 자원에 대한 접근이 제한되어 디버깅을 더욱 어렵게 한다.

임베디드 소프트웨어를 디버깅하는 전통적인 기법으로 실시간 개발에 편리하고 실시간 입출력의 오류 정정을 위한 하드웨어 및 소프트웨어 기능을 가지는 ICE(In Circuit Emulators)와 디지털 시스템의 논리 신호를 포착하여 디버깅하는 logic analyzer가 있다. 그러나 이러한 기법들은 타겟 시스템의 자원에 직접적으로 접근하여 시스템 상태를 조사하거나 제어하므로 내부 신호나 자원에 대한 접근이 제한되어 있는 Soc 프로그램을 디버깅하기는 부적합하다.

본 논문에서는 산업 표준화된 JTAG(Joint Test Action Group)을 기반으로 공개 소프트웨어인 gdb[GiSh03, Gat101, StPS03]를 연동하여 SoC 소프트웨어를 디버깅하는 도구를 제안한다. 제안된 도구는 타겟 시스템에 영향을 주지 않고 공개 소프트웨어인 gdb를 사용하므로 경제적으로 디버깅할 수 있는 환경을 제공한다. 또한, gdb와 연결되는 Debug Agent를 두어 디버깅 도구의 유연성을 두었으며 gdb가 지원하는 GUI를 그대로 수용하여 효과적인 디버깅을 할 수 있게 한다. 본 도구는 XScale 코어를 사용하는 Intel PXA250 프로세서에서 수행되는 프로그램을 디버깅하기 위해 설계 및 구현되었다. 또한, GDB와 TCP/IP로 연 결되는 디버그 에이전트를 타겟 시스템에 의존적인 부분과 독립적인 부분으로 구분하여 새로운 타겟 시 스템을 위한 디버거의 개발로 확장하기가 용이하도록 하였다.

2절에서는 본 연구의 배경으로 임베디드 소프트웨어 디버깅 기법과 JTAG에 대해서 살펴보고, 3절에서는 제안된 JTAG 기반의 디버깅 도구를 설명한다. 마지막으로 결론 및 향후 과제를 제시한다.

2. 임베디드 소프트웨어 디버깅

본 절에서는 대상으로 하는 디버깅 기법에 대해서 살펴보고, 기반으로 하는 JTAG에 대해서 본다.

2.1 디버깅 기법

임베디드 소프트웨어는 타겟 시스템의 자원과 타이밍에 민감하여 실제 타겟 시스템과 동일한 환경에서 디버깅해야 한다. 이를 위한 기존의 기법들은 ICE(In Circuit Emulators)와 logic analyzer가 있다. ICE는 하드웨어 모방이 가능하여 실시간 개발에 편리한 수단으로서 실시간 입출력의 오류 정정을 위한 하드웨어 및 소프트웨어 기능을 가진다. logic analyzer는 디지털 시스템의 논리 신호를 포착, 기록하고 나타내는 다수의 채널을 가지는 도구로 데이터 분석기와 시간 분석기의 두 가지 유형이 있다. 데이터 분석기는 데이터 블록을 음극선관(CRT)을 통해 2진수 형태로 나타내고, 시간 분석기는 포착된 데이터를 사각 파형 모양의 일련의 펄스 형태로 출력한다. 이러한 기법들은 타겟 시스템의 자원에 직접적으로 접근하여 시스템 상태를 조사하거나 제어하므로 내부 신호나 자원에 대한 접근이 제한되어 있는 Soc [JoPu02, Pete99]프로그램을 디버깅하기는 부적합하다. 그러므로 SoC 프로세서 자체에서 제공되는 디버그 모듈을 이용하여 SoC 소프트웨어를 디버깅하는 도구를 제안하기 위해서 JTAG을 기반으로 한다.

