내장 실시간 프로그래밍을 위한 C 언어의 타임아웃 기능의 확장

(An Extension to Time-out Facility in C Language for Embedded Real-Time Programming)

이 신 † 양 승 민 ††
(Sheen Lee) (Seung-Min Yang)

요 약 실시간 프로그래밍에 있어 타임아웃 기능은 매우 중요한 기본적인 기능 중 하나이다. 그러나 내장 실시간 시스템에서 가장 많이 사용하는 C 언어에서는 타임아웃 기능을 지원하지 않는다. 그래서 C 언어에서 실시간 프로그래밍 환경을 제공하기 위해 언어 자체를 확장하거나 실시간 엔진(또는 커널)을 지원하는 연구들이 있었다. 그러나 이것은 특정 실시간 커널과 컴파일러가 필요하다는 문제점을 갖고 있다. 본 논문에서는 운영체제에 최소한의 의존성을 갖고 라이브러리와 매크로 함수만으로 C 언어에서 타임아웃 기능을 제공한다. 또한 개발자가 편리하게 타임아웃 기능을 사용할 수 있도록 매크로 정의를 통해 구조화된 _within 문장을 제공한다. 이 기능은 리눅스의 단일 쓰레드 환경과 도스 환경뿐만 아니라 다중 쓰레드 환경인 POSIX 쓰레드에서도 구현하였다.

키워드 : 실시간 시스템, 실시간 프로그래밍, 타임아웃

Abstract Time-out is one of the basic but important functions in real-time programming. However, the C language used commonly in the embedded real-time systems doesn't support this capability. For this capability, there have been numerous studies on language extension and/or special purpose real-time kernel (or engine). Those require preprocessor or new kernel support. In this paper, we propose a time-out facility supported by a library and some macro functions with a minimum dependency on operating systems. Furthermore, we also provide a structured _within statement, a macro function which makes programming easy. We have implemented this for the LINUX and the DOS environment, and for the POSIX multithread environment as well.

Key words : real-time system, real-time programming, time-out, timeout

1. 서 론

실시간 프로그래밍에 있어 타임아웃 기능은 매우 중요한 기본적인 기능 중 하나이다. 실시간 프로그래밍에 있어 중요한 점은 수행 시간의 예측성이다. 그러나 프로그램의 최대 수행 시간을 예측하기 위해서는 많은 제약사항을 갖고 있는 것이 사실이다. 또한 막연히 외부로부터의 사건을 기다리는 일은 전혀 수행 시간을 예측할 수 없다. 이 경우 타임아웃은 수행 시간 예측에 대한 현실적인 대안이다.

그동안 제안된 많은 실시간 프로그래밍 언어들은 대부분 타임아웃 기능을 제공하고 있다[1][2]. 그러나 실제로 실시간 프로그래밍을 가장 많이 필요로 하는 내장 시스템에서는 제안된 실시간 프로그래밍 언어들보다는 C 언어를 더 많이 사용하고 있다[3]. 내장 시스템은 비교적 적은 메모리를 갖고 있기 때문에 효율적인 코드를 생성하고, 하드웨어를 손쉽게 제어할 수 있는 프로그래밍 언어가 필요했다. C 언어는 이에 적합한 언어이다. 뿐만 아니라 C 언어는 발표된 이래로 개발자들이 꾸준하게 사용해온 매우 친숙한 프로그래밍 언어이다. 그러나 실제로 C 언어에서 제공하는 실시간 프로그래밍 지원은 전혀 없다. 그래서 C 언어에서 실시간 프로그래밍

· 본 연구는 한국과학재단 특정기초연구과제(과제번호 1999-1-303-006-3) 지원으로 수행되었음
† 비 회 원 : (주)엠스톤 연구원
tactlee@realtime.ssu.ac.kr
†† 종신회원 : 숭실대학교 컴퓨터학부 교수
yang@computing.ssu.ac.kr
논문접수 : 2001년 9월 28일
심사완료 : 2002년 4월 16일

환경을 제공하기 위해 언어 자체를 확장하거나, 실시간 커널(또는 엔진)을 지원하는 연구들이 있었다[1][2]. 그러나 이것은 특정 실시간 커널과 컴파일러가 필요하다는 문제점을 갖고 있다.

