

임베디드 시스템 개발을 위한 타겟 에이전트

Target Agent for Embedded System Development

김 행 곤(대구가톨릭대학교 컴퓨터정보통신공학부)

정 란(삼척대학교 컴퓨터공학과)

요 약: 최근 내장형 시스템의 운영체제로 임베디드 리눅스가 점차 많이 사용되고 있다. 이에 따라 더욱 복잡한 내장형 소프트웨어를 빨리 개발하여 적시에 상품화하는데 필수적인 임베디드 리눅스용 원격 통합개발환경에 대한 요구가 증가하고 있다. 그런데, VxWorks나 QNX와 같은 내장형 운영체제의 개발 환경에 비해서 임베디드 리눅스용 개발 환경은 편리성과 기능성이 미흡하다. 따라서 본 논문에서 임베디드 시스템 개발환경 EIDE(Embedded Integrated Development Environments)를 제시한다. EIDE는 크로스 툴체인, 디버거, 모니터 등 다양한 도구들로 구성되며 모두 GUI에 기반한다.

또한 EIDE의 다양한 도구들이 타겟을 접근하는데 필요한 기능을 제공하는 타겟 에이전트를 제안한다. 타겟 에이전트는 호스트/타겟 환경이라 할지라도 개발자가 GUI로 된 EIDE의 통합개발환경 안에서 다양한 도구들을 이용하여 원격 개발에 필요한 모든 것을 편리하게 수행할 수 있도록 지원하면서 여러 도구의 요청을 동시에 처리할 수 있는 멀티쓰래드 구조의 타겟 에이전트를 제안하고, 타겟 에이전트가 EIDE에서의 원격 개발을 어떻게 더욱 편리하게 하는지 설명한다.

I. 서론

내장형 시스템 응용분야는 기존의 산업이나 군사 분야를 위한 제어 및 관제 장치에서 경보가전이나 지능형 전물 등을 위한 다양한 장치로 빠르게 범위가 넓어지고 있다. 특히 유비쿼터스 컴퓨팅 시대가 다가옴에 따라 내장형 시스템은 우리의 실생활과 더욱 밀접하고 가깝게 다양한 분야에 사용될 것이다.

이에 따라 복잡하고 다양한 기능의 처리가 필요한 내장형 응용 소프트웨어에 대한 개발 요구와 내장형 시스템을 빨리 개발하여 적시에 상품화하고자 하는 요구가 빠르게 증가되고 있다. 이를 위해서는 사용이 편리하고 기능이 풍부한 내장형 소프트웨어용 개발 환경이 필수적이다[1,2,3].

최근 내장형 시스템의 운영체제로 임베디드 리눅스가 많이 사용되고 있다. 이에 따라 빠르고 편리하게 임베디드 리눅스 응용을 개발할 수 있게 해주는 개발환경의 중요성이 대두되

본 연구는 2004년 한국 산학협동재단 산학협동 과제지원 사업에 의해 연구됨
고 있다. 블랜드의 Kylix처럼 리눅스용 네이티브 개발환경에는 마이크로소프트의 비주얼 C++ 만큼 편리한 통합개발환경이 존재한다. 그러나, 호스트와 타겟으로 구성되는 원격 환경에서 편리하게 응용을 개발할 수 있도록 해주는 임베디드 리눅스용 원격 통합개발환경은 거의 없다. 반면, VxWorks나 QNX와 같은 전통적인 내장형 운영체제를 위한 통합개발환경은 원격 환경에 필요한 다양한 기능을 제공하고 있다[4].

리눅스에서는 개발한 프로그램을 원격 디버깅할 경우 프로그램을 타겟으로 전송한 후

gdbserver를 실행시킨다. 호스트에서는 gdb(또는 GUI 기반인 Insight, DDD 등)를 실행시키고 gdbserver에 접속한 후 디버깅을 시작한다. 프로그램을 디버깅할 때마다 이처럼 타겟 창과 호스트 창을 오가며 작업해야 한다. 타겟의 프로세스나 메모리 자원 등을 모니터링하기 위해서는 타겟에 접속하여 ps나 free 명령을 실행한다.

