Qplus-T RTOS를 위한 원격 멀티 태스크 디버거의 개발

(Development of a Remote Multi-Task Debugger for Qplus-T RTOS)

이 광 용 † 김 흥 남 ††
(Kwang-Yong Lee) (Heung-Nam Kim)

요 약 본 논문에서 인터넷 정보가전과 같은 Qplus-T 내장형 시스템을 위한 멀티 태스크 디버깅 환경에 대해 제안한다. 효과적인 교차 개발을 지원하기 위해 원격 멀티 태스크 디버깅 환경의 구조 및 기능들을 제안할 것이다. 그리고, 좀더 효율적인 교차 개발 환경의 개발을 위하여 호스트-타겟 사이에 디버깅 커뮤니케이션 아키텍처를 개선할 것이다. 본 논문에서 제안하는 Q+Esto라는 원격 개발 도구들은 대화형 쉘, 원격 디버거, 리소스 모니터, 타겟 매니저, 그리고 디버그 에이전트들과 같이 몇 개의 독립된 도구들로 구성된다. 호스트에서 원격 멀티 태스크 디버거를 이용해서, 개발자는 타겟 실행 시스템 위에 태스크들을 생성시키거나 디버그 할 수 있으며, 실행 중인 태스크들에 접속하여 디버그 할 수 있다. 응용 코드는 C/C++ 소스레벨로 볼 수 있으며, 어셈블리 레벨 코드로도 볼 수 있다. 그리고, 소스코드, 레지스터들, 지역/전역 변수들, 스택 프레임, 메모리, 그리고 사건 트레이스 등등을 위한 다양한 디스플레이 윈도우들을 포함하고 있다. 타겟 매니저는 Q+Esto 도구들에 의해 공유되는 공통된 기능 즉, 호스트-타겟 커뮤니케이션, 오브젝트 파일 로딩, 타겟 상주 호스트 메모리 풀의 관리, 그리고 타겟 시스템 심볼 테이블 관리 등등의 기능들을 구현한다. 이러한 기능들을 개방형 C API라고 부르는데, Q+Esto의 도구들의 확장성을 크게 개선한다. 그리고, 타겟 매니저와 타겟 시스템 커뮤니케이션을 위한 상대파트 모듈 즉, 디버그 에이전트가 존재하는데, 이것은 타겟의 실시간 운영체제 위에서 데몬 태스크 형태로 수행된다. 디버거를 포함한 호스트 도구로부터의 디버깅 요청을 받아, 그것을 해석하고 실행하여, 그 결과를 호스트에 보내는 기능을 수행한다.

키워드 : 실시간운영체제, 정보가전, 원격멀티태스크디버거, 실시간시스템, 내장형시스템, 큐플러스-T 실시간운영체제, 큐플러스-에스토, 교차개발환경

Abstract In this paper, we present a multi-task debugging environment for Qplus-T embedded system such as internet information appliances. We will propose the structure and functions of a remote multi-task debugging environment supporting effective cross-development. And, we are going to enhance the communication architecture between the host and target system to provide more efficient cross-development environment. The remote development toolset called Q+Esto consists of several independent support tools: an interactive shell, a remote debugger, a resource monitor, a target manager and a debug agent. Excepting a debug agent, all these support tools reside on the host systems. Using the remote multi-task debugger on the host, the developer can spawn and debug tasks on the target run time system. It can also be attached to already-running tasks spawned from the application or from interactive shell. Application code can be viewed as C/C++ source, or as assembly-level code. It incorporates a variety of display windows for source, registers, local/global variables, stack frame, memory, event traces and so on. The target manager implements common functions that are shared by Q+Esto tools, e.g., the host-target communication, object file loading, and management of target-resident host tool's memory pool and target system's symbol-table, and so on.

† 정 회 원 : 한국전자통신연구원 임베디드S/W기술센터 연구원
kylee@etri.re.kr
†† 비 회 원 : 한국전자통신연구원 임베디드S/W기술센터 센터장
hnkim@etri.re.kr
논문접수 : 2002년 6월 28일
심사완료 : 2003년 5월 7일

These functions are called OPEN C APIs and they greatly improve the extensibility of the Q+Esto Toolset. The Q+Esto target manager is responsible for communicating between host and target systems. Also, there exist a counterpart on the target system communicating with the host target manager, which is called debug agent. Debug agent is a daemon task on real-time operating systems in the target system. It gets debugging requests from the host tools including debugger via target manager, interprets the requests, executes them and sends the results to the host.

Key words : Real-Time Operating System, Information Appliance, Remote multi-task debugger, Real-time System, Embedded System, Qplus-T RTOS, Q+Esto, Cross-development environment

1. 소 개

최근 정보기기의 급속한 발전과 많은 수요로 인해서 실시간 OS에 대한 기술 개발의 필요성이 대두되었고, 많은 실시간 OS 제품이나 개발도구가 개발되었다[1,2]. 그러나, 기존의 제품들은 주로 VRTX, pSOS, VxWorks 등과 같은 산업용 실시간 OS에서 파생되거나 Windows CE나 QNX와 같이 기존의 PC나 Workstation OS로부터 다운사이징 한 제품들이며, 이는 가전용 제품에 적합한 조립형 실시간 OS로써는 최적의 OS라 할 수 없다. 그리고, 최근 몇몇 업체와 학교, 연구소 등에서 실시간 OS 개발을 시도하고 있으나 국내의 실시간 OS 기술은 연구개발이 초기 단계에 있고, 외국 실시간 OS는 범용성을 중심으로 지원하기 때문에 가전용품에 대한 전문적인 지원에는 한계가 있다. 또한, 국내 업계는 점차적으로 가전용 제품의 국산화율을 높이고자 노력하고 있는데 현재 가전용 제품의 추세로 보아 조만간 많은 제품들이 실시간 OS를 필요로 하게 될 것이다. 그러나, 외국 제품의 실시간 OS들을 사용함으로써 기술료를 부담한다는 것은 국내 가전용 제품의 경쟁력을 급속히 약화시킬 우려가 있다. 이에, 본 실험실에서는 지난 수년동안 조립형 실시간 OS 개발 과제에서 인터넷 정보가전 제품을 위한 태스크(Task) 기반 Qplus-T RTOS 및 이와 연동하는 원격 디버깅 개발환경인 Q+Esto를 개발하였다. 현재는 프로세스(Process) 기반의 실시간 운영체제인 Qplus-P RTOS 커널을 위한 원격 멀티쓰레드 디버거를 개발 중에 있다[1-4].

2. Qplus-T 원격 멀티태스크 디버깅의 문제점 및 구현방안

2.1 문제점

정보가전 제품에서와 같은 내장형 시스템, 특히 Qplus-T RTOS를 포함하고 있는 내장형 시스템에서 수행되어야 하는 응용 프로그램들은 프로그램 크기는 작을지 모르나 개발자의 실수를 눈감아 주지 않는 개발하기 힘든 시스템이다. 사소한 실수에도 시스템이 멈춰버리게 되며, 특히 포인터를 잘못 사용할 경우를 대비하여 에러를 검출하는 인터럽트 처리 루틴이나 최종 상태나 에러 여부를 기록하는 진단 함수들을 개발하는 것이 중요하다. 그리고, 하드웨어 에뮬레이터와 같은 디버거를 사용하는 경우는 좀 더 나은 상태의 디버깅이 되겠지만, 이런 경우에도 assert와 같은 디버깅용 함수를 곳곳에 삽입하는 것이 좋다. 특히, 내장형 프로그램의 대표적인 예인 멀티 태스크 프로그램을 원격지에서 디버깅할 때 여러 가지 문제들이 존재하지만, 다음에 제시하는 문제들이 그 대표적인 문제들로 볼 수 있다[5-9].

① 비결정성 문제 : 비결정성 문제는 멀티 태스크 프로그램에서 공유자원을 액세스하는 태스크들에 의해 초래된다. 멀티 태스크 프로그램의 경우 그 실행을 달리할 때마다 공유자원 접근 순서가 달라지므로 실행결과도 매번 달라질 수 있다.

② 디버깅 정보의 증가 : 멀티 태스크 프로그램의 디버깅을 위해서는 시퀀셜 프로그램을 디버깅 할 때 보다 각각의 태스크들 별로 디버깅 정보를 관찰해야 하므로 디버깅 정보가 많아진다. 다음은 관찰해야 될 정보들이다.
- 태스크들의 내부 상태정보
- 태스크들간의 메시지 교환정보
- 태스크들간의 동기화 정보
- 공유 자원들의 접근상태

③ 전체시스템의 상태파악 문제: 비록 디버깅 시스템이 디버깅 정보를 잘 수집했더라도 사용자가 그 정보들 및 타겟 시스템의 상태를 빠르게 이해할 수 없다면 그것은 디버깅을 위해 쓸모 없는 정보가 될 것이다.

④ 실시간 성 문제 : 대부분의 정보가전 시스템은 강성 실시간 시스템(hard real-time)이라기보다는 연성 실시간 시스템(soft real-time)에 가깝지만 다양한 미디어를 통해 수신된 메시지들의 동기화를 맞춘다든

지 하는 실시간 성의 보장이 필요하다. 적절한 타이밍을 맞추지 못한 메시지들은 정보가전 제품의 품질을 떨어뜨릴 수 있다.

2.2 구현방안

본 논문에서 제시하는 원격 멀티태스크 디버거는 원격 개발환경에서 개발자들에게 효과적이고 용이한 디버깅 환경을 지원하기 위하여 특히, 앞 절에서 제시한 문제점 중 2번과 3번 문제들의 해결을 위해서 다음과 같은 개념을 기반으로 개발한다.

첫째, 기존의 독자(stand alone) 환경에서 익숙하게 사용하던 기본적인 디버깅 기능들이 원격(remote) 환경에서도 제공될 수 있게 한다. 정지점(breakpoint) 설정, 관찰점(watchpoint) 설정, 스택내외 이동, 한 스텝씩 실행(single-stepping) 기능과 실행중인 프로그램의 이미지를 읽어서 "Call Stack Trace", "Watch Expression", "Inspect Data" 등과 같은 정보를 보여주는 기능 등이 여기에 해당한다.

둘째, 유연하고 신뢰성 있는 구조를 가질 수 있게 한다. 원격 멀티 태스크 디버깅 환경에서는 원격지인 호스트에서 발생시킨 디버깅 명령들이 타겟 보드에 적재되어 실행되는 목적프로그램에 적용되어야 하기 때문에 기본적으로 클라이언트-서버 아키텍처를 갖추게 된다. 또한 호스트나 타겟의 OS 발전에 대비할 수 있도록 계층적 형태의 아키텍처의 구성과 기능이 추가되거나 변경되었을 때 유연하게 대처할 수 있도록 블록(객체)단위로의 구성이 필요하다. 특히 시스템 전체의 제어흐름과 이해를 명확히 하기 위해 디버깅 상태를 관리할 수 있도록 하며, 원격 소프트웨어 개발환경에 대한 신뢰성을 개발자에게 제공할 수 있는 구조가 되어야 한다. 셋째, 효과적인 멀티태스크 디버깅에 대한 지원이 가능하게 한다. 멀티 태스크들에 의해 유발되는 오류는 디버깅하는 것은 단일 태스크(single task)에 대한 디버깅에 비해 매우 어려운 작업이다. 이러한 어려움을 감소시키기 위하여 이미 실행 중인 태스크에 대한 디버깅을 할 수 있도록 정시교정(Just-In-Time Debugging)과 같은 멀티 태스크들의 동작을 효과적으로 관찰할 수 있는 기능이 지원되어야 한다. 그러나, 이러한 기능은 목적프로그램에 대한 디버깅 정보의 삽입으로 인해 발생할 수 있는 탐침 효과(probe effect)가 극소화되도록 할 필요가 있다.