본 논문에서는 운영체제에 최소한의 의존성을 갖고 라이브러리와 매크로 함수만으로 C 언어에서 타임아웃 기능을 제공한다. 또한 개발자가 편리하게 타임아웃 기능을 사용할 수 있도록 매크로 함수 정의를 통해 구조화된 __within 문장을 제공한다. 이 기능은 리눅스의 단일 쓰레드 환경과 도스 환경뿐만 아니라 리눅스의 다중 쓰레드 환경인 POSIX 쓰레드에서도 구현하였다. 그동안 C 언어에서 타임아웃 기능을 사용하기 위해 alarm(), signal(), setjmp(), longjmp() 등의 함수를 조합하는 방식을 사용했다[4]. 그러나 이러한 방식은 1) 초 단위의 낮은 시간 해상도만을 제공하고, 2) 중복(nesting)해서 타임아웃을 사용할 수 없으며, 3) 다중 쓰레드 환경에서는 사용할 수 없는 문제점을 갖고 있다. 본 논문에서 제공하는 타임아웃 기능은 이러한 문제점들을 모두 해결하고 있다. 또한 __within 문장을 통해서 내장 실시간 시스템 개발자는 좀 더 편리하게 실시간 프로그래밍을 할 수 있다.

본 논문에서는 먼저 C 언어에서 타임아웃 기능을 사용하기 위한 기존 연구들에 대한 소개와 문제점을 알아본다. 3장에서는 C 언어에서 타임아웃 기능을 제공하기 위한 제약사항과 요구사항에 대해 설명하고, 타임아웃 라이브러리와 __within 문장의 구현 원리와 API를 설명한다. 4장에서는 __within 문장을 이용하여 손쉽게 타임아웃 기능을 사용하는 예를 보여준다. 5 장에서는 결론 및 향후 연구 방향에 대해 논한다.

2. 기존 연구

2장에서는 그동안 C 언어에서 타임아웃 기능을 사용했던 방법과 문제점에 대해 알아보고, C 언어를 확장한 대표적인 기존 실시간 프로그래밍 언어들에 대한 소개와 문제점을 알아본다.

2.1 타임아웃 기능을 사용하기 위한 C 프로그래밍 기법

C 언어는 문법에 예외 처리 기능을 갖고 있지 않다. 때문에 C 언어에서 예외 처리 기능을 제공하기 위한 연구들도 있었고[5][6][7], C++에서는 예외 처리 문장이 문법에 추가되기도 하였다. 이들은 기본적으로 setjmp()와 longjmp()를 이용하고 매크로 함수를 제공하여 편의성을 높이고 있다. C 언어에서 타임아웃을 지원하기 위해 사용했던 방법도 동일한 방법이다. 다음은 C 언어에서 타임아웃 기능을 위해 사용했던 방법을 설명하기 위해 작성한 예제 프로그램이다.

```
 1: #include <stdio.h>
 2: #include <setjmp.h>
 3: #include <signal.h>
 4:
 5: #define MaxLine 1024
 6:
 7: jmp_buf jmpbuf;
 8: char line[MaxLine];
 9:
10: sig_alrm(int sig_no)
11: {
12:   longjmp(jmpbuf, 1)
13: }
14:
15: main()
16: {
17:   signal(SIGALRM, sig_alrm);
18:   alarm(5);
19:   if (setjmp(jmpbuf) == 0)
20:   {
21:     if ( (n = read(0, line, MaxLine)) < 0)
22:       err_exit("read error!");
23:   }
24:   else
25:   {
26:     err_exit("read timeout!");
27:   }
28:   alarm(0);
30:   write(1, line, n);
31: }
```

위 예제 프로그램의 18번 줄에서 alarm(5)을 호출하면, 이 후 5초 후에 SIGALRM 시그널이 발생할 것이다. 만일 21번 줄에서 read()가 5초 이전에 일을 마치지 못한다면 SIGALRM 시그널이 발생하여, 17번 줄에서 등록한 시그널 처리 함수 sig_alrm()이 호출되고, 12번 줄의 longjmp()가 실행될 것이다. 그러면 실행은 19번 줄의 setjmp()로 옮겨지고, 되돌림 값은 0이 아닌 longjmp()에서 넘겨준 1이라는 값을 넘겨준다. 결국 24번 줄의 else 이후 문장들이 실행된다. 만일 SIGALRM 시그널이 발생하기 전에 read()가 실행을 마친다면, 28번 줄의 alarm(0)이 실행되어 타이머를 정지시킬 것이다.