이와 같은 복잡한 절차는 리눅스에서 개발 경험이 많은 개발자라 할지라도 앞으로 더욱 짊어지는 내장형 시스템 개발 기간을 맞추면서도 신뢰성 있는 내장형 시스템을 개발하는 것은 어려워질 것이다. 따라서, 사용이 편리하고 기능이 풍부한 임베디드 리눅스용 원격 통합 개발환경은 매우 중요하다.

SE-Lab에서는 Qplus-P 임베디드 리눅스용 원격 통합개발환경인 EIDE를 개발하였다. EIDE는 소스코드 편집기, 프로젝트 관리자, 크로스 개발 툴체인(크로스 컴파일러, 링커, 바이너리 유ти리티, 라이브러리), 원격 디버거, 원격 모니터 등의 도구들로 구성되며 모두 사용하기 편리한 GUI 기반이다. EIDE의 목표는 호스트/타겟 환경이라 할지라도 개발자가 GUI로 된 통합개발환경을 벗어나지 않으면서 다양한 도구들을 이용하여 원격 개발에 필요한 모든 것을 편리하게 수행할 수 있도록 하는 것이다.

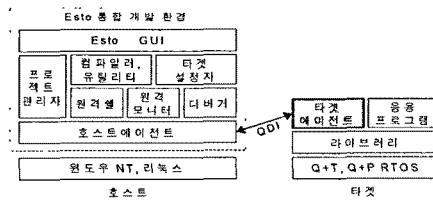
본 논문은 위와 같은 EIDE의 목표를 만족시키기 위한 타겟 에이전트를 제안한다. 타겟 에이전트는 타겟에서 실행되는 대본 프로세스로써 EIDE의 다양한 도구가 타겟에 접근하는데 필요한 다양한 기본 기능들을 제공한다. 타겟 에이전트 하나가 여러 도구의 요청을 처리하므로 타겟이 부팅된 후 타겟 에이전트를 한 번만 실행시키면 된다. 따라서, 각 도구를 사용할 때마다 별도의 서버를 타겟에 실행시켜야 하는 번거로움이 없어진다. EIDE 사용자는 응용을 개발하는 동안 타겟에 대해 신경쓰지 않아도 되며 통합개발환경을 벗어날 필요가 없다.

II. 연구 배경

2.1 Qplus-P/EIDE : Qplus-P를 위한 원격 통합개발환경

Qplus-P는 홈서버, PDA 등과 같은 정보가전을 주된 대상으로 하는 임베디드 리눅스이다. 임베디드 리눅스는 리눅스가 가지는 장점(안정성, 공개 소스, 다양한 아키텍쳐 지원 등)을 그대로 활용할 수 있다. 그러나 아직 충분히 검증되지 않았고 개발환경이 어렵다는 단점도 있다. 특히 통합되어 있지 않은, 다양한 기능의 복잡한 명령들에 의존해야 하는 어려운 개발환경은 초보 개발자뿐만 아니라 다수의 편리한 개발 도구에 익숙해져 있는 기존의 임베디드 소프트웨어 개발자들의 접근을 방해하는 주된 원인이 되고 있다. 그러므로 사용하기 쉬운 소프트웨어 개발 도구의 개발은 임베디드 리눅스 시스템 개발자들을 위해 매우 중요하다.

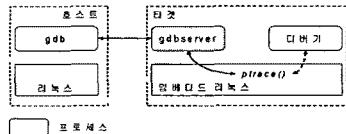
EIDE는 임베디드 리눅스를 위해 개발된 소프트웨어 개발환경으로 에디터, 크로스 컴파일러를 포함하는 툴 체인(Tool chain), 디버거 등의 기본적인 개발 도구들을 비롯하여 시스템 모니터와 소프트웨어 전력소모 측정기 등을 통합개발환경으로 제공한다. 그리고 호스트에이전트와 타겟 에이전트가 호스트와 타겟 사이의 명령어나 데이터 전달을 위한 통신 인터페이스를 지원한다.



(그림 1) EIDE의 구조

2.2 일반적인 디버깅 원리

리눅스에서 어떤 프로그램을 디버깅하기 위해서는 `ptrace()` 시스템콜을 이용한다. (그림 2)는 호스트에 있는 gdb가 타겟에 있는 디버거(debuggee, 디버깅 중인 프로세스)를 원격 디버깅하기 위하여 `gdbserver`를 이용하는 구조를 보여준다. gdb가 어떤 명령을 `gdbserver`에게 보내면, `gdbserver`는 `ptrace()` 시스템콜을 사용하여 디버기를 제어하거나 디버기에 대한 정보를 알아내서 gdb에게 보내준다.