넷째, 효과적인 사용자 인터페이스 환경을 제공한다. 앞서 제시한 정지점 기반 멀티태스크 디버깅 작업이 가능한 통합된GUI를 통해 이루어질 수 있도록 해야 하며, 진단명령(Diagnostic Commands)처럼 GUI를 통하여 사용할 수 없는 입/출력이 어려운 기능을 위해 커맨드라인 인터페이스가 제공될 필요가 있다.

3. Qplus-T RTOS를 위한 원격 디버깅 개발 환경 구조

3.1 교차 개발 환경

Qplus-T RTOS를 위한 교차-개발(cross-development) 환경은 다음 그림 1에서와 같다. 이 그림에서와 같이 교차-개발 환경은 교차 디버거가 수행되는 호스트 시스템과, 디버깅 모니터를 수행하는 타겟 시스템(여기에서는 SA110을 탑재한 EBSA21285 보드를 지칭함)[10], 그리고 이들을 연결하는 시리얼 라인 혹은 이더넷 라인으로 구성된다. 대부분의 원격 디버깅 환경에서는 시리얼라인과 이더넷라인을 동시에 활용한다. 이의 구현 형태를 좀 더 자세히 살펴보면 호스트 시스템은 Windows NT/95/98을 플랫폼으로 하는 범용 컴퓨터로 구성되어 있으며, 이 호스트 컴퓨터에서 사용하는 원격 디버거는 Cygnus에서 제공하는 GNU GDB [11]를 수정/보완하여 사용한다. 타겟시스템은 EBSA 21285 하드웨어를 기반으로 구성되어 있으며, 타겟위에서 하드웨어 시스템 초기화 루틴들과 커널 시스템을 적재시키기 위한 단순한 로직들로 구성된 커널 모니터링 시스템으로 구성되어 있다. 이 커널 모니터는 GNU GDB와 연동하여 동작한다. 여기에서 GNU GDB를 호스트용 교차 디버거로 주로 사용하는 이유는 이 GDB는 공개된 소프트웨어로 로열티에 문제가 없으며, 다양한 타겟을 지원 가능한 형태로 구조화되어 있고, 디버그 자체 버그를 그동안 많이 수정/보완이 되었기 때문이다. 그림 1과 같은 구성에서 교차 디버깅은 호스트 시스템에서 타겟 시스템용으로 교차 컴파일된 실행파일을 타겟보드로 전송한 후 호스트에서 타겟에 적재된 실행파일을 제어하면서 디버깅을 수행한다.

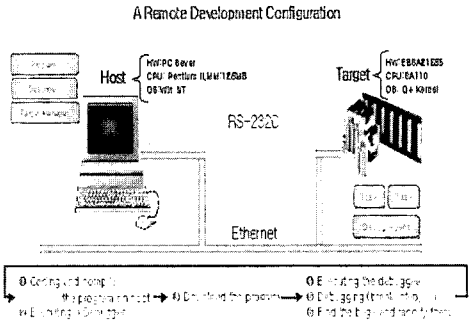

그림 1 커널 교차 디버깅 환경

3.2 태스크 기반 Qplus-T RTOS

Qplus-T RTOS는 한국전자통신연구원에서 1998.11-2000.12 사이에 개발한 정보가전용 실시간 운영체제로 디지털 TV와 인터넷 TV 등에 활용이 가능한 기능을 갖추고 있으며, 국제 표준 및 업계 표준과의 호환성을 최대한 보장한다. 또한 이 Qplus-T RTOS는 다양한 정보가전기기들에 쉽게 활용될 수 있게 하기 위하여 아래 그림 2와 같이 조립성 및 확장성을 가질 수 있는 다계층(Multi-layer) 구조로 구성되어 있다[3,4].

타겟 하드웨어와의 인터페이스 계층인 하드웨어 추상화 계층(HAL:Hardware Abstraction Layer)은 커널의 하드웨어 의존성을 피하기 위한 계층으로서 CPU와 메모리, 그리고 타이머 등과 같은 핵심 하드웨어 요소에 대한 추상화 함수를 제공하여 이식성을 높인 부분이다. 그 다음 계층은 기본 커널(PK: primitive kernel)로 쓰레드를 기반으로 하는 요소들로 구성되어 있으며, 기본 스케줄러 및 동기화 메커니즘을 제공한다. 이 부분은 Qplus-T의 핵심 부분이라 할 수 있으며 신뢰도가 높은 RTOS인 VRTX[12]의 나노커널(Nanokernel)을 기반으로 구현된 부분이다. Qplus-T 커널 계층(QP: Qplus-T kernel)은 태스크들 및 ISR(Interrupt Service Routine)들 사이의 통신 및 동기화 메커니즘을 위해 프로세서간 통신을 제공하며, 태스크와 네트워크, 그리고 메모리 관리에 대한 사용자 API를 제공한다. 그리고 I/O 서브시스템(IOS)은 I/O 디바이스를 통한 데이타 입출력의 일관성 유지를 위한 POSIX 호환 인터페이스 부분으로서, I/O 디바이스 드라이버의 등록과 제거를 용이하게 해준다.

3.3 ESTO 개발도구

Qplus-T의 통합개발도구인 Q+Esto는 원격지 호스트(remote host)에서 목적프로그램(object program)을 개발하여 타겟(target) 보드에 적재/실행하고, 실행상태와 자원사용을 추적/관찰하며 발생하는 오류의 원인을 찾아 제거하도록 도와주는 개발환경이다. 이 개발환경에는 크로스 컴파일러, 유틸리티, 원격 디버거, 대화형 쉘, 자원 모니터 등의 도구들과 타겟 매니저, 디버그 에이전트 등으로 구성되어 있다. Q+Esto에서 제공되는 크로스 컴파일러는 윈도우 환경에서 작동하는 gcc 컴파일러로 ANSI C 형태로 작성한 프로그램을 ARM 계열의 StrongARM SA-110 혹은 1110 마이크로프로세서(micro-processor)에서 실행할 수 있는 형태의 ELF (Executable and Linking Format) 파일 서식을 만들어 준다. 그리고, 원격 디버거는 호스트에서 타겟에 적재된 프로그램을 실행시키는 과정에서 오류를 쉽게 발견하고 수정할 수 있게 해주는 기능을 가진다. 또한, 대화형 쉘은 타겟의 메모리에 목적프로그램을 적재하거나 제거할 수 있으며 타겟에 적재된 커널과 라이브러리 응용 API의 어떠한 함수도 호출하고 상호작용을 할 수 있게 한다. 자원모니터는 조립형 실시간 운영체제가 탑재되는 타겟 시스템의 상태와 운영체제 자원에 대한 정보를 원격에서 제공한다. 한편, 타겟 매니저와 디버그 에이전트는 각각 호스트와 타겟에 상주하여 상호 통신을 통해 호스트와 타겟간의 인터페이스를 구성해주는 디버깅 미들웨어이다. 그림 3에서 볼 수 있듯이, 특히 디버그에이전트는 커널 레벨에 존재하고 있고, Qplus-T RTOS의 플랫 메모리 관리 특성상 응용 프로그램뿐만 아니라 커널 레벨의 소스들도 디버깅 가능하게 되어 있다. 본 논문에서는 원격지에서 멀티 태스크 디버깅을 위한 개발환경 중심 즉, 그림 3에서 보듯이 원격 디버거, 타겟매니저, 디버그에이전트의 구현 기술 중심으로 설명한다.

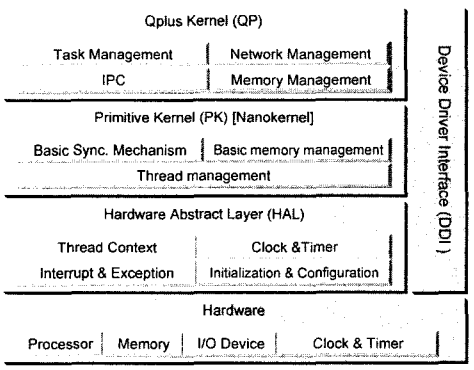

그림 2 Qplus-T 커널구조

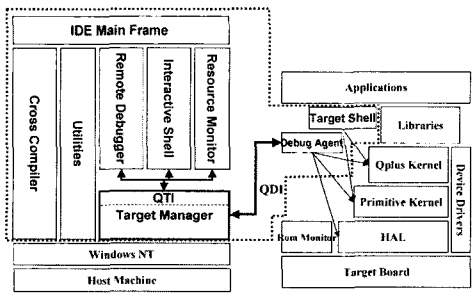

그림 3 Q+Esto 원격 개발환경 시스템 아키텍처

4. 원격 멀티태스크 디버거 구현 기술

원격 멀티 태스크 디버깅은 원격지에서 타겟보드에

적재된 멀티 태스크 프로그램을 실행시키는 과정에서 오류를 발견하고 수정할 수 있게 하는 기법이다. 이를 위해, 일반적으로 디버거가 직접 실행 중인 특정 태스크를 선택, 정지시킨 후 그 태스크의 컨텍스트(context)를 시퀀셜 디버깅과 마찬가지 방법으로 step, next, continue 등과 같은 디버깅 명령어를 이용하여 디버깅할 수 있어야 한다. 특히, 앞 장에서 설명한 Qplus-T RTOS에서의 원격 멀티 태스크들의 태스크 핸들링 기법은 아래와 같은 특징을 갖고 있어 구현 시 고려해야 한다.

- 태스크들은 RR-스케줄러에 의해 컨트롤을 갖게 됨.
- Linux에서의 쓰레드 핸들링과 같이 어떤 태스크에 브레이크 인터럽트가 걸려도 모든 태스크가 임의의 위치에 정지하는 것이 아니라, Qplus-T RTOS에서의 태스크들의 핸들링은 브레이크 걸리게 된 태스크만 잠시 멈추게(suspend) 됨.

따라서, 본 논문에서의 멀티 태스크 디버깅 기능은 다음과 같이 멀티 컨텍스트 디버깅(*multi-context debugging*)이 가능한 형태로 구현하였다.

- 사용자가 호스트 디버거에서 디버깅하기를 원하는 쓰레드로 스위칭 한 후에야 step, next, 등등의 디버깅이 가능함.
- 어떤 태스크에 브레이크 인터럽트가 걸려도 현재 스위칭이 되서 사용중인 태스크인지에 따라 인터럽트 정보를 호스트의 디버거에 알려주는 형태로 구현함. 즉, 태스크 스위칭이 되면, 그 때서야 비로소 태스크의 상태를 호스트 디버거에게 알려주게 함.

4.1 디버거 구조

그림 4는 Q+Esto원격 디버거의 내부 구조를 보여준다. 이 교차 디버거의 내부는 멀티 태스크 디버깅 GUI 모듈, 디버깅 엔진 모듈, 디버깅 미들웨어 모듈, 그리고 타겟 디버깅 모듈 들로 구성되어 있다. 먼저, 멀티 태스크 디버깅 GUI모듈은 태스크 윈도우 및 디버거 메인 윈도우를 통해 개발자로부터의 디버깅 입력을 받아 그것을 디버깅 매니저에 의해 디버깅 컴맨드로 변환하여 디버깅 엔진으로 전달하고, 디버깅 엔진의 처리 결과를 수집하여 윈도우에 전시하는 기능을 수행한다. 태스크 윈도우와 메인 디버거 윈도우는 소켓 스트림을 통해 컴맨드들을 주고 받게 되고 있으며 그림 4에서 보듯이, 멀티 컨텍스트 디버깅을 위해 메인 디버거 윈도우 하나에 다수개의 태스크 윈도우에서 클라이언트-서버 아키텍쳐로 연결되어 있다. 그리고, 이 GUI 모듈과 하위 모듈인 디버깅 엔진 모듈의 연결은 스트림 파이프 프로세스를 통해 연결하여 마치 디버거를 컴맨드 콘솔에 백그라운드 데몬 형태로 띄워서 처리하는 것으로 만들었다. 이러한 형태의 장점은 서로 독립적인 형태로 만들어진 GUI 모듈과 실행파일(*.exe 파일)을 쉽게 통합시킬 수 있다는 것이다.