위 예제 프로그램은 alarm(), signal(), setjmp(), longjmp() 등을 이용해서 타임아웃 기능을 구현하고 있다. 그러나 이러한 방식은 1) 초 단위의 낮은 시간 해상도만을 제공하고, 2) 중복(nesting)해서 타임아웃을 사용할 수 없으며, 3) 다중 쓰레드 환경에서는 사용할 수

없는 문제점을 갖고 있다.

문제 1)과 2)의 원인은 alarm()에 있다. alarm()은 초 단위의 낮은 시간 해상도만을 제공할 뿐만 아니라, 프로세스 당 단 한 개만 사용할 수 있다[8]. 이러한 문제점을 해결하기 위해 [9]에서는 다중 타이머를 구현하고 있다.

문제 3)의 원인은 시그널과 setjmp(), longjmp()에 있다. setjmp()는 현재의 스택 환경을 보관하는 일을 한다. 그리고 longjmp()는 setjmp()에서 보관해놓은 스택 환경을 복원하는 일을 한다. POSIX 쓰레드 환경에서 SIGALRM 시그널은 쓰레드가 아닌 프로세스에게만 전달되며, setjmp()와 longjmp()가 서로 다른 쓰레드에서 동작할 가능성이 있다[10]. 각 쓰레드는 자신의 스택을 갖고 있기 때문에, 만일 setjmp()와 longjmp()가 서로 다른 스택에서 실행된다면, longjmp() 실행 시 다른 쓰레드에서 실행된 setjmp()가 보관해 놓은 스택 환경을 현재 실행되고 있는 쓰레드의 스택에 복원하게 되므로, 프로그램 실행에 치명적인 오류가 발생하게 된다.

2.2 C 언어를 확장한 대표적인 기존 실시간 언어

C 언어를 확장한 대표적인 기존 실시간 프로그래밍 언어로는 Real-Time Concurrent C[1]와 RTC++[2]를 들 수 있다. Real-Time Concurrent C는 Concurrent C[11][12]에 실시간 특성을 부여한 프로그래밍 언어이다. 유닉스 환경을 기반으로 구현되어 있으며 타임아웃 기능은 다음과 같이 within deadline 문장을 통해 제공하고 있다.

```
within deadline (time)
    statement1
else
    statement2
```

RTC++는 ARTS 커널을 기반으로 구현되었으며, ARTS/RTC++ 플랫폼은 분산 객체지향 실시간 시스템 개발 환경을 제공한다. RTC++ 역시 다음과 같이 within 문장을 통해서 타임아웃 기능을 제공한다.

```
within (time) {
    statement1
} except {
case timeout:
    statement2
}
```

Real-Time Concurrent C와 RTC++는 모두 1990년대 초에 만들어진 실시간 프로그래밍 언어이다. 이들은 모두 편리하게 타임아웃 기능을 사용할 수 있는 방법을 제공하고 있다. 그러나 10년이 지난 현재, 이들이 내장 실시간 시스템 프로그래밍에 널리 사용되고 있지는 못하다. 이들을 사용하기 위해서는 C 컴파일러 이외에 별도의 컴파일러와 실시간 커널이 필요하기 때문이다. 이것은 내장 시스템에서 사용되는 매우 다양한 마이크로 컨트롤러들에 적응하지 못하는 이유가 되었다.

3. 설계 및 구현

3.1 목표 및 요구사항

본 논문에서 제공하는 타임아웃 기능의 설계 및 구현 목표는 적응성이 높으면서도 Real-Time Concurrent C나 RTC++처럼 편리한 타임아웃 기능을 제공하는 것이다. 이를 위한 세부적인 제약사항 및 요구사항은 다음과 같다.