(그림 2) gdb와 gdbserver를 이용한 원격 디버깅 구조

메모리나 레지스터의 값을 읽거나 쓸 경우, `gdbserver`는 (그림 3)처럼 `ptrace()`를 호출한다. 정지점에서 정지되어 있는 디버거를 `continue`(다음 정지점까지 계속 실행)시키거나 `step`(한 명령어만 실행)시키는 경우 `ptrace()` 시스템콜뿐만 아니라 `wait()` 시스템콜도 사용한다. gdb가 `gdbserver`에게 `continue`(또는 `step`) 명령을 보내면 `gdbserver`는 (그림 4)처럼 `ptrace()`를 호출하여 디버거를 `continue`(또는 `step`) 시킨다. 그런 다음 `gdbserver`는 `wait()` 시스템콜을 호출하여 디버기가 다음 정지점에서 멈출 때까지 기다린다. 디버기가 정지점을 만나면(또는 한 명령어를 실행하고 나면) 디버기는 멈추게 되고 리눅스 커널은 `SIGCHLD` 시그널을 `gdbserver`에게 보내준다. `wait()` 시스템콜에서 블록되어 있던 `gdbserver`는 `SIGCHLD` 시그널을 받고 깨어나서 디버기의 상태 정보를 gdb에게 알려준다.

```

/* Read 32 bits from memory at addr of process pid */
buf = ptrace(PTRACE_PEEKTEXT, pid, addr, 0);
/* Write 32 bits to memory at addr of process pid */
ptrace(PTRACE_POKETEXT, pid, addr, buffer);
/* Read a register of process pid */
ptrace(PTRACE_PEEKUSER, pid, regaddr, 0);
/* Write to a register of process pid */
ptrace(PTRACE_POKEUSER, pid, regaddr, regs);

```

(그림 3) 메모리와 레지스터 읽기/쓰기

```

/* Continuing a process */
ptrace(PTRACE_CONT, pid, 1, signal),
pid = wait(&status);

/* Stepping a process */
ptrace(PTRACE_SINGLESTEP, pid, 1, signal);
pid = wait(&status);

```

(그림 4) 프로그램을 continue 또는 step 시키기

본 논문에서는 ptrace() 시스템콜을 사용하는 기능을 ptrace() 의존적인 기능이라고 한다. ptrace() 의존적인 기능으로는 메모리 읽기/쓰기, 레지스터 읽기/쓰기, continue, step의 여섯 가지 기본 기능 외에 멀티쓰레드 디버깅을 위한 심볼 정보 요청과 쓰래드 정보 알림 기능이 있다.

III. 타겟 에이전트의 기본 기능

타겟 에이전트의 기능은 크게 네 가지 종류로 구분된다.

(1) 호스트에이전트와 연결

호스트에이전트와 연결을 설정하거나 끊는 기능

(2) 파일 및 디렉토리 관리

- 호스트의 파일을 타겟에 전송
- 타겟의 파일을 호스트로 복사 또는 이동
- 타겟의 파일을 삭제
- 타겟의 디렉토리 정보 알림

(3) 타겟의 프로세스 제어

- 타겟에 있는 프로그램을 실행
- 타겟에서 실행 중인 프로세스를 일시정지(suspend) 또는 종료(kill)
- 일시정지된 프로세스의 재시작(resume)

(4) 디버깅

타겟 에이전트의 정지점(breakpoint) 기반 디버깅은 멀티쓰레드 디버깅을 지원한다.

- 프로세스나 쓰래드의 메모리 읽고 쓰기.
- 프로세스나 쓰래드의 레지스터 읽고 쓰기
- 타겟에서 발생한 이벤트를 호스트에 알리기
- 정지점에 의해 정지된 프로세스나 쓰래드를 다음 정지점까지 실행(continue)
- 정지점에 의해 정지된 프로세스나 쓰래드의 한 어셈블리 명령어만 실행(step)
- 쓰래드 정보를 호스트에 알림