다음으로, 디버깅엔진 모듈은 디버깅 정보를 읽어 내부 심볼 테이블을 구성하고 사용자와의 인터페이스를 통하여 소스 파일을 디버깅한다. 이 모듈에서는 상위 GUI 모듈로부터의 다양한 컴맨드 스트림을 받아 다양한 연산 및 타겟의 상태를 저장/처리 하는 부분으로 구성되어 있으며, 무엇보다 중요한 작업은 디버깅 미들웨어와의 공용 프로토콜을 사용하여 타겟을 제어하는 리모트 디버깅 인터페이스이다.

본 논문에서 제시하는 원격 멀티 태스크 디버거에서는 디버깅 미들웨어 모듈과 디버깅 엔진모듈을 연결하기 위하여 QTI 즉, Qplus Tool Interface 프로토콜을 사용한다. 이 QTI 프로토콜은 멀티 태스크 윈도우들에서의 멀티 컨텍스트 디버깅 메시지를 타겟의 디버깅 에이전트에 일관성있게 연결하는 통합채널 역할을 담당한다. 일종의 디버깅 미들웨어인 타겟 매니저 모듈은 호스트의 전처리 기능으로 디버거뿐만 아니라 다른 도구들에서 사용 가능한 공용 QTI 프로토콜을 제공하고 있으며, 후처리 기능으로 타겟과의 실제적인 디버깅 연결을 담당하는 QDI 프로토콜을 제공한다. 또, 내부 기능으로써는 심볼 테이블 관리 기능, 메모리 캐싱기능 등 여러 가지 기능들이 있다.

QDI는 Qplus Debugging Interface의 약자로 원격지에 떨어져 있는 디버깅을 위한 디버그 에이전트 모듈과 RPC(Remote Procedure Call)를 통해 연결을 담당하는 기본적인 인터페이스 기능들을 제공한다. RPC를 사용함으로써의 장점은 호스트의 디버깅 미들웨어에서 사용한 구조체 및 함수를 그대로 타겟에서도 사용할 수 있다는 것이다.

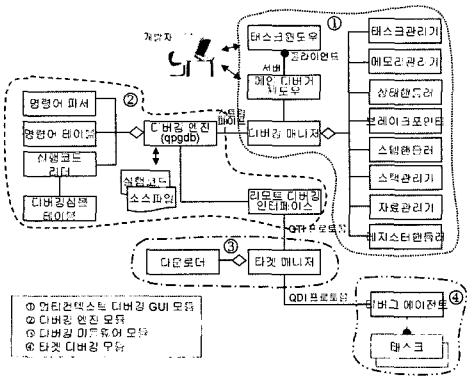

그림 4 멀티 태스크 디버거 구조

4.2 멀티 태스크 디버깅 GUI

디버거 GUI 모듈은 사용자로부터 디버깅 명령을 입력으로 받아 명령의 종류별로 디버깅 매니저에 보고하고 디버깅 매니저의 처리결과를 받아 사용자에게 그래픽하게 전시하는 역할을 담당한다. 그림 5에서 보듯이, 이 모듈에는 기본적으로 디버깅 소스코드와 리소스 정보를 보기위한 디버깅 메인 윈도우(main-window)와 멀티 태스크들을 위한 태스크 윈도우(task-window)로 구성되어 있다. 예를 들어, 멀티 태스크 디버깅을 위한 처리 메카니즘을 보면, 그림 5에서와 같이 ①→②→③→④→⑤의 순으로 처리되는데, ①의 스텝버튼 이벤트에 대해 "CThreadWndView"의 "OnDebugStep()" 함수가 반응하게 되고, 이 함수에서는 차례대로 "DebugManager::WinStep()", "StepHandler ::StepCommand()", 그리고, "CCommExClient::SetCmd()" 함수들을 호출하여, ②의 스텝 컴맨드(cmd)를 "Task Window 1"으로부터 "Main Window"로 전달한다. 그 다음 "Main Window"에서는 이 컴맨드에 반응하여 ③의 클라이언트 쓰레드(ClientThread)를 생성하고, 이 쓰레드에서는 디버거 매니저 모듈을 통해 ④과 같이 qpgdb 엔진에 직접 "step" 컴맨드를 내림으로써 ⑤의 스텝핑을 처리하고 그 결과를 입수하여 역순으로 태스크 윈도우로 그 결과를 전달하여 태스크에서의 스텝핑을 처리한다. 이 그림에서 보면, "Main Window" 혹은 "Task Window"에서의 디버거 매니저를 다루는 구조는 동일한 형태를 취하나 "Task Window"에서의 소켓을 통한 컴맨드 송수신을 하는 세부처리 방식과 "Main Window"의 디버거 매니저에서의 파이프 프로세스(pipe process)를 통한 컴맨드 처리 방식은 다르게 구현되어 있다. 이처럼 구현한 것은 객체지향 개념에 맞게 "Task Window"와 "Main Window"는 동등한 레벨의 객체이기 때문이며, "Main Window"와 qpgdb 엔진과는 서로 다른 레벨의 객체들이기 때문이다. 그리고, 무엇보다 GUI 레벨의 객체들은 VC++로 구현되었고, qpgdb 엔진의 경우는 GNU에서 제공하는 gcc 컴파일러로 구현되어 있어 서로 공통의 라이브러리를 공유할 수 없는 구조로 되어 있기 때문이다. 이들의 기능들을 하나로 묶기 위해 파이프 프로세스(pipe process)를 통한 두 프로세스간의 IPC(Inter Process Communication) 구현을 이루었다. step 컴맨드 외에 continue, next, finish 등의 컴맨드들도 동일한 방식으로 구현을 한다. 이처럼, 본 논문에서 제시하는 멀티 태스크 디버거는 태스크 윈도우와 메인 윈도우 사이에 클라이언트-서버 아키텍쳐(client-server architecture) 형태로 명령어 통신모듈을 이용하여 명령어를 주고 받는 형태로 구현되어 있다. 이 클라이언트-서버 구조는 타겟의 병행적인 실행 상태를 구조적으로 반영하기 용이하게 함으로써 메인 윈도우 및 태스크 윈도우에서의 디버깅 처리 오버헤드를 최소화시킬 수 있는 장점이 있다. 그러나, 단점은 소켓을 통한 클라이언트-서버 통신 오버헤드로 인해 디버깅 성능이 떨어질 우려가 있으나 현재, 적용 실험한 결과는 이 부분에서는 문제가 되고 있지 않다.

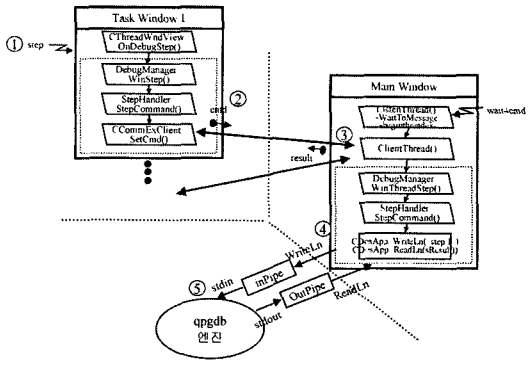

그림 5 멀티 태스크 디버깅 GUI 구조

디버깅매니저(DebugManager) 모듈은 사용자 인터페이스 모듈로부터의 명령을 해석하고 이를 실행 엔진인 디버깅 엔진(qpgdb 엔진)에게 전달하여 디버깅 명령을 수행하고 그 결과를 사용자 인터페이스 모듈에 전달하는 중간 브리지 역할을 담당하는 객체들로 구성되어 있다. 특히, 디버깅 엔진의 실행 상태를 관리하기 위해 상태 핸들러를 독립적인 객체로 두고 있기 때문에 사용자 인터페이스의 디버깅 과정을 통제할 수 있도록 되어 있다. 이 모듈에는 디버거 GUI 모듈의 여러 관점들을 지원하기 위한 특별한 객체들로 구성된다.

디버거 매니저 클래스는 여러 개의 유닛 객체들로 구성되어 있다. 기본적으로 디버깅 매니저 유닛 객체에서는 디버깅 엔진 초기화, 프로그램 실행, 오브젝트 모듈 로딩, 타겟과의 연결 등등의 기능을 수행하고, 브레이크 포인트 유닛에서는 브레이크의 설정 해제 등과 관련된 기능을 수행하고 있으며, 스텝핸들러 유닛 객체는 step, next, continue, finish 등의 명령어를 다룬다. 그 밖에 리소스 관리를 위한 스택관리 유닛, 자료관리 유닛, 레지스터 및 메모리 관리 유닛, 태스크 관리 유닛 등이 있다. 구체적으로, 디버거 GUI로부터의 WinStep 함수 호출에 반응하여 디버거 매니저에서 처리하는 과정을 살펴보면 다음과 같다.

```
int DebugManager::WinStep(CString& args)
{
    int lineNumber;
    // step !!!
    CString qdb_msg;
    CString res = m_pStepHandler->StepCommand(args);
    lineNumber = 0;
    // gdb의 출력 메세지를 분석한다.
    while(!res.IsEmpty()) {
        if(res.Find("Breakpoint") != -1 &&
        res.Find(',') != -1 && res.Find(':') != -1)
        ...
        else if(res.Find("Breakpoint") != -1 &&
                res.Find(',') != -1)
        ...
    }
    ...
    return lineNumber;
}

CString StepHandler::StepCommand(CString args)
{
    // initalize data
    CString command = "step "+args; // step [ count]
    CString sResult = "";
    ...
    // stepping
    m_pDosApp->WriteLn(command);
    // read result stream
    while(1)
    {
        BOOL succ = m_pDosApp >
            ReadLnWithThread(rdBuffer, BUFSIZE-1, nRdBytes);
        qdb_prompt = _T(rdBuffer);
        if(qdb_prompt.Find("__EOM__") >= 0) break;
        ...
    }
    // return
    return sResult;
}
```

호출

그림 6 디버거 매니저 클래스의 스텝핑 처리

디버거 GUI로부터의 WinStep 함수 호출에 의해 프리미티브한 스텝 핸들러 유닛 객체가 동작하게 되고, 스텝 핸들러 객체는 WinStep 함수에 대응하는 StepCommand 함수를 호출한다. 이 StepCommand 함수에서는 CDosApp 객체의 WriteLn 함수를 이용하여 qpgdb 엔진에 스텝 컴맨드를 내리고, ReadLnWithThread 함수를 이용하여 그 결과를 입수하여 스트링 형태로 상위 모듈로 리턴한다. 그 다음 입수된 스트링을 해석하여 스텝핑 라인번호(lineNumber)정보를 상위 호출모듈로 전달한다.

4.3 디버깅 엔진

디버깅 엔진 모듈은 디버깅 매니저로부터의 통제에 따라 실제 디버깅 명령을 수행하는 모듈과 타겟과의 통신을 위한 리모트 디버깅 인터페이스 모듈 들로 구성되어 있다. 디버깅 엔진 모듈은 GNU gdb 버전 4.17을 수정/보완하여 개발하였으나, 본 실험실에서 개발하고 있는 Qplus-T RTOS 환경에서의 멀티 태스크 디버깅 기능을 지원하기 위해, 예를 들면, 태스크 스위칭 기능(예, thread 10), 태스크 목록 보기 기능(예, info thread), 등등을 GNU gdb에서 제공하고 있는 호스트 gdb와 타겟 gdbserver간의 "remote" 프로토콜을 사용하여 구현하지 않고, 대신에 본 논문에서 제안하고 있는 "qti" 프로토콜을 사용하여 새롭게 구현하였다. 또한, 타겟과 호스트간의 원격 디버깅 인터페이스 예를 들면, 타겟의 메모리 값을 읽고 쓴다든지 혹은 타겟의 레지스터 값을 읽고 쓴다든지 하는, 등등의 기본적인 디버깅 기능들을 구현하기 위해 qti 프로토콜을 사용하는 리모트 디버깅 인터페이스 모듈을 새롭게 구현하였다.