요구 1 : 컴파일러의 확장 없이 매크로 함수와 라이브러리만으로 확장해야 한다.

요구 2 : 리눅스(유닉스)의 단일 쓰레드 환경과 도스 환경에서 모두 사용 가능해야 한다.

요구 3 : POSIX 다중 쓰레드 환경에서 사용 가능해야 한다.

요구 4 : alarm()에서 제공하는 초 단위보다 높은 시간 해상도를 제공해야 한다.

요구 5 : 타임아웃 블록을 중복(nesting)해서 사용할 수 있어야 한다.

요구 6 : break 문장을 사용하여 타임아웃 블록을 빠져나올 수 있어야 한다.

3.2 구조 및 구현 원리

3.2.1 구조

요구 1을 만족하기 위해 본 논문에서는 매크로 함수와 라이브러리만으로 C 언어에서 타임아웃 기능을 편리하게 사용할 수 있도록 구현했다. 그림 1은 본 논문에서 구현한 타임아웃 라이브러리와 매크로 함수의 구성을 보여준다.

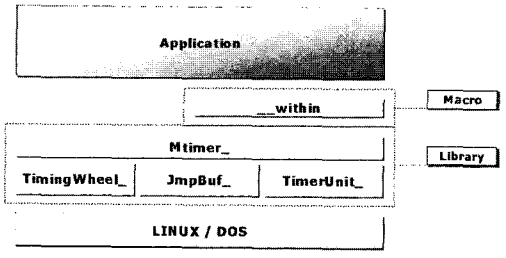

그림 1 타임아웃 라이브러리의 구조

타임아웃 라이브러리는 크게 Mtimer_, TimingWheel_, JmpBuf_, TimerUnit_의 4부분으로 구성되어 있다. Mtimer_는 나머지 3부분에 대한 인터페이스를 담당하고, 실제 기능 구현은 나머지 3부분에서 담당한다. TimingWheel_은 다중 타이머 기능을 제공한다. TimingWheel_은 [13]에서 제안한 타이밍 휠(timing wheels) 구조를 사용하고 있다. 이 구조는 리눅스 커널에서 타이머를 관리하기 위한 구조로 사용하고 있다[14]. [13]에서는 휠의 크기가 커지게 될 경우, 시간 해상도가 다른 여러 개의 휠을 두는 방법과 휠에 등록된 타이머에 카운터를 두는 방식을 제안하고 있다. 리눅스에서 구현된 타이머는 전자를 사용하고 있고, 본 논문에서 구현한 다중 타이머는 후자를 사용하고 있다. JmpBuf_는 setjmp()와 longjmp()에서 사용하는 jmp_buf 구조체들의 동적 할당을 관리한다. TimerUnit_는 타이머 구조체의 동적 할당을 관리한다.

__within 매크로는 C 언어에서 타임아웃 기능을 편리하게 사용할 수 있도록 만든 매크로 함수들을 정의한 것이다. Real-Time Concurrent C와 RTC++에서 제공하는 *within* 문장을 *__within* 문장으로 표현하면 다음과 같다.

```
__within (time)
    statement1
__within_timeout
    statement2
__end_of_within
```

3.2.2 리눅스의 단일 쓰레드 및 도스 환경에서의 구현 원리

타임아웃 기능을 구현하기 위한 리눅스의 단일 쓰레드 환경과 도스 환경은 매우 유사하다. 타임아웃 기능을 사용하기 위해서는 타이머를 구현해야 한다. 리눅스에서는 타이머를 구현하기 위해 주기적으로 시그널을 발생시키고 그때마다 시그널 처리기를 실행시킨다. 도스 환경도 이와 비슷하여 시그널 대신 인터럽트를 발생시키고 그때마다 인터럽트 처리기를 실행시킨다. 즉 리눅스의 단일 쓰레드 환경과 도스 환경은 초기 시그널 처리기(또는 인터럽트 처리기)를 설정하는 부분과 주기적으로 시그널(또는 인터럽트)을 발생시키도록 설정하는 부분만 달라진다. 이 부분은 최초 라이브러리 소스 컴파일 시 조건부 컴파일을 한다.