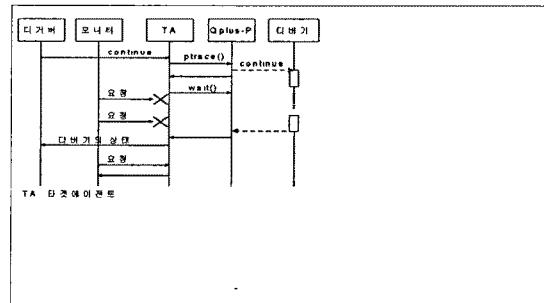
타겟 에이전트는 실시간 응용을 정지시키지 않고 디버깅할 수 있는 논스탑(non-stop) 디

버깅[7]을 지원한다. 원하는 곳에 정지점 대신 추적점(tracepoint)을 설정하고, 각 추적점마다 어떤 정보를 수집할지 지정한 후 프로그램을 실행시킨다. 타겟 에이전트는 추적점을 만날 때마다 사용자가 지정한 정보를 수집한다. 그런 다음, 사용자는 재생(replay) 기능을 이용하여 실제로 디버깅하는 것처럼 각 추적점들에서 레지스터, 메모리, 지역변수, 전역변수, 시간 등의 정보를 확인한다. 멀티쓰레드 디버깅 및 논스탑 디버깅에 대한 구체적인 방법은 본 논문에서 다루지 않는다.

IV. 타겟 에이전트의 기본 구조

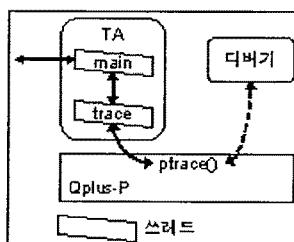
타겟 에이전트는 여러 도구들의 요청을 동시에 처리할 수 있어야 한다. 즉, 디버거의 요청을 오랫동안 처리하는 동안에도 모니터의 요청을 받으면 이를 처리할 수 있어야 한다.

그런데, 타겟 에이전트가 싱글 쓰레드로 구성되어 있는 경우 (그림 5)처럼 동시에 여러 도구의 요청을 처리할 수 없다. 타겟 에이전트가 디버거로부터 continue 요청을 받으면 ptrace() 시스템콜을 호출하여 디버거를 continue 시킨다. 그런 다음, 타겟 에이전트는 바로 wait() 시스템콜을 호출하여 디버거가 멈출 때까지 기다린다. 만약, 디버거가 다음 정지점을 만나기 전까지 오랫동안 실행을 하게 되거나 무한루프를 만나게 되면 타겟 에이전트는 오랫동안 블록되어 있게 되므로 모니터와 같은 다른 도구가 보내는 요청을 받을 수 없다. 즉, EIDE의 여러 도구들은 정상적으로 동작할 수 없게 된다.



(그림 5) 타겟에이전트가 싱글쓰레드인 경우의 문제점

본 논문에서는 이와 같은 문제를 해결하기 위하여 멀티쓰레드 구조로 된 타겟 에이전트를 제안한다. 타겟 에이전트는 (그림 6)처럼 하나의 메인(main) 쓰레드와 여러 개의 트레이스(trace) 쓰레드로 구분된다. 트레이스 쓰레드는 ptrace() 의존적인 기능만 처리한다. 메인 쓰레드는 타겟 에이전트의 실행이 시작될 때 자신을 초기화한 후 도구들의 요청을 받아서 ptrace() 의존적인 기능이 아닌 경우 직접 처리한다.



(그림 6) 타겟 에이전트의 구조

호스트의 디버거가 타겟의 어떤 프로그램을 디버깅하겠다고 요청하면, 타겟 에이전트는 트레이스 쓰레드를 생성한다. 트레이스 쓰레드는 요청한 프로그램을 자식 프로세스로 생성한다. 디버거가 ptrace() 의존적인 요청을 보내면, 타겟 에이전트의 메인 쓰레드는 이 요청을 트레이스 쓰레드에게 넘긴다. 트레이스 쓰레드가 ptrace() 시스템콜을 이용하여 요청을 처리한 후 그 결과를 메인 쓰레드에게 보내면, 메인 쓰레드는 디버거에게 결과를 리턴한다.

디버기가 다음 정지점을 만날 때까지 오래 걸리더라도 트레이스 쓰레드만 블록되므로 메인 쓰레드는 다른 도구의 요청을 받아서 처리할 수 있다.