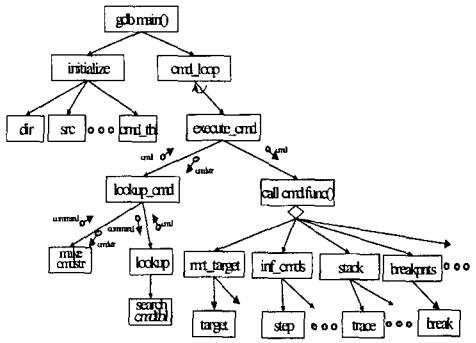

그림 7 디버깅 엔진 기능

다음은 이들의 구현 기법에 대하여 하나씩 설명한다. 먼저, 디버깅 엔진은 명령어 파서 객체, 명령어 테이블, 실행코드 리더, 디버거 심볼 테이블 들로 구성되어 있다. 다음 그림 7에서처럼 디버깅 디렉토리, 소스파일, 명령어 테이블, 그리고 심볼 테이블 등을 엔진 시작과 동시에 초기화하고 나서 실제 디버깅 명령을 입력으로 하여 필요한 기능을 수행하는 구조로 되어 있다. 예를 들어, step 명령어의 경우 컴맨드 심볼 테이블(cmd_tbl)로부터 해당 컴맨드 함수를 찾은(lookup_cmd) 다음 내부 컴맨드 모듈(execute_cmd)에서 그에 맞는 함수 즉, step_1 함수를 호출하는 방식으로 처리된다. 그림 8에서 보듯이, step_1 함수에서의 처리는 응용 프로그램이 멈추어져 있는 현재 스택 프레임(current stack frame)을 찾은 다음 그 스택의 현재 pc(program counter)에서 다음 스텝 주소로 진행(proceed)하는 방식으로 되어 있다. 물론, 실제 pc의 진행은 타겟의 디버그 에이전트에 의해 이루어진다. proceed 함수는 스텝핑해서 멈추어야 할 다음 주소에 먼저 브레이크포인트를 설정(insert_breakpoints)한 후, 타겟의 응용 프로그램을 재실행(resume)시키고 그 결과를 기다렸다(wait_for_inferior) 디스플레이(normal_stop) 해주는 방식으로 처리된다. 이 알고리즘에서 insert_breakpoints 함수, resume 함수, wait_for_inferior 함수 등은 각각 리모트 디버깅 인터페이스 모듈의 qpInsertBreakpoint, qpResume, qpWait 함수들과 연결되어 타겟의 디버깅 에이전트에게 해당 기능들을 수행할 수 있게 되어 있다.

```
static void step_1(int skip_subroutines,
                   int single_inst,char *count_string)
{
1.  count = count_string ?
              parse_and_eval_address (count_string) : 1;
2.  for (; count > 0; count--)
3.  {
4.    clear_proceed_status ();
5.    frame = get_current_frame ();
6.    step_frame_address = FRAME_FP (frame);
7.    step_sp = read_sp ();
8.    if (!single_inst) { … };
9.    if (skip_subroutines)  step_over_calls = 1;
10.    step_multi = (count > 1);
11.   proceed ((CORE_ADDR) -1, TARGET_SIGNAL_DEFAULT, 1);
12. } /* end of for */
}


void proceed (CORE_ADDR  addr,  enum target_signal siggnal,
int step)

{
1.    int oneproc = 0;
2.    if (step > 0)
         step_start_function = find_pc_function(read_pc());
3.    if (step < 0)    stop_after_trap = 1;
4.    if (addr == (CORE_ADDR)-1)
5.       if (breakpoint_here_p (read_pc ())) oneproc = 1;
6.    else
7.       write_pc (addr);
8.    if (oneproc) …
9.    resume(oneproc || step || bpstat_should_step(),
             stop_signal);
10.  wait_for_inferior ();
11.  normal_stop ();
}
```

그림 8 qpgdb 엔진의 스텝핑 처리

다음으로, 리모트 디버깅 인터페이스 모듈(remote-qti 모듈)은 교차 디버거의 상위 모듈과 타겟 디버그 에이전트(target debug agent)와의 연결을 담당하는 모듈로써 디버깅 매니저로부터의 디버깅 명령을 받아 디버깅 미들웨어인 타겟 매니저를 통해 타겟 디버그 에이전트에 그 명령을 전달하고 수행결과를 수집하여 상위 모듈들로 전달하는 역할을 담당한다. 예를 들어, 리모트 디버깅 인터페이스 모듈의 qpWait 알고리즘을 보면, 폴링 방식으로 타겟 매니저의 qtiEventGet 함수를 통해 타겟에서 발생된 이벤트 스트링을 받은 다음 그 이벤트를 해석하여 적절한 작업을 수행한다. QTI_EVENT_TEXT_ACCESS라는 이벤트는 타겟에서 브레이크가 발생되었음을 알려주는 이벤트로 호스트에서는 이벤트에 따라 태스크의 컨텍스트 id, 타잎, 현재 위치정보 등의 값을 알려주고 호스트의 리소스 심볼 테이블의 값을 이에 따라 변경한다. qpInsertBreakpoint 함수를 보면, 멀티 컨텍스트 디버깅을 위해 태스크의 브레이크 컨텍스트 id와 브레이크 어드레스 정보를 구하고 이것을 qtiBreakpointAdd()함수를 이용하여 타겟에 전송하는 알고리즘으로 되어 있다. 물론, 타겟에서는 브레이크 컨텍스트 id 및 어드레스 정보를 이용하여 현재 수행중인 태스크에 브레이크 코드를 삽입할 것이다.

한편, 기존 gdb에서 멀티 태스크 디버깅이 가능하도록 다음 그림 10과 같은 taskListBuild() 함수를 추가하였다. taskListBuild() 함수를 보면, 타겟으로부터 태스

```
static int qpWait (int pidToWaitFor,
                   struct target_waitstatus * status)
{
1.  pid = -1;
2.  eventStr = 0;
3.  do {
4.     pEvent = qtiEventGet(hQti);
5.     timeout_flag = qtiErrGet(hQti);
6.     if(pEvent != NULL)  eventStr = pEvent >event;
7.  } while (eventStr == 0 || eventStr[0] == '\0');
8.  switch(qtiStrToEventType(hQti,strtok(eventStr," "))){
9.     case QTI_EVENT_NONE: …
10.    case QTI_EVENT_CTX_EXIT: …
11.    case QTI_EVENT_EXCEPTION: …
12.    case QTI_EVENT_TEXT_ACCESS: // == breakpoint
13.     status >kind = TARGET_WAITKIND_STOPPED;
14.     status->value.sig = TARGET_SIGNAL_TRAP;
15.     pid = qtiStrToContextId (hQti, strtok (0, " "));
16.     ctxType = qtiStrToContextType (hQti,strtok(0," "));
17.     pcRegVal = qtiStrToTgtAddr(hQti, strtok(0, " "));
18.     fpRegVal = qtiStrToTgtAddr(hQti, strtok(0, " "));
19.     spRegVal = qtiStrToTgtAddr(hQti, strtok(0, " "));
20.     supplyRegFromHostVal (PC_REGNUM, pcRegVal);
21.    break;
22.   ...
23. }
24. return pid;
}

static int qpInsertBreakpoint
   (CORE_ADDR addr,char *contents)
{

1.  breakCtxId = inferior_ptid.pid;
2.  token = strtok(contents," \t");
3.  if(token != NULL &&
       (token[0]=='t' && token[1]=='h' && token[2]=='r'))
4.  {
5.        int thread_no;
6.        token = strtok(0," \t");
7.        thread_no = atoi(token);
8.        if(thread_no == 0)
9.                breakCtxId = 0; // BP_ANY
10.       else {
11.               ptid_t ptid;
12.               ptid = thread_id_to_pid (thread_no);
13.               breakCtxId = ptid.pid;
14.       }
15. }
16. if(block_start_addr <= addr && addr <= block_end_addr)
17.       bIsUserArea = TRUE;
18. if(!bIsUserArea) return 0;
19. breakId = qtiBreakpointAdd
              (hQti,modeToCtxType(),breakCtxId,addr);
20. 브레이크포인트 이벤트 리스트에 브레이포인트 정보 추가;
21. return 0;
}
```

그림 9 리모트 디버깅 인터페이스 모듈의 qpWait 및 qpInsertBreakpoint 기능

크 정보를 스트림(stream) 형태로 읽어와서 표 1과 같은 TASK_NODE 구조체 리스트를 만드는 기능을 한다. 이 함수는 타겟의 상태가 변경되면 항상 호출이 되어 호스트에서 타겟의 상태를 일관되게 유지할 수 있게 한다. 즉, 앞서 살펴본 qpWait() 함수에서 어떤 이벤트가 발생하게 되면 이전과 달라진 태스크들의 상태를 반영하기 위하여 이 함수를 호출한다.

표 1 TASK_NODE 구조체

필드	설명
id	태스크 id 번호
valid	태스크의 유효성 boolean 값
name	태스크의 이름
stackBase, stackEnd	태스크의 스택처음 및 스택끝 주소
current	현재 실행중인 태스크인가를 표시하는 boolean값
threadEntry	태스크 실행 시작 주소
regBlockIU	레지스터 블럭 벡터 값
regFp, regSp, regPc	프레임포인트, 스택포인터, 프로그램 카운터 값

```
void taskListBuild (CORE_ADDR sp)
{
1.  MyCoreInit();
2.  // old 리스트로 만들기
3.  oldTaskListHead = taskListHead;
4.  // 태스크 리스트를 형성한다.
5.  taskListHead = 0;
6.  for(index=0;index<QPLUS_MAX_TASK_ID;index++)
7.  {
8.  TASK_NODE *pNode;
9.  if(tsk_list[index]==0) continue;
10. // 타겟으로부터 태스크 정보를 스트림 형태로 읽어온다.
11. task_info = threadInfoGet(tsk_list[index]);
12. if(task_info==NULL)        return;
13. // threadInfoNode 구조체를 만든다.
14. pNode = xmalloc (sizeof (TASK_NODE));
15. // 태스크 id, valid, name 값 구하기
16. value = strtok (task_info, " \t");
17. pNode->id = ...
18. pNode->valid = ...
19. pNode->name = ...
20. // 태스크 stackbase, stackend 값
21. pNode >stackBase = ...
22. pNode->stackEnd = pNode->stackBase + 스택크기;
23. // Figure out whether this task is running or not
24. pNode->current =
25. ((pNode->stackBase <= sp && sp <= pNode->stackEnd) ||
26. (pNode->stackBase >= sp && sp >= pNode->stackEnd));
27. // 태스크 시작주소 구하기
28. pNode->threadEntry = ...
29. // 태스크 기타 정보(pc,sp,fp,regBlockIU 등) 추출
30. pNode->regBlockIU = ...
31. pNode->regFp = ...
32. pNode->regSp = ...
33. pNode->regPc = ...
34. // 리스트 목록에 추가
35. listNodeAdd((LIST_NODE **)&taskListHead, pNode);
36. // task_info 스트림 메모리 할당해제
37. free(task_info);
38. } // end of for_loop
39. // 리스트 목록을 이용하여 gdb 쓰레드 리스트를 업데이트
40. threadListUpdate();
41. // old 태스크 리스트를 제거한다.
42. listFree(...);
}
```