그림 2는 리눅스의 단일 쓰레드 및 도스 환경에서 타임아웃 기능이 동작하는 순서를 보여준다. *__within* 문장에서는 JmpBuf를 할당받아 식별자를 보관하고, setjmp()를 호출하여 되돌림 값을 보관한다. 그리고 TimerUnit를 할당받아 식별자를 보관하고, TimerUnit과 JmpBuf를 연결하여 TimingWheel에 등록한다. TimingWheel은 매 주기마다 시계의 초바늘처럼 시간을 증가시키고 현재 시각에 실행시켜야할 타이머가 있는 지 검사한다. 만일 현재 시각에 실행시켜야할 타이머가 있다면 TimerUnit에 보관하고 있던 JmpBuf를 이용하여 longjmp()를 호출한다. 그러면 __within 문장 내부에서 호출했던 setjmp()로 제어가 옮겨진다.

__within_timeout 문장에서는 __within 문장에서 보관해 놓았던 *setjmp()*의 호출 결과를 보고 타임아웃 여부를 판단한다. 만일 타임아웃이라고 판단되면 *__within_timeout* 문장 뒤의 실행문으로 제어를 옮긴다. 그렇지 않다면 *__end_of_within* 문장으로 제어를 옮긴다.

__end_of_within 문장에서는 *__within* 문장에서 보관해 놓았던 *JmpBuf*와 *TimerUnit*의 식별자를 이용해서 할당받은 *JmpBuf*와 *TimerUnit*를 해제한다.

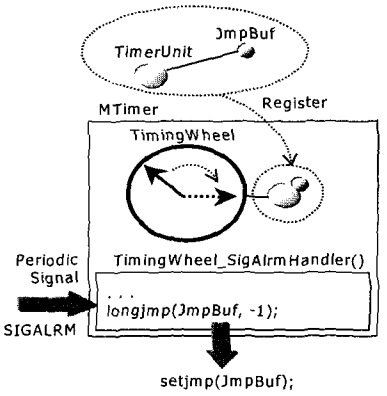

그림 2 단일 쓰레드 및 도스 환경에서 타임아웃 기능의 구현 원리

__within 문장 뒤에 이어오는 __within_timeout 문장과 *__end_of_within* 문장에서는 *__within* 문장에서 보관해 놓은 여러 값들을 사용한다. 그러므로 요구 5처럼 __within 문장이 또 다른 __within 문장을 내포하기 위해서는 각 __within 문장에서 보관하는 값은 서로 독립된 공간에 보관되어야 한다. 이를 위해 본 논문에서는 __within 문장을 매크로 함수로 정의할 때 블록을 구성하도록 하였다. 이렇게 하면 __within 문장 내부에서 정의되는 변수는 바깥 블록의 __within 문장에서 정의된 변수와 비록 이름은 같더라도 식별자의 가시권(visibility)이 달라지므로 동일 스택의 서로 다른 위치에 공간을

할당받는다. 다음은 __within 문장의 매크로 정의의 일부분을 보여준다.

```
1: #define __within(t) {\
2:     int __e, __timer_unit_i, __retval;\
3:     jmpbuf_t* __jmpbuf_p = JmbBuf_Alloc();\
4:     do {\
5:     if (!__jmpbuf_p)\
6:         __e = 0; /* error! */\
7:     . . .
```

위의 4번 줄에서는 do-while 문장을 이용하여 블록을 구성하는 것을 볼 수 있다. 이것은 요구 6을 만족하기 위한 것으로 break 문장을 이용해서 __within 블록을 탈출할 수 있도록 한 것이다. 다음은 do-while 문장을 완성하기 위한 __end_of_within의 매크로 정의를 보여준다.

```
1: #define __end_of_within }while(0);\
2:     JmpBuf_Free(__jmpbuf_p);\
3:     Timer_Cancel(__timer_unit_i);}
```

3.2.3 POSIX 쓰레드 환경에서의 구현 원리

POSIX 쓰레드 환경에서 타임아웃 기능을 구현하기 위해 가장 중요한 점은 setjmp()와 longjmp()가 동일한 쓰레드의 스택에서 실행될 수 있도록 처리해야 한다는 것이다. 리눅스의 단일 쓰레드 및 도스 환경에서는 setjmp()와 longjmp()를 호출한 쓰레드가 동일한 스택을 사용한다는 가정을 갖고 있다. 그러나 POSIX 쓰레드 환경에서는 각 쓰레드가 자신의 스택을 별도로 갖는다. 만일 setjmp()와 longjmp()가 서로 다른 쓰레드에서 실행된다면, setjmp()가 보관하고 있던 스택 환경과 longjmp()가 복원할 스택 환경이 달라지므로 프로그램 실행에 치명적인 오류가 발생할 것이다.