2.2 JTAG (Joint Test Action Group)

70년대 중반에는 "bed-of-nails"라는 테크닉을 사용하여 직접 PCBs(print circuit boards)에 접촉하는 방식으로 board를 테스트하였다. 그러나 보드 단자 사이의 거리가 좁아짐에 따라 테스트가 어려워지고 다중층 보드(multi layer board)가 나타나면서 테스트하는 것이 불가능하게 되었다. 이러한 문제점들을 해결하기 위해서 80년대 중반에 Joint European Test Access Group이 결성되고 그 이후에 North American company와 함께 JTAG으로 변경 후, IEEE에서 1990년에 표준화하여 IEEE 표준 1149.1-1990, IEEE 표준 1149.1a-1993의 Standard Test Access Port and Boundary-Scan Architecture를 따른다.

Boundary-Scan으로 더 많이 알려져 있는 JTAG은 칩 내부에 Boundary Cell을 두어 외부의 핀과 일대 일로 연결하여 프로세서가 할 수 있는 동작을 Cell을 통하여 모든 동작을 인위적으로 수행하므로써 하드웨어 테스트나 연결 상태 등을 체크할 수 있는 기능을 가진다. 전체적인 인터페이스는 TAP(Test Access Port)라고 하는 5개의 핀(TDI, TMS, TCK, nTRST, TDO)에 의해서 제어되며 이를 이용하므로써 프로세서의 상태와는 상관없이 디바이스의 모든 외부 핀을 구동시키거나 값을 읽어 들일 수 있다. 응용 프로세서상의 JTAG 인터페이스는 Intel XScale 시스템[Inte00,

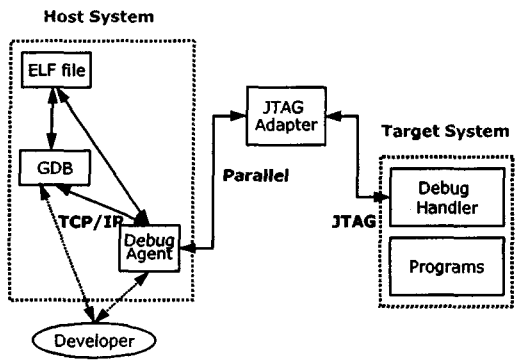

그림 3.1 EDebugger 구조

Inte01, Inte02ip, Inte03]의 소프트웨어 디버깅을 위한 하드웨어 인터페이스로써 사용된다.

3. JTAG 기반 디버깅 도구

본 절에서는 JTAG 인터페이스를 사용하여 Intel PXA250 프로세서에서 수행되는 프로그램을 타겟 시스템에 영향을 주지 않고 경제적으로 디버깅하기 위한 도구인 EDebugger의 전체적 구조와 하위 모듈에 대해서 살펴보고 EDebugger의 실행과 이를 이용한 디버깅을 살펴본다.

3.1 EDebugger 구조

본 논문에서 제안하는 도구는 타겟 시스템에 영향을 주지 않고 경제적으로 디버깅할 수 있는 환경을 제공하기 위해서 호스트 시스템의 디버거로 공개소프트웨어인 GNU Debugger인 GDB를 사용하고, 호스트 시스템과 타겟 시스템간의 통신을 위한 고가의 ICE 장비를 패러럴 포트[Harr97, RuCo01]와 JTAG 포트의 인터페이스로 대체하였다. 그리고 디버깅 도구의 유연성을 위해서 원격 시리얼 프로토콜을 사용하여 GDB와 TCP/IP로 연결되는 Debug Agent를 구현하였다. 개발자가 Debug Agent를 서버/클라이언트 모델로 수행시킬 경우에는 GDB가 클라이언트가 되고 Debug Agent가 서버로 운영되며, 단독으로 수행할 경우에는 타겟 시스템에 대한 하드웨어 수준의 디버깅을 지원하도록 하였다.