그림 10 리모트디버깅인터페이스 모듈의 taskListBuild 기능

4.4 디버깅 미들웨어 : 타겟 매니저

타겟 매니저는 호스트의 사용자개발도구에 위치하면서 타겟의 디버그 에이전트와 연결하여 응용 프로그램을 개발할 수 있게 하는 타겟 관리자(target manager) 역할을 한다. 현재, 하나의 타겟에는 하나의 타겟 매니저가 동작하도록 하였으며, 호스트 상의 모든 도구는 이 매니저를 통해 타겟에 접근할 수 있게 하는 타겟 브로커로써의 역할을 한다. 타겟 매니저는 타겟과 통신하는 것과 관련된 모든 사항을 관리한다. 다음 그림 11에서 보듯이 왼쪽의 그림에서처럼 기존의 도구와 타겟 시스템의 연결 방식에서는 n개의 도구에 최대 n개의 프로토콜 즉, 호스트-타겟 연결 방식이 필요하기 때문에 도구의 기능을 업그레이드 한다든지 할 때 그에 맞는 프로토콜을 찾아 기능을 새롭게 추가해야 하는 어려움이 존재하였으며, 새로운 도구를 추가한다고 할 때도 새로운 프로토콜을 만들어야 하는 문제가 있다. 그리고, 다양한 프로토콜이 존재하기 때문에 디버거를 만들기 위한 기본 설계 원리인 하이젠베르크의 원리[13,14] 즉, 타겟 응용 프로그램의 탐침효과(probe effect)를 최소화시키기

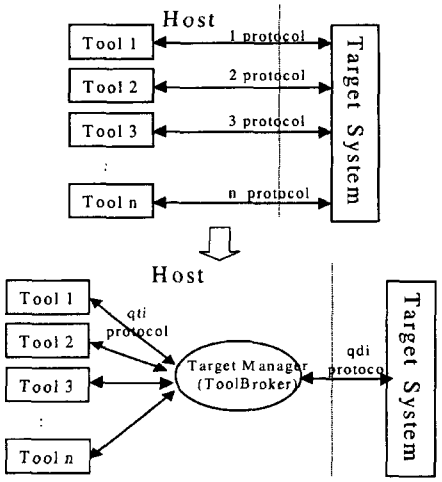

그림 11 디버깅 미들웨어 : 타겟 매니저

위한 원리를 따르기가 어렵다. 또한, 타겟 시스템과의 디버깅을 위한 통신 방법이 다양하기 때문에 통신상의 신뢰성을 확보하기가 어려웠다. 본 논문에서 제시하는 시스템에서는 그림 11의 오른쪽과 같은 형태로 디버깅 미들웨어를 중심으로 호스트 도구들과 타겟을 연결시킴으로써 타겟 접근을 최소화 하여 탐침효과를 최소화시키고, 하나의 타겟에는 하나의 통신 채널만을 존재시킴으로써 디버깅 신뢰성을 확보한다. 그리고, 공용 프로토콜(QTI와 QDI)을 제공함으로써 도구 확장성을 제공하는 것을 가능케 하였다.

먼저, 타겟 매니저의 구성을 살펴보면, 타겟 매니저 블록은 그림 11에서 볼 수 있듯이 호스트 도구들과의 연결을 담당하는 QTI API 부분, 타겟 시스템의 디버그 에이전트와 연결을 위한 QDI 처리모듈, 그리고 디버깅 도구들의 중간 관리자로써 역할을 위한 내부 처리 모듈의 크게 3부분으로 되어있다. 내부 처리 모듈은 다시 그 기능에 따라 심볼 테이블 관리부분, 오브젝트 모듈 관리부분, 호스트-풀 메모리 관리부분으로 나뉜다. 다음에서는 이들의 기능에 대해 간략히 소개한다.

- QTI(Qplus Tool Interface) API 모듈

이 모듈은 타겟 매니저의 전 모듈을 관리하고 사용자가 편리하게 타겟 매니저를 설정할 수 있도록 GUI 인터페이스를 제공한다. 이 모듈을 통해 타겟 매니저를 설정, 시작, 정보 열람 등이 가능하고 여러 타겟 매니저를 관리할 수 있다. 또한 호스트 상의 모든 도구들에게 공통 인터페이스를 제공한다. 각 도구들은 이들 API를 통해 타겟 매니저에게 요구하게 되는데, 타겟 매니저는 타겟 매니저 내부 모듈에게 전달하여 그에 따라 적절한

결과를 반환한다. 도구와 타겟 매니저 간의 프로토콜을 정의하고, 사용자의 편의를 고려하여 API형태로 구현한다. 도구의 요구를 해석하여 그 요구를 호스트 상에서 수행할 것인지 타겟으로 보내기 위해 QDI 백엔드(Back-End) 통신모듈로 보낼 것인 지를 결정한다.

다음은 도구에게 제공하는 APIs의 기능을 나열한 것이다.

- 타겟 레지스터 오퍼레이션, 타겟 메모리 오퍼레이션, 타겟 이벤트 관리, 멀티 태스크를 위한 컨텍스트 관리 및 디버그
- 세션 관리, 타겟 매니저 정보, 실행 모듈 관리, 심볼 관리, 호스트-풀 메모리 관리
- 기타-실행 시간 객체 정보, 서비스 관리 및 등록 오퍼레이션

각 API의 기능을 요약하면 표 2와 같다.

표 2 타겟 매니저의 qti 프로토콜

C API	설명
qtiBreakpointAdd	새로운 브레이크포인트를 추가함
qtiContextCont	타겟의 컨텍스트를 계속 진행함
qtiContextCreate	타겟에 컨텍스트 하나를 생성함
qtiContextExitEventAdd	컨텍스트를 빠져나갈 이벤트포인트를 추가
qtiContextKill	타겟 컨텍스트를 kill
qtiContextResume	중단된 타겟 컨텍스트를 재시작
qtiContextStep	타겟 컨텍스트를 한 스텝 진행
qtiContextSuspend	타겟 컨텍스트 대기(suspend)
qtiEventGet	타겟으로부터 이벤트를 얻어옴
qtiEventpointDelete	타겟에서 온 이벤트포인트를 삭제
qtiEventpointList	타겟 매니저의 이벤트포인트를 나열
qtiMemAddToPool	디버그 에이전트 메모리 풀에 메모리 증가
qtiMemAlloc	호스트에 있는 target memory pool에 메모리 블록 할당
qtiMemChecksum	타겟 메모리 체크썸 실행
qtiMemFree	타겟 메모리 블록 free
qtiMemInfoGet	target-server-managed memory pool 정보 반환
qtiMemRead	타겟 메모리 읽기
qtiMemWrite	로컬 메모리를 타겟에 씀
qtiObjectInfoGet	ID에 해당한 모듈의 정보 반환
qtiObjectLoad	새로운 실행 모듈 로드
qtiObjectUnload	타겟의 실행 모듈 언로드
qtiRegsGet	타겟의 레지스터를 읽어옴
qtiRegsSet	타겟의 레지스터에 데이타를 씀
qtiSymAdd	하나의 심볼의 이름, 값, 타입을 저장
qtiSymFind	심볼의 이름 혹은 값으로 찾기
qtiSymListGet	조건에 맞는 심볼 리스트 반환
qtiSymRemove	타겟 매니저 심볼 테이블에서 특정 심볼 삭제
qtiConnectTargetAgent	타겟을 reattatch

- 내부처리모듈

타겟 매니저는 타겟 시스템 상의 모든 함수와 변수를 가지고 있는 호스트-상주 심볼 테이블을 Build하기 위해 모든 실행 모듈에 관련된 심볼 테이블을 사용한다. Object Module Loader는 외부 참조 및 심볼릭 참조 재배치 등을 위해 심볼 테이블을 사용한다. 호스트 상의 도구들도 타겟 매니저와의 인터페이스를 통해 심볼 테이블에 접근하여 원하는 요구를 실행할 수 있다. 다음으로, Object Module Manager는 도구가 요구하는 가장 핵심적인 기능 중의 하나로 볼 수 있다. 로더는 로드되고 있는 미정의된 참조를 정의하기 위해 심볼 테이블을 사용한다. 또한 타겟 시스템에 로드된 모듈을 언로드시킬 수 있어야 한다. 하나의 모듈이 언로드될 때 관련된 모든 내용이 심볼 테이블에서 제거되어야 한다. 실행 모듈은 운영 체제에 따라 다양할 수 있다. 타겟 매니저는 COFF와 ELF 형태의 실행 파일을 로드할 수 있도록 한다. 또한, 로더는 각 도구가 동작 중일 때 실행 모듈 로드를 하기 위해 메모리를 요구하게 되는데, Host-Pool Memory Manager는 타겟 매니저는 호스트 상의 메모리 할당에 대한 요구를 해결하기 위해 타겟 메모리 풀을 관리하게 된다. 타겟 메모리 매니저는 비어 있는 블록 리스트나 블록 헤더와 같은 정보를 호스트 상에 유지하고 있다가 타겟 메모리 할당에 최소한의 시간을 사용하게 한다. 관리될 타겟 메모리 크기는 타겟을 시작할 때 결정되고, 동적으로 증가하게 된다. 타겟 메모리의 일정 영역은 캐싱하기에 좋은 후보가 될 수 있다. 당연히 타겟 메모리 캐쉬는 모든 타겟 상주 모듈의 프로그램 텍스트 부분이 된다. 그 부분은 수정하는 것이 예외 사항 추가 등을 제외하면 매우 이례적인 일이기 때문에 그렇게 하는데, 이는 크로스 디버깅의 성능 향상에 고무적인 역할을 한다.

- QDI(Qplus Debugging Interface) 백엔드(back -end)

이 모듈은 도구들이 타겟과 연결 방식에 독립적이도록 해준다. Back-End는 통신 방식마다 하나씩, 즉, 이더넷 연결인지, 시리얼 연결인지에 따라 해당 부분이 독립적으로 구현된 모듈이며 상위 모듈인 Back-End Manager의 제어를 받는다. 타겟 매니저 Back-End는 디버그 에이전트가 제공하는 QDI 프로토콜에 맞게 요구를 하게 된다. QDI 프로토콜은 디버그 에이전트가 제공하는 모듈로서, 타겟 매니저가 타겟에 접근하는 용도에 따라 적절한 서비스 함수를 호출하는 형태로 사용되어진다. 즉, 디버그 에이전트가 서버로서, 타겟 매니저가 클라이언트로서 서로의 역할을 수행하는데 매개체가 되는 인터페이스라고 볼 수 있다. QDI 프로토콜에 대한

세부기능에 대해서는 다음 절에서 설명한다.

4.5 타겟 디버깅 : 디버그 에이전트

디버그 에이전트는 호스트 상의 도구들이 타겟 매니저를 통하여 디버깅 기능을 수행할 수 있도록 타겟상에서 대리자로서의 역할을 한다. Qplus-T RTOS에서 디버그 에이전트는 태스크 모드로 동작한다. 여기서 태스크 모드란 타겟 보드상에 실시간 OS가 적재되고 디버그 에이전트는 적재된 RTOS의 하나의 태스크로 동작하면서 사용자의 응용 프로그램을 디버깅할 수 있는 모드이다. 디버그 에이전트는 타겟상에 상주하면서 동작하므로 타겟 시스템에 최소한의 부하를 주도록 구성한다.