메인 쓰레드와 트레이스 쓰레드는 전역변수를 공유하기 때문에 메모리 복사가 없다. 두 쓰레드 간의 동기화에는 세마포어를 사용하였다.

V. 타겟 에이전트의 구현

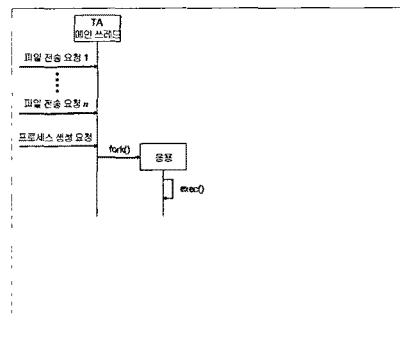
본 장에서는 타겟 에이전트가 EIDE의 도구와 어떻게 동작하고, 어떤 면에서 원격 개발을 편리하게 해주는지 설명한다.

임베디드 리눅스가 부팅되면서 타겟 에이전트를 테몬으로 실행시키면 EIDE 사용자는 호스트에이전트를 타겟 에이전트에 연결하는 작업만 하면 된다.

5.1 원격 실행

EIDE의 IDE에는 "Execute in Target" 메뉴와 버튼이 있는데 이를 선택하면 개발 중인 프로그램을 타겟에서 편리하게 실행할 수 있다.

"Execute in Target"을 선택했을 때 타겟 에이전트는 (그림 7)처럼 실행된다. EIDE의 IDE는 호스트에 빌드되어 있는 실행파일을 타겟 에이전트에게 보낸다. 타겟 에이전트는 실행파일 받아서 타겟에 저장한다. 실행 파일이 모두 전송되면 IDE는 타겟 에이전트에게 프로세스 생성을 요청한다. 이 요청의 인자로 실행할 프로그램의 위치(path)와 환경변수, 그리고 실행옵션으로 0을 보낸다. 타겟 에이전트의 메인 쓰레드는 fork() 시스템콜을 호출하여 새로운 프로세스를 생성한다. 새로운 프로세스는 exec() 시스템콜을 호출하여 EIDE IDE가 요청한 프로그램을 실행한다. 실행을 마치면, EIDE IDE는 타겟 에이전트에게 프로그램 삭제 요청을 보내서 타겟에 전송한 프로그램을 삭제함으로써 "Execute in Target"을 끝낸다.



(그림 7) 타겟에서 프로그램 실행

5.2 원격 디버깅

EIDE의 디버거는 타겟 에이전트에게 파일 전송 요청을 보내서 디버깅할 프로그램을 타겟에 전송한다. 그런 다음 디버깅할 프로그램을 실행하기 위하여 프로세스 생성 요청을 보내

는데, 이 때 요청의 인자 중에서 실행옵션을 1로 한다. 그러면, 타겟 에이전트는 트레이스 쓰레드를 생성한다. 트레이스 쓰레드는 (그림 8)의 trace_main() 함수를 실행한다.

```

01: void* trace_main(void* arg)
02: {
03:     pid = fork();
04:     if (pid == 0) {
05:         ptrace(PTRACE_TRACEME, 0, 0, 0);
06:         exec();
07:     }
08:
09:     while (1) {
10:         sem_wait(sem_m2d);
11:         get command and arguments
12:         in shared memory;
13:         process command;
14:         write result to shared memory;
15:         sem_post(sem_d2m);
16:
17:         if (command is cont or step) {
18:             wato;
19:             make event,
20:             notify IR of event occurrence;
21:         }
22:     }
23: }

```

(그림 8) 트레이스 쓰레드의 알고리즘

먼저 디버깅할 프로세스를 생성한다(라인 3). 새로 생성된 디버기 프로세스는 자신을 디버깅 될 수 있는 상태로 만들기 위하여 PTRACE_TRACEME 옵션으로 ptrace() 시스템콜을 호출(라인 5)한 후 exec() 시스템콜을 호출하여 디버깅할 프로그램으로 자신의 텍스트를 바꾼다. 디버기 프로세스는 exec()을 마친 후 새로운 프로그램의 첫번째 줄에서 멈추는데 이는 디버기 프로세스가 PTRACE_TRACEME 옵션으로 ptrace() 시스템콜을 호출하였기 때문이다.