그림 3.1은 본 논문에서 제안하는 JTAG 기반 디버깅 도구인 EDebugger의 전체 구조도이다. EDebugger는 크게 GDB와 TCP/IP로 연결되거나 단독으로 실행되는 호스트 시스템의 Debug Agent, 호스트 시스템과 타겟 시스템의 통신을 중재하는 JTAG Adapter, 타겟 시스템의 명령 캐쉬에 상주하여 프로그램의 디버깅을 처리하는 Debug Handler로 구성된다. 그림 3.2는 Debug Agent의 구조이다. 이 모듈은 개발자의 디버깅 명령과 GDB에서 전송된 디버깅 메시지를 분석하는 모듈로 명령과 메시지를 분석하는 GDB 서버, 분석된 내용을 타겟 시스템의 디버깅 명령으로 전환하는 PseudoCPU, 그리고 전환된 명령에 따라 타겟 시스템의 JTAG 인터페이스를 제어하는 신호를 송수신하는 JTAG Controller로 구성된다. 이러한 Debug Agent는 타겟 시스템에 의존적인 부분과 독립적인 부분으로 구분하여 새로운 타겟 시스템을 위한 디버거의 개발로 확장하기에 용이하며, 이를 이용함으로써 GDB 코드를 수정하지 않고 GDB가 지원하는 GUI를 그대로 사용할 수 있어서 효과적인 디버깅이 가능하다. JTAG Adapter는 호스트 시스템의 패러럴 포트와 타겟 시스템의 JTAG 포트를 중재하는 하드웨어 장치로 각각에 버퍼를 두어 두 통신 포트의 전송속도 차이로 인한 데이터의 손실을 방지하고, 패러럴 포트에서 출력되는 전압과 타겟 시스템에서 사용되는 전압 차이에 의한 시스템의 오동작을 방지한다. Debug Handler는 타겟 시스템의 미니 명령 캐쉬에 상주하여 호스트 시스템의 Debug Agent와 디버깅 메시지를 통신하는 모듈로 Intel XScale 코어 디버거 구조의 hot-debug 솔루션을 이용한다.

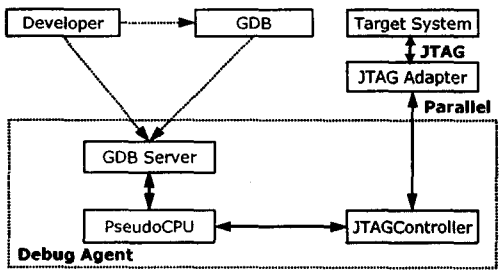

그림 3.2 Debug Agent

3.2 EDebugger 실행

표 3.1은 EDebugger의 구동을 위해서 사용되는 옵션이다. 호스트 시스템의 패러럴 포트는 0x378번 포터로 고정되어 있으므로 호스트 시스템 부팅 시에 포트번호를 설정해야한다. EDebugger는 먼저 타겟 프로세서의 식별 번호를 읽어 와서 Intel PXA250 프로세서라는 것을 확인하면 타겟 시스템의 미니 명령 캐쉬에 디버그 핸들러를 다운로드하게 된다. PXA250 프로세서가 아니면 오류 메세지를 보내고 종료한다. 그림 3.3-(e)는 EDebugger의 정상적인 실행화면이다.

표 3.2는 EDebugger의 시스템 명령어, 메모리/레지스터 명령어, 디버깅 제어 명령어를 보인다. 시스템 명령어는 타겟 시스템의 적절한 설치에 대한 검사, 명령 캐쉬에 프로그램을 다운로드, 타겟 시스템의 초기화와 재부팅을 담당한다. 메모리/레지스터 명령어는 타겟 시스템의 메모리/레지스터의 내용을 읽고 쓰는 기능을 담당한다. 마지막으로 디버깅 제어 명령어는 타겟 시스템에서 수행되는 프로그램에 대해서 해당 주소에 중지점을 설정하거나 해제하는 기능과 중지점으로 부터 프로그램을 재실행하는 기능을 담당한다.