타겟 매니저와 디버그 에이전트는 SUN의 RPC (Remote Procedure Call) 메카니즘을 이용하여 통신하고, 통신시 자료 교환은 XDR(eXternal Data Representation)을 이용한다[15]. RPC를 지원하기 위하여, RPC 송수신 기능, 등등을 갖는 소스코드 사이즈 9 KB짜리, 오브젝트 모듈 크기 3 KB짜리 RPC 라이브러리 파일을 만들었으며, 데이타 송수신을 위해, 모듈 모두를 합쳐서 소스코드 사이즈가 56 KB가 되고, 오브젝트 모듈로는 20 KB가 되는 XDR 라이브러리 파일을 만들었다. 본 논문에서는 이들의 구현 기법에 대하여 설명하는 것이 아니기 때문에 이들의 구체적인 구현 기법에 대해서는 생략하기로 하겠다. 대신에 이들을 사용하여 호스트와 타겟간의 통신을 하는 기법 즉, QDI 프로토콜에 대하여 설명한다.

그림 12는 RPC를 이용한 기본 통신 메커니즘이다. 디버그 에이전트의 초기화 과정에서 디버그 에이전트 인터페이스 함수와 서비스 루틴 함수들의 포인터 초기화가 이루어지며, 이들에 대한 정보를 갖고 있는 테이블을 생성한다. 이 테이블은 나중에 타겟 매니저에서 RPC 호출이 있을 경우 해당 서비스에 대한 프로시저를 호출하기 위해 사용되며 구현된 서비스 루틴 테이블은 다음 표 3과 같다. 여기서 프로시저 번호는 디버그 에이전트에서 구현한 통신 모듈과 메인 모듈사이의 QDI 프로토

표 3 디버그 에이전트 서비스 루틴 테이블

번호	호출 함수	파라미터에 대한 XDR 필터	반환값에 대한 XDR 필터
0	ta_target_ping	xdr_void	xdr_void
1	ta_target_connec	xdr_void	xdr_TA_TGT_INFO
2	ta_target_disconnec	xdr_void	xdr_void
3	ta_target_mode_set	xdr_u_int	xdr_void
4	ta_target_mode_get	xdr_void	xdr_u_int
10	ta_mem_read	xdr_TA_MEM_REGION	xdr_TA_MEM_XFER
11	ta_mem_write	xdr_TA_MEM_XFER	xdr_void
12	ta_mem_fill	xdr_TA_MEM_REGION	xdr_void
13	ta_mem_move	xdr_TA_MEM_REGION	xdr_void
14	ta_mem_checksum	xdr_TA_MEM_REGION	xdr_UINT32
15	ta_mem_protect	xdr_TA_MEM_REGION	xdr_void
16	ta_mem_txt_update	xdr_TA_MEM_REGION	xdr_void
17	ta_mem_scan	xdr_TA_MEM_SCAN_DESC	xdr_TGT_ADDR_T
30	ta_ctx_create	xdr_TA_CTX_CREATE_DESC	xdr_UINT32
31	ta_ctx_kill	xdr_TA_CTX	xdr_void
32	ta_ctx_suspend	xdr_TA_CTX	xdr_void
33	ta_ctx_resume	xdr_TA_CTX	xdr_void
40	ta_reg_get	xdr_TA_REG_READ_DESC	xdr_TA_MEM_XFER
41	ta_reg_set	xdr_TA_REG_WRITE_DESC	xdr_void
60	ta_evt_pt_add	xdr_TA_EVTPT_ADD_DESC	xdr_UINT32
61	ta_evt_pt_delete	xdr_TA_EVTPT_DEL_DESC	xdr_void
70	ta_event_get	xdr_void	xdr_TA_EVT_DATA
80	ta_cont	xdr_TA_CTX	xdr_void
81	ta_step	xdr_TA_CTX_STEP_DESC	xdr_void
90	ta_func_call	xdr_TA_CREATE_DESC	xdr_UINT32
91	ta_evaluate_gopher	xdr_TA_STRING_T	xdr_TA_MEM_XFER
92	ta_direct_call	xdr_TA_CTX_CREATE_DESC	xdr_UINT32

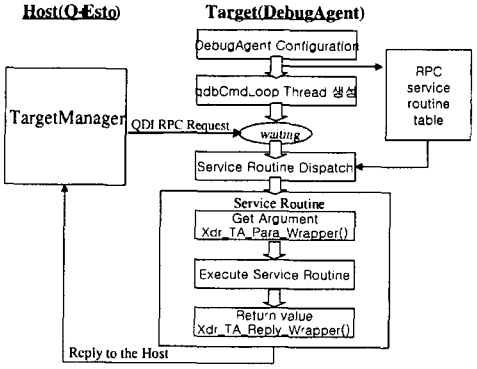

그림 12 디버그 에이전트의 동작

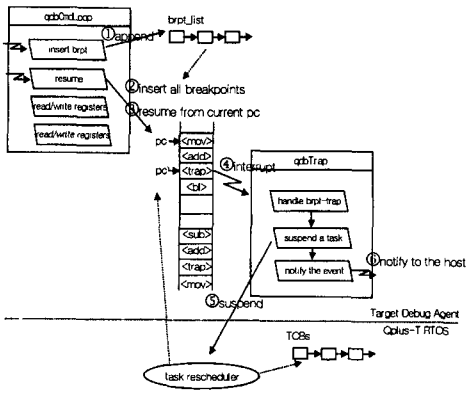

그림 13 타겟에서의 브레이크 설정 및 재실행

콜이 사용되며, 디버그 에이전트는 이 번호를 이용하여 Q+Esto가 RPC 호출로 요구한 프로시저를 구분한다. 호출 함수는 실제 서비스되는 프로시저를 나타내며 해당 프로시저의 시작 주소가 저장되어 있다. 파라미터 필터는 Q+Esto로부터 요구된 서비스 함수의 파라미터가 XDR 형식이기 때문에 이를 해석하기 위해 사용된다. 마지막의 반환값 XDR 필터는 프로시저 실행 후의 반환값을 Q+Esto로 되돌려 주기 위해서 XDR 형태로 변환해 주는 역할을 한다.

디버그 에이전트는 타겟 매니저의 백엔드 모듈로부터 전달 받은 서비스 요청을 처리한다. 그림 13은 타겟 매니저로부터의 QDI_BP 및 QDI_CTX_RESUME 이벤트에 대한 디버그 에이전트의 대표적인 동작 메카니즘을 표현한 것이다. 먼저, QDI_BP 이벤트 요청에 의해 디버그 에이전트는 qgdCmdLoop 모듈에서 브레이크 리스트에 QDI_BP 이벤트에 대한 ① 브레이크 이벤트 정보를 추가하고, 그 다음 QDI_CTX_RESUME 서비스 요청에 의해 ② 브레이크 포인트 리스트 정보를 이용하여 트랩코드를 실행이미지에 삽입한 다음 현재위치 즉, 현재 PC 값으로부터 ③ 재실행을 한다. 실행 중 트랩코드를 만나게 되면 ④ s/w 트랩 인터럽트가 발생하게 되고 디버그 에이전트의 qdbTrap 모듈은 그 인터럽트에 반응하여 트랩이 발생한 PC 값을 토대로 하여 해당 태스크를 ⑤ suspend 시키고 ⑥ QDI_EVENT_NOTIFY 이벤트를 발생시킨다.

타겟 매니저로부터의 요청은 타겟 매니저와 연동하는 호스트상의 도구들에 의한 것인데 이것들에는 다음과 같은 것으로 구성되어 있다.

- 호스트 상주 도구(원격 소스 수준 디버거, 대화형 쉘, 원격 자원 모니터 등)로부터의 요청(디버깅을 위한 브레이크포인트, 쉘 명령어 처리, 이벤트 감시 등)
- 타겟 시스템상의 컨텍스트(context) 생성, 소멸, 재시작을 위한 것
- 타겟 시스템상의 이벤트 알림기능
- 타겟 메모리에 적재된 오브젝트 모듈의 실행 및 종료 인지
- 타겟에 적재된 오브젝트 코드내의 함수 호출
- 호스트상의 도구들이 요청하는 타겟의 메모리, 레지스터 상태 정보
- 타겟 시스템에 콘솔이 없을 경우 그 출력값을 호스트상의 터미날로 출력할 수 있게 가상 IO로의 출력에 관한 요청

5. 적용실험

5.1 실험예제 소개

본 논문에서는 내장형 시스템 개발환경에서 비정지 실시간 디버깅을 위한 원격 멀티태스크 디버거의 유용성을 검증하기 위해 다음 그림 14와 같은 간단하지만 내장형 시스템에서의 가장 대표적인 예제를 적용 실험 대상으로 선택하였다. 이 그림에서 보듯이 producer 태스크는 *msgq_send()*라는 함수를 이용하여 하나의 메시지큐 즉, msgq에 sdata 자료를 보내게 되고, 이것을 consumer 태스크는 msgq_receive() 함수를 통해 적절한 시점에 rdata 자료를 가져온다.

실험 검증은 본 논문에서 제안한 디버깅 환경을 이용하여, 태스크들을 선택하여, 이들에 의해 다중으로 태스

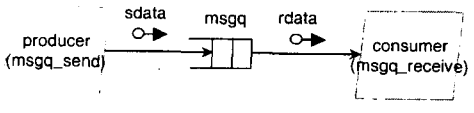

그림 14 producer-consumer 프로그램

크 윈도우들이 생성되는 과정과 이들을 통해 일목요연하게 디버깅 리소스들 예를 들면, 세마포어 혹은 메시지 큐, 등등의 내용을 본다든지 혹은 멀티 태스크 윈도우별로 정지점을 설정하여 한단계 씩 소스레벨에서 디버깅 가능한지, 그리고, 디스어셈블 코드와 함께 디버깅이 가능한지를 확인하는 것으로 하겠다.

5.2 Q+Esto 디버거 실행

멀티 태스크 디버깅은 호스트 시스템에서 타겟 시스템용으로 교차 컴파일된 실행파일을 타겟보드로 전송한 후 호스트에서 타겟에 적재된 멀티 태스크 실행 파일들을 제어하면서 디버깅을 수행하는 것이다. 이때, 호스트에 있는 디버거는 타겟과의 통신 트랙픽을 최소화하기 위해 디버깅 정보가 포함된 실행파일과 이것의 소스 프로그램을 필요로 한다. 원격 디버거는 이 실행파일에 포함된 디버깅 정보를 이용하여 내부 심볼 테이블을 구축하고 소스프로그램의 열람, 변수의 모든 출력 등 타겟 시스템에서 직접적인 수행을 필요로 하지 않는 기능을 목적 시스템과의 통신없이 사용자에 알려준다.

멀티 태스크 디버깅을 위해 먼저, 개발자는 Qplus T RTOS를 Q+Esto IDE의 부트로더를 통해 부팅 후 실행시킨다. 이로써, 타겟에서는 디버그 에이전트가 Qplus -T RTOS의 실행과 동시에 수행이 되고 디버깅 서버로써 호스트로부터의 어떤 메시지를 기다리는 상태가 된다. 그 다음, 호스트에서는 디버깅 미들웨어인 그림 15와 같이 타겟 매니저를 수행시키게 되면 디버그 에이전트와 타겟 매니저가 QDI 프로토콜을 사용하여 디버깅을 위한 초기 세션을 형성시킨다. 타겟 매니저에는 디버깅 미들웨어의 이름, 타겟 IP 어드레스, 그리고 Qplus T RTOS 이미지 파일등에 관련 정보가 등록되어 있다.

일단, 타겟과 호스트간의 디버깅 세션이 형성이 되면, Q+Esto 디버거를 실행시킬 수 있게 된다. 다음 그림 16은 Q+Esto 디버거의 사용자 인터페이스를 보여 주고있다. 이 그림에서 보듯이 Q+Esto 디버거는 메인 윈도우 하나와 다수의 멀티 태스크 윈도우들로 구성되며, 메인 및 태스크 윈도우들에서 공유하는 공유 리소스뷰(resource view)들로 구성되어 있다. 공유 리소스뷰에는 기본적으로 레지스터 및 메모리 정보들, 그리고 메시지 큐(message queue), 세마포어(semaphore), 뮤텍스(mutex) 등등의 공유 리소스 정보도 함께 보여주는 윈도우들로 구성되어 있다.