POSIX 쓰레드 환경에서는 SIGALRM 시그널이 쓰레드가 아닌 프로세스에게 전달된다[10]. 즉 SIGALRM 시그널이 발생하면 이에 대한 시그널 처리기는 main() 함수와 동일한 스택 환경에서 실행된다. 3.2.1에서는 이곳에서 longjmp()을 호출했다. 그러나 POSIX 쓰레드 환경에서는 setjmp()가 다른 쓰레드에서 실행될 수 있기 때문에 SIGALRM 시그널 처리기에서 longjmp()를 호출하는 것은 매우 위험하다. 이 문제를 해결하기 위해서는 longjmp()를 호출하는 시그널 처리기가 setjmp()를 호출한 쓰레드와 동일한 쓰레드의 문맥에서 실행되도록 해야 한다.

본 논문에서는 이 문제를 해결하기 위해서 __within 문장에서는 TimerUnit에 JmpBuf와 함께 쓰레드 식별자인 ThreadId를 보관한다. 매 주기마다 SIGALRM 시그널이 발생하고 이것은 TimingWheel을 움직인다. TimingWheel에서는 현재 시각에 등록되어 있는 TimerUnit에서 ThreadId를 보고 pthread_kill()을 이용해서 해당 쓰레드에게 SIGUSR1 시그널을 보낸다. 결국 해당 쓰레드의 스택에서 SIGUSR1을 처리하기 위한 시그널 처리기가 실행되고 이곳에서 longjmp()를 호출한다. 그림 3은 pthread_kill() 함수를 이용하여 POSIX 쓰레드 환경에서 타임아웃 기능이 동작하는 순서를 보여준다.

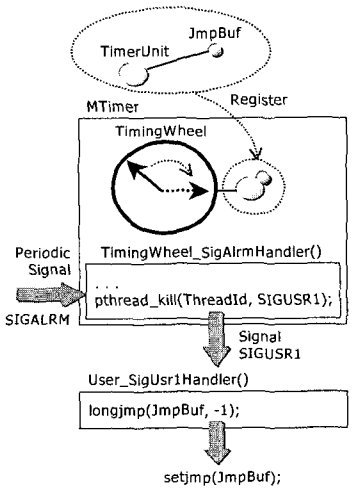

그림 3 POSIX 쓰레드 환경에서 타임아웃 기능의 구현 원리

3.3 API

다음은 본 논문에서 제공하는 타임아웃 라이브러리의 API이다.

Timer_Init() 함수는 타임아웃 라이브러리를 전체적으로 초기화하는 함수로서 타임아웃 기능을 사용하기에 앞서 제일 먼저 호출해야만 한다. Timer_Register() 함수는 타이밍 휠에 타이머를 등록하는 함수이다. 이 함수를 좀 더 편리하게 사용하기 위해서 Timer_Register JmpBuf()와 Timer_RegisterCallBack()의 두 함수를 제공한다. Timer_RegisterJmpBuf()는 타임아웃이 발생할 경우 jmpbuf_t에 대한 포인터를 이용해서 longjmp()를

```
void  Timer_Init(
            void);
int   Timer_Register(
            unsigned long timeout,
            timer_unit_state_t,
            void*,
            unsigned int param_ui,
            unsigned long param_ul);
int   Timer_RegisterJmpBuf(
            unsigned long timeout,
            jmpbuf_t*);
int   Timer_RegisterCallBack(
            unsigned long timeout,
            void (*function_p)(
                        unsigned int,
                        unsigned long),
            unsigned int param_ui,
            unsigned long param_ul);
void  Timer_Cancel(
            int timer_unit_i);
```

하도록 되어 있다. __within 문장의 이 함수를 이용하여 구현하였다. Timer_RegisterCallBack() 함수는 타임아웃이 발생할 경우 전달인자로 지정한 함수를 실행하도록 되어 있다. 이때 param_ui와 param_ul은 실행되는 콜백 함수에게 인자로서 전달된다.