3.3 디버깅

옵션	의 미
-d	GDB의 원격 시리얼 메시지 출력
-p	호스트 시스템의 패러럴 포트를 통한 접속
-i port_num	GDB와 EDebugger의 TCP/IP 연결을 위한 포트 번호 지정 (생략시 3141번이 디폴트)
-h	EDebugger의 수행 옵션을 보여주기

표 3.1 EDebugger 실행옵션

분류	명령어	설명
시스템 명령어	check	프로세서의 ID를 검사.
	load	외부메모리에 프로그램을 적재.
	loadic	명령 캐쉬에 프로그램을 적재.
	reboot	프로세서 재부팅.
	reset	JTAG 컨트롤러 초기화.
	quit	EDebugger 종료.
메모리/레지스터 명령어	get	메모리로부터 데이터를 가져옴.
	put	데이터를 특정 메모리 위치에 기록.
	register	레지스터의 내용을 변경하고 확인.
디버깅 제어 명령어	continue	중지된 프로그램 이후를 수행.
	hwbreak	프로그램의 특정 번지에 대한 중지점 설정/해제.

표 3.2 EDebugger 명령어

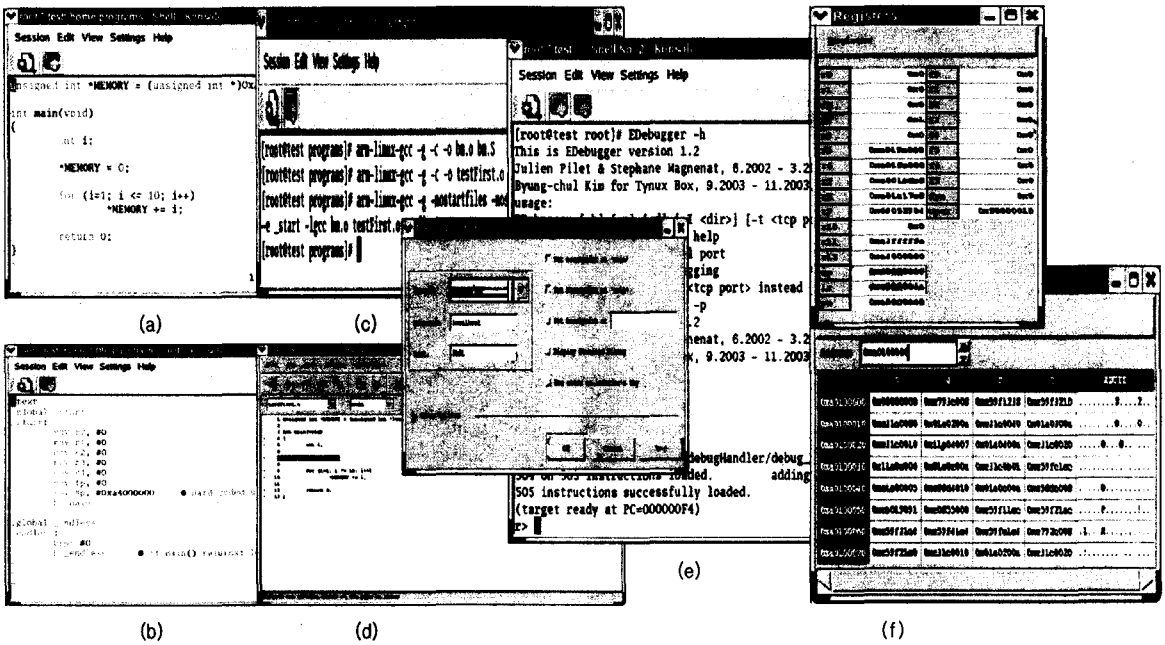

그림 3.3 경제적 디버깅 도구를 이용한 디버깅

C언어로 작성된 간단한 프로그램을 제안된 EDebugger와 GDB를 사용하여 디버깅하는 과정을 그림 3.3에서 보인다. (a)는 C로 작성된 프로그램이고 (b)는 초기화 루틴이다. 초기화 루틴은 각 레지스터를 초기화하고 프로그램이 종료되었을 경우 프로그램의 마지막에 중지점을 설정하도록 하였다. 다음으로 (c)에서는 (a)프로그램과 (b)초기화 루틴을 각각 컴파일한 후 원하는 번지부터 실행 가능하도록 링크시킨다. 그리고 EDebugger를 패러럴 포트로 타겟 시스템과 연결되도록 옵션을 지정해서 실행시키고, GDB를 실행하여 EDebugger와 연결을 설정한다. (d)는 GDB의 환경설정 부분과 실행하는 것을 보인다. 마지막으로 EDebugger에 접속한 후 프로그램을 다운로드하고 필요한 부분에 중지점을 설정하여 프로그램을 중지점까지 실행하여 원하는 값들을 확인하고, 현재 디버깅하고 있는 프로그램의 작업을 중지하여 EDebugger와의 접속을 끊는다. (e)는 EDebugger가 정상적으로 수행된 것을 보이며 (f)는 현재 타겟 시스템의 레지스터에 저장된 값과 특정 메모리에 존재하는 값들을 보여준다.

4. 결 론

본 논문에서는 산업 표준화된 JTAG(Joint Test Action Group)을 기반으로 공개 소프트웨어인 gdb를 연동하여 SoC 소프트웨어를 디버깅하는 도구를 제안하였다. 제안된 도구는 타겟 시스템에 영향을 주지 않고, 공개 소프트웨어인 gdb를 사용하므로 경제적으로 디버깅할 수 있는 환경을 제공한다. 또한, gdb와 연결되는 Debug Agent를 두어 디버깅 도구의 유연성을 두었으며 gdb가 지원하는 GUI를 그대로 수용하여 효과적인 디버깅을 할 수 있게 한다.

참고 문헌