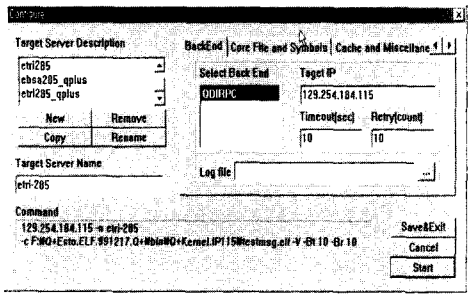

그림 15 타겟 매니저의 실행

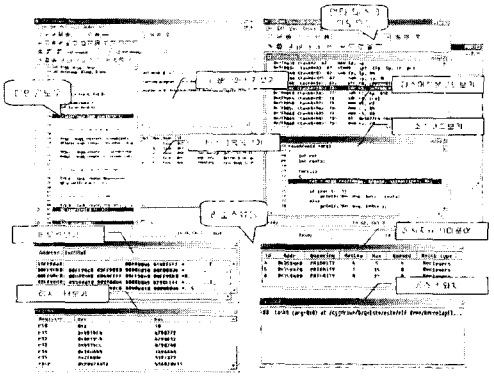

그림 16 멀티태스크 디버거 사용자인터페이스

5.3 멀티컨텍스 디버깅: 생산자-소비자 태스크 윈도우 생성

멀티 태스크 프로그램의 디버깅에서 태스크의 디버깅은 현재 실행 중인 태스크에 대해서만 가능하다. 먼저, 태스크 목록 윈도우로부터 현재 실행 중인 태스크들의 상태를 파악한 후 새로운 태스크 윈도우들 생성하는 작업을 수행한다. 생산자-소비자 태스크 윈도우를 생성하는 방법은 다음 그림 17에서처럼 간단히 태스크 목록

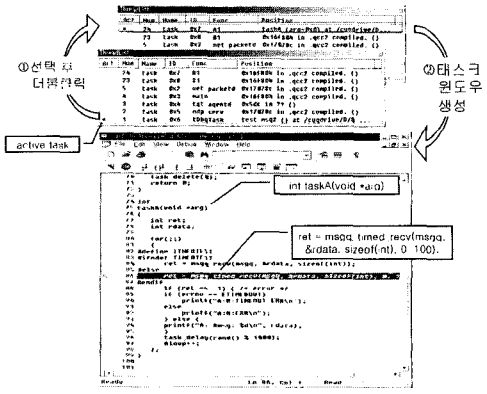

그림 17 태스크 윈도우 생성

윈도우로부터 해당 태스크를 ①선택한 후 더블클릭하여, ②태스크 윈도우를 생성하면 된다. 이 그림은 소비자 태스크에 해당하는 taskA 태스크 윈도우를 실행시키는 과정을 보여주고 있다. 이 프로그램은 무한 루프 안에서 100ms의 msgq_timed_recv 함수를 사용하는 타임아웃을 가진 메시지 수신 알고리즘으로 되어 있다. 생산자 태스크 윈도우도 마찬가지 방법으로 실행시킨다.

5.4 공유자원 디버깅 : 메시지 큐 목록 보기 윈도우 실행 및 메시지 수신처리 기능 디버깅

본 논문에서는 기존의 멀티 태스크 디버거들과는 달리, 공유자원에 대한 디버깅 윈도우들을 따로 제공하고 있다. 본 절에서는 그 한 예로, 메시지 큐 공유자원에서의 디버깅에 대해 설명하기로 한다. 일단, 메시지 큐 수신처리 기능이 원하는 형태로 이루어지는지를 디버깅하기 위해 메시지 큐 목록 윈도우를 통해 메시지 큐 리소스의 실행 상태를 점검한다. 일단, 메시지 수신 태스크에 브레이크 포인트 설정하여 디버깅하기 위해 메시지 수신처리부분에서 태스크의 실행을 잠시 멈춘다. 그 다음, 현재 정지된 위치에서 한단계 스텝처리를 하여 메시지 큐로부터 원하는 정보를 가져오는지를 점검한다. 그림 18과 같이, 본 실험에서는 스텝처리 후 결과로써 rdata값을 295로 가져왔으며 메시지 큐 목록 값의 첫번째 필드의 값 즉, "1 0xff98c0 4 28 01 00 00 ...." 으로 볼 때 첫번째 메시지는 메시지 큐의 0xff98c0 어드레스에 4 byte 사이즈로 들어 있으며 메모리 내용은 hexa 형태로 낮은 주소로부터 높은 주소로 "28 01 00 00"로 되어있다. 따라서, 첫번째 필드값은 0x00000128 이므로 정수값으로 환산하면 296이다. 이것으로써, 생산자-소비자 프로그램의 알고리즘에 따라 메시지 수신 프로그램이 메시지 큐로부터 정확히 데이타를 가져가고 있음을 확인할 수 있다.

5.5 소스윈도우 및 디스어셈블 연동 디버깅

내장형 시스템을 디버깅을 할 때 필요에 따라서는 어

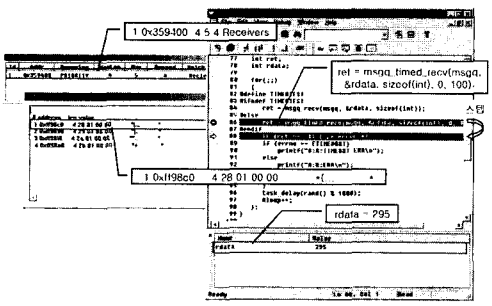

그림 18 메시지 큐 디버깅 - 메시지수신처리

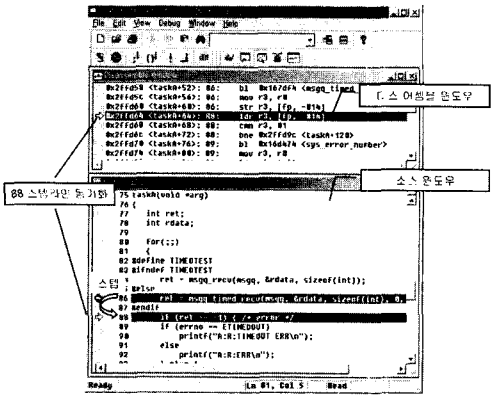

그림 19 소스 및 디스어셈블 디버깅

셈블리 언어 수준의 디버깅도 필요한 경우가 많다. 이러한 상황을 지원하기 위해, 그림 19와 같이 디스 어셈블 윈도우와 소스 코드 윈도우와의 디버깅 연동기능을 제공하고 있다.

6. 관련 연구들 및 비교평가

본 장에서는 본 연구와 관련된 관련 연구들 및 비교평가에 대하여 논하기로 한다. 본 논문에서는 내장형 시스템 특히, Qplus-T RTOS를 탑재한 내장형 시스템에서의 원격 디버깅 환경 구현 기법을 제시하였다. 이것은 디버깅을 위해 특별한 H/W를 사용하는 방식 예를 들면, ICE(In Circuit Emulator)와 같은 H/W 장비를 이용하여 디버깅하는 방식과 달리, S/W 디버깅 에이전트를 통하여 타겟의 커널 및 응용 프로그램을 디버깅하는 다른 방식을 사용하고 있다. 그래서, 본 논문에서 제시하는 디버거와의 차이는 특별한 디버깅 H/W에서 제공되는 여러 가지 장점 혹은 단점, 예를 들면, 독립적인 H/W를 사용함으로써 디버깅 간섭 시간이 오버헤드로 작용하지 않는다든지, 타겟의 커널이 멈추게 되어도 그 상태를 볼 수 있다든지 등의 장점은 있으나 대체적으로 디버깅 도구의 가격은 매우 비싸다든지 하는 등등의 장/단점에서 오는 것이기 때문에 H/W 기반의 방식의 디버깅 도구들과의 비교는 생략하기로 한다. 그리고, 이 장에서는 주로 S/W 기반의 디버깅 기능을 가진 도구들과의 비교를 설명하기로 하겠다.

- Thread-Aware Debugger with an Open Interface[16]

본 논문에서 제시하는 도구와 마찬가지로 GNU GDB를 수정하여 개방형 인터페이스를 제공함으로써 확장성, 이식성, 그리고 유연성이 좋은 멀티 쓰레드 디버거를 구

현하고 있다. 또한, 본 논문에서 제시하는 도구와 마찬가지로 쓰레드 인식 디버거를 만들기 위해 특정 쓰레드에 브레이크 포인트를 설정하는 기능과 스케줄러를 필요에 따라 멈추게 하거나 재가동시키는 기능 그리고, 쓰레드들의 상태를 볼 수 있는 명령어들을 제공하고 있다. 그러나, 이 디버거에서 제공하는 기능은 첫번째 호스트-타겟으로 이루어진 원격개발환경을 위한 멀티 쓰레드 디버거의 구현문제가 아니며, 개방형 인터페이스의 제공도 공용 질의어 해석기를 통해 디버거와 멀티쓰레드 프로그램과의 디버깅 인터페이스를 취하는 방식으로 되어 있어 본 논문에서의 QTI 및 QDI 개방형 C 인터페이스 방식과는 다르다. 본 논문에서는 C 함수 수준의 프리미티브한 인터페이스 기능들을 제공함으로써 원격 디버깅을 구현한다. 또한, RPC를 활용함으로써 원격 디버거의 구현을 용이하게 할 수 있다.

- KDB: A Multi-threaded Debugger for Multi-threaded Applications[17]

이 논문에서 제공하는 디버거도 원격개발환경에서 멀티 쓰레드 응용 프로그램을 위한 멀티 쓰레드 디버거를 제공한다. 이 디버거는 하나의 디자인 모델로 사용자 및 커널 쓰레드의 디버깅을 지원하는 구조로 되어 있다. 각각의 커널 쓰레드들은 타겟에 있는 지역디버거들에 의해 *ptrace* 및 */proc* 파일 시스템을 이용하여 제어받게 되어 있으며, 메인 디버거는 사용자에 의해 구동이 되며 모든 지역 디버거들의 상태를 쓰레드 목록 정보를 통해 통제한다. 이 디버거는 Unix OS에서 소켓(socket)을 통해 비동기적(asynchronous) 멀티 컨텍스트 디버깅 기능을 제공한다. 그러나, 본 논문에서 제시하는 디버거와는 다르게 확장성, 유연성, 이식성을 위한 개방형 인터페이스 구조를 제공하고 있지 못하고 있다.

- GNU gdb-5.x 멀티 쓰레드 디버깅[11]

GNU GDB-5.x 버젼에서는 멀티 쓰레드 디버깅 기능을 제공하나 원격용 멀티 쓰레드 디버거는 아직 미공개되어 있는 상태이다. 현재, gdb의 경우 stand-alone 형으로 멀티 쓰레드 디버깅 기능을 제공하고 있으나 아직 오류가 많은 상태이며, 본 논문에서 제시하는 방식의 개방형 인터페이스는 제공되지 못하고 있다. 그리고, 본 논문에서 제시하는 방식의 비동기적 멀티 컨텍스트 디버깅 기능이 아니라 동기적(synchronous) 형태의 멀티 쓰레드 디버깅 기능 즉, 특정 쓰레드에 브레이크가 걸릴 경우 다른 모든 쓰레드들도 정지하게 되고, 그 쓰레드에서 재실행이 되는 경우 모든 쓰레드가 다시 구동하게 되는 방식을 제공하고 있다. 본 논문에서는 GNU GDB 5.x 버젼의 소프트웨어 아키텍처를 손상하지 않는 상태에서 확장성, 유연성, 이식성이 좋은 개방형 원격 디버깅 인터페이스를 추가함으로써 비동기적 멀티 컨텍스트 디버깅이 가능한 원격 멀티 태스크 디버거를 구현하였다.