본 논문에서는 위의 API 이외에도 좀더 편리하게 타임아웃 기능을 사용할 수 있도록 구조화된 __within 문장을 제공한다. __within 문장의 사용 형식은 다음과 같다.

__within (time)
    statement_normal
__within_error
    statement_error
__within_timeout
    statement_timeout
__end_of_within

__within 문장에서 지정한 시간 안에 statement_normal이 실행을 마치면, 제어는 __end_of_within 문장으로 옮겨진다. 그렇지 않다면 제어는 __within_timeout 문장의 뒤, 즉 statement_timeout으로 옮겨진다. 만일 __within 문장에서 JmpBuf나 TimerUnit 등을 할당받는데 문제가 발생한다면, 제어는 __within_error 문장 뒤, 즉 statement_error로 옮겨진다. __within_error 문장은 생략이 가능하다.

4. 사용 예 및 평가

다음은 2.1에서 설명했던 예제 프로그램을 __within 문장을 이용하여 바꾼 예이다. 2.1에서 설명했던 예제 프로그램보다 훨씬 가독성이 뛰어날 뿐만 아니라 코딩된 줄 수도 9줄이 줄어든 것을 알 수 있다. 뿐만 아니라 본 논문에서 제공하는 __within 문장은 3.1에서 언급한 6가지 요구사항을 모두 만족하고 있으면서도 타임아웃 라이브러리는 3k 바이트 정도의 작은 크기를 유지하고 있다. 또한 운영체제에 의존적인 부분은 타이머를 구동하기 위해 시그널 처리기 또는 인터럽트 처리기를 설정하는 부분 단 한 곳이므로 매우 높은 이식성을 갖고 있다.

```
 1: #include <stdio.h>
 2: #include "mtimer.h"
 3:
 4: #define MaxLine 1024
 5:
 6: char line[MaxLine];
 7:
 8: main()
 9: {
10:    Timer_Init();
11:    __within (500)
12:    {
13:       if ( (n = read(0, line, Maxline)) < 0)
14:          err_exit("read error!");
15:    }
16:    __within_timeout
17:    {
18:       err_exit("read timeout!");
19:    }
20:    __end_of_within
21:    write(1, line, n);
22: }
```

본 논문에서 제공하는 __within 문장은 비록 매크로 함수로 정의되어 있지만, 구조화 프로그래밍이 가능하도록 설계되어 있고, __within 문장 내에 또 다른 __within 문장들을 내포할 수 있도록 되어 있다. 그리고 break 문장을 이용해서 __within 블록을 빠져나오는 것도 가능하다. 그러나 매크로 함수 정의의 한계로 __within 블록 내에서 외부로 goto 문장을 이용해서 빠져나오면 __end_of_within 문장을 실행하지 못하여 오류가 발생할 수 있다.

또한 __within 블록 내에 또 다른 __within 블록을 사용하는 경우에는 __within 문장에서 타임아웃 시간을 주의해서 지정해야 한다. 만일 외부의 __within 문장에서 지정한 타임아웃 시간이 내부의 __within 문장에서 지정한 타임아웃 시간보다 작다면, 내부의 __end_of_

within 문장이 실행되지 못하는 문제가 발생할 수 있다.

5. 결론 및 향후 연구 방향

실시간 프로그래밍에서 타임아웃 기능의 중요성에도 불구하고, 내장 실시간 시스템을 구축하기 위해 가장 많이 사용되는 C 언어에서는 타임아웃 기능에 대한 지원이 미미하였다. 본 논문에서는 타임아웃 라이브러리와 매크로 함수를 통해 구조화된 __within 문장을 제공하여, 내장 실시간 시스템 개발자로 하여금 편리하게 타임아웃 기능을 사용할 수 있도록 하였다.