[Elso00] Elson, J., *PARAPIN: A Parallel Port Pin Programming Library for Linux*, 2000.

[GiSh03] Gilmore, J., and S. Shebs, GDB Internals, 2003.

[Gat101] Galiff, W., *Implementing a Remote Debugging Agent Using the GNU Debugger*, 2001.

[Harr97] Harries, L., *Interfacing to The IBM-PC Parallel Printer Port*, 1997.

[Inte01] Intel Corp., *Intel XScale Microarchitecture Assembly Language Quick Reference Card ARM Instruction Set*, 2001.

[Inte02ip] Intel Corp., *PXA250 and PXA210 Application Processors*, 2002.

[Inte03ix] Intel Corp., *XScale Microarchitecuture for the PXA255 Processor*, 2003.

[JoPu02] Johnson, M., and N. Puthuff, "Debugging Embedded SoC Systems", *RF Design*, Feb. 2002.

[KiKL03] Kim, C., H. Kim, and C., Lim, *Technology Trends and Development Strategies on Embedded Software for Ubiquitous Computing Era*, 2003.

[Laut03] Lauterbach Ltd., *RTOS-Linux*, 2003.

[Lee01] A. Lee,E., *Embedded Software*, 2001.

[Pete99] H. Peters K., "Software Development and Debug for System-On-A-Chip", *Embedded Systems Conference*, 1999.

[RuCo01] Rubini, A., and J. Corbert, *Linux Device Drivers Second Edition*, O'Reilly & Associate, June 2001.

[StPS03] Stallman, R., R. Pesch, and S. Shebs, et al., *Debugging With GDB*, 2003.

[Vink98] Vink, G., "Trends in Debugging Technology", Embedded Systems Conference East, March. 1998.

[Yagh03] Yaghmour, K., *Building Embedded Linux Systmes*, O'Reilly & Associate, 2003.