- VxWorks-Tornado[18,19]와 VRTX-Spectra[12]

VxWorks의 Tornado에서 제공하는 디버거는 본 논문의 도구와 마찬가지로 VxWorks와 같은 특정 운영체제를 위한 개방형 인터페이스를 제공함으로써 디버깅 기능의 확장이 용이하게 하고 있으며, 비동기적 멀티 컨텍스트 디버깅 기능을 제공하고 있다. 그리고, 쓰레드 인식 디버거를 만들기 위해 개방형 인터페이스를 통해 타겟 쓰레드들의 상태를 본다든지 스케줄링을 잠시 멈추게 한다든지 혹은 재실행 시킨다든지 하는 기능들을 제공하고 있다. 그러나, 문제는 하나의 디버거에서 하나의 쓰레드를 선택(attach)하여 디버깅하고 그 쓰레드에서 생성된 다른 쓰레드를 디버깅하기 위해서는 또 다른 디버거를 실행시켜야 되기 때문에 쓰레드들의 상태를 일괄성 있게 살피는 것이 힘들다. 그리고, 쓰레드 그룹을 보기 위해서는 디버거가 아닌 원격 쉘과 같은 다른 도구를 활용해야 한다. 또한, 토네이도의 경우 상업용 도구로 제공되고 있어 유지보수 비용이 비싸다. 한편, VRTX의 Spectra에서 제공하는 디버거의 경우 VxWorks의 Tornado에 비해 쓰레드 인식 디버깅 기능을 제공하나 개방형 인터페이스를 제공하고 있지 않아 확장성, 유연성, 이식성에는 문제가 있으며, VxWorks의 Tornado에 비해 쓰레드 디버깅을 위한 기능이 잘 갖추어져 있는 편이다. 그러나, X-Window로 작성이 되어 있으며 아직 오류가 많아 사용하기 불편하다. 본 논문에서 제시하는 디버거는 VC++의 MFC로 작성되어 있으며 윈도우 개발환경에서 사용할 수 있게 되어 있어 사용자에게 더욱 친숙하다. 현재, 윈도우 운영체제와 LINUX 운영체제에서 상호 호환이 되도록 Qt를 사용하여 확장하고 있는 중이다.

- kgdb[20]를 포함하는 DDD[21] 디버거

현재, kgdb는 멀티 태스크들을 기반으로 하는 VxWorks, VRTX 및 Qplus-T RTOS 방식과 다른 프로세스 기반의 리눅스 커널을 디버깅하는 기능으로 구현되어 있으며, 지원 CPU도 X86으로 한정되어 있다. 그리고, 본 연구소를 포함한 몇몇 업체들에서 ARM CPU도 지원하기 위해 ARM용으로 포팅한 수준에 불과하다. 또한, 디버깅 기법도 상용으로 사용하기에는 아직 문제가 있어 보이는 안정화되지 못한 도구이다. 그리고, 확장성과 이식성을 위해, kgdb와 같은 디버깅 엔진과 GUI 사이의 연결을 DDD에서 사용한 방식 즉, gdb 내부 모듈에 GUI연결하기 위한 특별한 함수들을 구현하

는 방식과 다르게 파이프 프로세스를 통한 GUI 모듈과 gdb엔진간의 연동하는 기법을 사용하였다. 앞서 4.2절에서 설명하였듯이, 이 방식을 채택하게 된 이유는 GUI는 Visual C++로 개발되었고, 디버깅 엔진은 GNU gcc로 개발되었기 때문이다. 즉, 두 개의 다른 컴파일러로 개발된 모듈을 하나로 합치기 위한 기법으로 구현되었다. DDD는 GNU에서 제공하는 그래픽 라이브러를 사용하고, 모두 gcc 컴파일러를 사용하여 컴파일해야 한다.

7. 결론 및 향후연구방향

20세기 후반부터 시작된 정보화 혁명이 산업체와 사무실을 대상으로 진행되어 왔다면, 21세기 정보화 혁명은 가정을 중심으로 이루어질 것으로 예상된다. 현재, 각각의 정보가전 제품에는 기반 소프트웨어로 RTOS (Real-Time Operating System)를 내장시킬 예정이며, 정보가전용 RTOS의 성능 및 가격에 따라 정보가전 제품의 경쟁력도 비례할 것으로 예측되고 있다.본 논문에서는 이러한 인터넷 정보가전 시스템과 같은 Qplus-T 내장형 시스템을 위한 멀티 태스크 디버깅 환경의 구현 기술에 대하여 제안하였다. 이의 세부사항으로 Q+Esto 원격 개발 환경에 대하여 기술하였으며, 또한 호스트에서의 원격 멀티태스크 디버거 구현기술 및 디버깅 미들웨어인 타겟 매니저 구현기술에 대하여 기술하였고, 원격지 타겟에서의 디버그 에이전트 구현기술에 대하여 기술하였다.

본 디버거는 GNU GDB를 기반으로 구현되어 있으며, 정보가전용 내장형 시스템과 같이 프로그램 크기는 작지만 사소한 실수에도 응용 프로그램 자체가 제기능 다하지 못하는 시스템 혹은 단일 CPU내에서 멀티 태스크들이 병행적으로 수행되어야 하는 시스템을 디버깅하기위해 만들어졌다. 그리고, 제공되는 GUI로는 MS Windows의 MFC를 사용하고 있어 디버거를 직관적으로 사용하기 편리하게 하였다. 또한, Qplus-T RTOS를 위한 개방형 인터페이스 함수(Open C Interface)를 제공함으로써 디버깅 기능의 확장이 용이하게 하고 있으며, 디버깅 미들웨어인 타겟 매니저와 디버그 에이전트를 기반으로 신뢰성 있는 디버깅 커뮤니케이션을 확보해줌으로써 멀티 태스크들의 비동기적 멀티 컨텍스트 디버깅 기능을 오류없이 수행할 수 있게 하고 있다. 그리고, 타겟에서는 멀티 태스크 인식 디버거를 만들기 위해 개방형 인터페이스를 통해 타겟 쓰레드들의 상태를 본다든지 스케줄링을 잠시 멈추게 한다든지 혹은 재실행 시킨다든지 하는 기능들을 제공하게 하였다.

현재는 Strong-ARM110과 1110 CPU를 갖는 타겟시스템에서 활용가능하나, 앞으로는 가능한 많이 활용되는 CPU에 대해서 지원을 확대할 예정이며, 쓰레드 기반 운영체제인 Qplus-T뿐만이 아니라, 프로세스 기반의 Qplus-P RTOS에서도 연동하여 수행될 수 있는 디버깅 환경으로 발전시킬 예정이다. 그리고, 본 논문에서 제안하고 있는 디버거는 GNU GDB 4.18 버전을 기반으로 만들어져 있으나, GNU GDB의 업그레이드에 맞추어 (현재는 GDB 5.2가 가장 최신 버전임) 디버깅 엔진도 업그레이드할 예정이다. Qplus-P용 GDB엔진은 현재 GDB 5.1.1버전으로 업그레이드 작업중에 있다.

2001년도의 1단계 과제에서는 Qplus-P RTOS를 위한 Q+Esto 개발환경의 1차버젼을 개발 완료하였으며, 2002년 현재에는 트레이스포인트를 이용한 비정지 실시간 디버깅 기능의 추가 및 정보가전 단말에서 필요한 저전력 측정 기능의 추가 등등의 다양한 특성을 가진 디버거로의 확장도 고려하고 있는 중이다.

그리고, 본 논문에서 제안하는 Qplus-T용 Q+Esto 멀티 태스크 디버깅 환경은 현재, 공개소프트웨어로 제공되고 있어 누구나 활용가능하다[22]. 단, 현재는 SA-110 혹은 SA-1110 CPU를 채택한 개발보드에서 활용가능하다.

참 고 문 헌

[1] 김선자, 김홍남, 김채규, "인터넷 정보가전용 RTOS 기술 현황", 한국정보과학회지, 제19권, 제4호, pp.57-64, 한국정보과학회, 2001.

[2] 김선자, 김홍남, 김채규, "정보가전용 임베디드 운영체제 기술", 한국통신학회지, 제18권 제12호, pp.72-81, 한국통신학회, 2001.

[3] 김홍남, "사용자개발도구연구", 정보가전용 실시간 OS 컨퍼런스(RTOS 99) 자료집, ETRI, pp.178-196, Nov. 17, 1999.

[4] 임채덕, "Q+ 사용자개발도구 기술연구", 정보가전용 실시간 OS 컨퍼런스(RTOS 2000) 자료집, ETRI, pp.107-125, Nov. 3, 2000.

[5] Eldad Maniv, "New Trends in Real Time Software Debugging," Real-Time Magazine 99-2 (http://www.realtime info.com), pp.23 25, 1999.

[6] T. Yasuda, K. Ueki, "A Debugging Technique Using Event History," *Proceedings of the Conference on Real-Time Computing Systems and Applications*, pp.137-141, 1994.

[7] 이광용, 김창갑, 김홍남, "정보가전용 내장형 소프트웨어 개발을 위한 원격 디버거의 설계 및 구현", 한국정보처리학회지, 2000년 춘계 학술발표논문집, 제7권, 제1호, 한국정보처리학회, 2000.

[8] Kwangyong Lee, Chaedeok Lim, Kisok Kong, Heung Nam Kim, "A Design and Implementation

of a Remote Debugging Environment for Embedded Internet Software," Lecture Notes in Computer Science vol. 1985, Springer Verlag, pp.199 203, 2001.
[9] Hideyuki Tokuda and Makoto Kotera, "A Real Time Tool Set for the ARTS Kernel," *Proceedings of Real Time Systems Symposium*, 1988.
[10] Intel, *StrongARM EBSA 285 Evaluation Board*, 1998.
[11] GNU, GDB: The GNU Project Debugger, http://sources. redhat.com/gdb
[12] Microtec, *Spectra Boot and VRTX Real Time OS*, 1996.
[13] Jonathan B. Rosenberg, *How Debuggers Work*, John Wiley & Sons, 1996.
[14] Michael Snyder and Jim Blandy, The Heisenberg Debugging Technology, http://sources.redhat.com/gdb/ talks/esc west 1999/INTROSPECT.html
[15] W. Richard Stevens, Unix Network Programming: Interprocess Communications, Prentice Hall, 1999.
[16] Daniel Schulz and Frank Mueller, A Thread Aware Debugger with an Open Interface, http://citeseer.nj.nec.com/schulz00threadaware.html, 1998.
[17] Peter A. Buhr, Martin Karsten and Jun Shih, KDB: A Multi-threaded Debugger for Mutlti- threaded Applications, http://citeseer.nj.nec.com/buhr96kdb.html, 1996.
[18] WindRiver, *Tornado Users Guide*, 1995.
[19] WindRiver, *Tornado API Guide 1.0.1*, 1997.
[20] Amit S. Kale, "kgdb: linux kernel source level debugger," http://kgdb.sourceforge.net, 2000
[21] GNU, "DDD: Data Display Debugger," http://www.gnu.org/software/ddd
[22] ETRI & Dasan, "확장가능 조립형 실시간 OS", http:// embedix.com/ qplus

이 광 용
정보과학회논문지 : 컴퓨팅의 실제
제 9 권 제 2 호 참조

김 홍 남
정보과학회논문지 : 컴퓨팅의 실제
제 9 권 제 2 호 참조