트레이스 쓰레드는 while 문에 진입하여 메인 쓰레드로부터의 요청을 기다렸다가 요청을 처리한 후 다시 메인 쓰레드로 돌려주는 것을 디버기가 종료될 때까지 반복한다. 이 때 메인 쓰레드와 트레이스 쓰래드 간의 동기화가 필요한데, 타겟 에이전트는 두 개의 세마포어를 이용한다. sem_m2d 세마포어는 트레이스 쓰래드를 활성화시키기 위한 것이고, sem_d2m 세마포어는 메인 쓰래드를 활성화시키기 위한 것이다. (그림 9)는 트레이스 쓰래드와 동기화하기 위한 메인 쓰래드의 알고리즘이다.

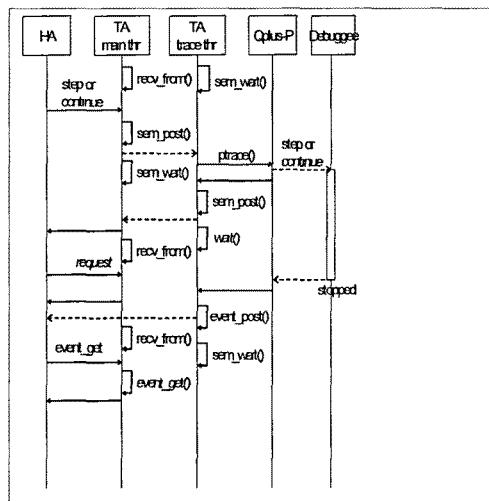
01: save current cmd arguments
02: latched variable,
03: sem_post(sem_m2d),
04: sem_wait(sem_d2m),
05: return result,

(그림 9) 트레이스 쓰래드와 동기화하기 위한 메인 쓰래드 알고리즘

(그림 10)은 디버기를 continue 또는 step 시키는 절차를 보여준다. 메인 쓰래드가 호스트로부터 요청을 받으면, 요청에 대한 명령과 인자를 공유 변수에 저장하고 sem_post()를 호출하여 트레이스 쓰래드를 활성화시킨다(그림 9의 라인 1~3). 그러면, sem_wait()를 호출하여 블록되어 있던 트레이스 쓰래드(그림 8의 라인 10)가 깨어나서 공유 변수에 기록되어 있는 명령어(continue 나 step)와 인자를 읽은 다음 요청을 처리한다(그림 8의 라인 11~13). 트레이스 쓰래드는 ptrace() 시스템콜을 호출하여 디버기를 continue 또는 step 시킨다. continue나step은 결과값이 없어서 공유 변수에 기록할 것이 없지만, 메모리 읽기처럼 결과

가 있는 경우에는 공유 변수에 기록한다. 그런 다음, 트레이스 쓰레드는 세마포어를 사용하여 메인 쓰레드를 활성화(그림 8의 라인 14~15)시킨다. 블록되어 있던 메인 쓰레드(그림 9의 라인 4)는 깨어나서 트레이스 쓰레드가 리턴한 결과를 호스트의 도구에게 전송한다.

트레이스 쓰레드가 요청받은 명령이 continue 나 step 인 경우 wait() 시스템콜을 호출하여 디버기가 정지할 때까지 블록된다(그림 8의 라인 18). 디버기가 정지하게 되면 트레이스 쓰레드는 다시 깨어나서 디버기가 어떤 지점에서 어떤 시그널을 받아서 멈추었는지 등을 알아내어서 호스트에 이전트에게 보낼 이벤트로 만들고 전송한다(그림 8의 라인 19~20). 트레이스 쓰레드는 while 문의 첫번째 문장인 sem_wait()를 호출하여 다음 요청을 받을 때까지 블록된다(그림 8의 라인 10).



(그림 10) continue 또는 step

5.3 원격 모니터링

- EIDE의 모니터는 타겟에서 실행되는 프로세스의 상태, 메모리의 상태, 시스템 정보 등을 보여준다. 타겟 에이전트에는 모니터를 지원하기 위하여 타겟에서 실행한 프로그램의 표준 출력을 호스트로 보내주는 기능이 있다. 모니터가 타겟의 프로세스 정보를 알아내기 위해 ps를 실행줄 것을 타겟 에이전트에게 요청하면, 타겟 에이전트는 ps 명령을 실행한 후 그 표준 출력을 타겟의 콘솔이 아닌 모니터에게 보내준다. 모니터는 이를 가공하여 개발자에게 보여 준다. 이와 같은 모니터의 요청은 디버깅을 하고 있는 동안에도 언제든지 실행될 수 있다.