최근 리눅스를 내장 시스템에서 사용하려는 움직임이 매우 활발하다. 이에 따라 리눅스에서 디바이스 드라이버를 작성하기 위한 편리한 개발 환경의 중요성이 높아지고 있다. 타임아웃 기능은 리눅스의 사용자 프로세스에서뿐만 아니라 디바이스 드라이버를 작성할 때에도 많은 도움이 될 것이다. 그러나 본 논문에서 제공하는 타임아웃 라이브러리는 사용자 프로세스 수준에서의 구현이었다(리눅스 환경의 경우). 때문에 향후 리눅스의 디바이스 드라이버에서 타임아웃 기능을 쉽게 사용할 수 있는 방법에 대한 연구가 필요하다.

참 고 문 헌

[1] N. Gehani and K. Ramamritham, "Real-Time Concurrent C: A Language for Programming Dynamic Real-Time Systems," *Journal of Real-Time Systems*, Vol. 2, No. 3, pp. 377-405, December 1991.

[2] Y. Ishikawa, H. Tokuda, and C. W. Mercer, "Object-Oriented Real-Time Language Design: Constructs for Timing Constraints," *Technical Report CMU-CS-90-111*, Carnegie Mellon University, March 1990.

[3] Michael Barr, *Programming Embedded Systems in C and C++*, O'Reilly, 1999.

[4] W. Richard Stevens, *Advanced Programming in the UNIX Environment*, pp. 285-290, Addison-Wesley, 1992.

[5] Lee P., "Exception Handling in C Programs," *Software-Practice and Experience*, Vol. 13, No. 5, pp. 389-406, 1983.

[6] Open Systems Resources, Inc., "The Exception To The Rule: Structured Exception Handling," *The NT Insider*, Vol. 6, Issue 2, March/April 1999, http://www.osr.com/ntinsider/1999/seh.htm

[7] Bruce W. Bigby, "The GEF General Exception-Handling Library," *Dr. Dobb's Journal*, November 1998, http://home.rochester.rr.com/bigbyofrocny/GEF/GEF.html

[8] Neil Matthew and Richard Stones, *Beginning Linux Programming*, 2nd ed, Wrox Press, 1999.

[9] Kay A. Robbins and Steven Robbins, *Practical UNIX Programming: A Guide to Concurrency, Communication, and Multithreading*, Prentice-Hall, 1996.

[10] Bil Lewis and Daniel J. Berg, *Multithreaded Programming with Pthreads*, Prentice-Hall, 1998.

[11] N. H. Gehani et al, "Concurrent C," *Software Practice and Experience*, Vol. 16, No. 9, pp. 821-844, 1986.

[12] N. H. Gehani and W. D. Roome, *Concurrent C*, AT&T Bell Laboratories, 1985, http://www.cs.engr.uky.edu/~dclarke/cs570_98s/concurrentc.ps

[13] George Varghse and Anthony Lauck, "Hashed and Hierarchical Timing Wheels: Efficient Data Structures for Implementing a Timer Facility," *IEEE/ACM Transactions on Networking*, Vol. 5, No. 6, pp. 824-834, December 1997.

[14] Daniel P. Bovet and Macro Cesati, *Understanding the Linux Kernel*, O'Reilly, 2001.

이 신

1994년 2월 숭실대학교 전자계산학과 졸업(학사). 1997년 2월 숭실대학교 전자계산학과 졸업(석사). 2000년 2월 숭실대학교 컴퓨터학과 박사. (주)엠스톤 선임연구원. 관심분야는 실시간 시스템, 운영체제, 프로그래밍 언어 등

양 승 민

1978년 2월 서울대학교 전자공학과 졸업(학사). 1983년 4월 미국 Uni. of South Florida 전산학(석사). 1986년 12월 미국 Univ. of South Florida 전산학(박사). 1988년 ~ 1993년 미국 Univ. of Texas at Arlington 조교수. 1993년 ~ 현재 숭실대학교 컴퓨터학과 부교수. 1996년 ~ 1998년 국회도서관 정보처리국장. 관심분야는 실시간 시스템, 운영체제, 결함허용 시스템 등