VI. 결론

타겟 에이전트는 Qplus-P/EIDE의 다양한 도구들이 타겟에 접근하는 데 필요한 기능들을 제공한다. 이 기능들은 호스트/타겟 환경에서 개발자가 GUI로 된 EIDE의 통합개발환경을 벗어나지 않고 다양한 도구들을 이용하여 원격 개발에 필요한 모든 것을 편리하게 수행하는 데 필요한 것들이다.

다른 임베디드 리눅스용 개발환경에서는 한 프로그램을 디버깅할 때마다 개발자가 프로그램을 타겟으로 전송한 후 gdbserver를 실행시키고, 다시 호스트 창에서 gdb를 실행시켜야 하므로 번거롭다. 타겟 에이전트는 한 번만 실행하면 여러 도구들의 요청을 처리할뿐만 아니라 사용자가 파일 전송, 프로그램 실행 등도 통합개발환경 안에서 할 수 있게 지원한다.

현재 타겟 에이전트는 반드시 호스트에이전트를 통해서 도구와 통신해야 하는데 이는 통신 오버헤드를 증가시킨다. 더욱 성능이 좋은 개발환경을 위해서 호스트에이전트 없이 도구와 타겟 에이전트가 직접 통신할 수 있도록 수정할 계획이다. 도구와 타겟 에이전트 간의 통신 프로토콜은 gdb 리모트 시리얼 프로토콜[1]을 확장하여 사용한다. 이 프로토콜을 사용하면 gdb와 같은 GNU 도구들과 쉽게 연동될 수 있다.

현재 타겟 에이전트에는 오로지 하나의 디버거만 연결될 수 있다. 여러 EIDE 사용자들이 하나의 타겟에 연결하여 디버깅할 수 있으려면 타겟 에이전트는 여러 디버깅 세션을 동시에 지원할 수 있어야 하는데 이는 향후과제이다.

참 고 문 헌

- [1] 김정시, 임형태, 김홍남, "Qplus-P 임베디드 소프트웨어를 위한 GUI 기반 디버깅 도구," 2002 한국정보과학회 컴퓨터시스템연구회 추계 학술발표회, pp.65-71.
- [2] 우덕균, 임채덕, 김홍남, 표창우, "임베디드 리눅스용 응용 소프트웨어 개발을 위한 IDE 구현," 2002 한국정보과학회 봄 학술발표논문집, pp.100-102.
- [3] R. Stallman, R. Pesh, S. Shebs, et al., 'Debugging with GDB: The GNU Source-Level Debugger,' 8th Ed., Mar. 2000.
- [4] WindRiver, "VxWorks 5.4", <http://www.windriver.com>
- [5] C.D. Lim, etc., "A Tool Broker in Remote Development Environments for Embedded Applications," IASTED 2000, Feb., 2000.
- [6] S.W. Son, etc., "Debugging Protocol for Remote Cross Development Environment," RTCSA 2000, Dec., 2000.
- [7] K.Y. Lee, etc., "A Desgin and Implementation of a Remote Debugging Environment for Embedded Internet Software," ACM SIGPLAN 2000 Workshop on LCTES, June. 2000.
- [8] K.S. Kong, etc., "A Design and Implementation of Debug Agent for Remote Debugging of Embedded Real-time Software," IASTED AI 2000, Feb., 2000.
- [9] T. Akgul, P. Kuacharoen, V.J. Mooney, and V.K. Madisetti, "A Debugger RTOS for Embedded Systems," Proc. of 27th Euromicro Conference, pp.264-269, 2001.
- [10] Timothy Kelly, "Debugging trends in embedded software development," PC/104 Embedded Solutions, 2001.
- [11] Harry Koehnemann and Timothy Lindquist, "Towards target-level testing and debugging tools for embedded software," Proc. of the conference on TRI-Ada '93, Oct., 1993.
- [12] Sang-Joon Nam, J. H. Lee, B. W. Kim, Y. H. Im, Y. S. Kwon, C. M. Kyung and K. G. Kang, "Fast development of source-level debugging system using hardware emulation," ASP-DAC 2000, pp. 401-404, ACM, 2000.