논문 2004-41SC-1-4

고속 영상 검지기 시스템 개발에 관한 연구

(A Study On Development of Fast Image Detector System)

김 병 철*, 하 동 문*, 김 용 득**

(Byung Chul Kim, Dong Mun Ha, and Yong Deak Kim)

요 약

교통 분야에서도 역시 영상을 이용한 시스템의 개발이 주요 이슈가 되고 있다. 이는 영상을 이용한 시스템의 경우 설치비용이 기존 시스템들에 비해 엄청나게 저렴하다는 것과 설치하는 기간 중에도 교통의 흐름을 거의 방해하지 않고 설치가 가능하다는 장점을 가지고 있기 때문이다.

본 연구에서는 임베디드 시스템 환경에서 영상 검지기 시스템의 구현을 제안하였다. 전체 시스템은 호스트 컨트롤러 보드 부분과 영상처리 보드 부분으로 나뉜다. 호스트 컨트롤러 보드 부분은 전체 시스템의 제어와 외부와의 인터페이스, 그리고 OSD(On Screen Display) 부분을 담당하게 된다. 영상처리 보드 부분은 알고리즘의 적용, 마우스 신호의 베어를 담당하고 있다. 그리고 안정적인 호스트 컨트롤러의 보드의 운영을 위해 uC/OS-II를 호스트 컨트롤러 보드에 포팅하였다.

Abstract

Nowadays image processing is very useful for some field of traffic applications. The one reason is we can construct the system in a low price, the other is the improvement of hardware processing power, it can be more fast to processing the data.

In traffic field, the development of image using system is interesting issue. Because it has the advantage of price of installation and it does not obstruct traffic during the installation.

In this study, I propose the traffic monitoring system that implement on the embedded system environment. The whole system consists of two main part, one is host controller board, the other is image processing board. The part of host controller board take charge of control the total system, interface of external environment, and OSD(On screen display). The part of image processing board takes charge of image input and output using video encoder and decoder, image classification and memory control of using FPGA, control of mouse signal. And finally, for stable operation of host controller board, uC/OS-II operating system is ported on the board.

Keywords: 영상 검지기, 호스트 컨트롤 보드, 영상 처리 보드, uC/OS-II

I. 서 론

한정된 도로 및 시설자원에서 가장 극대화한 효율을 내기 위해서는 교토의 편중을 균등하게 하는 전문적이고 세밀한 교통 정도 전달 시스템의 구축과 함께 도로시설의 파손 및 교통의 흐름을 방해하는 요소를 제거하는 정책이 필요하다. 이 때 도로의 상황을 파악하여 정보를 생성하는 것이 중요하며 영상을 이용한 차량 인식 기법이 필수적으로 요구된다.

현재 국내외에서 개발되어 실용화된 대부분의 차량 인식 시스템은 검지센서 방식으로 가종 검지

세서들을 도로에 매설하거나 도로변에 설치하여 데이터를 수집한 후 이를 기반으로 차량을 검지하는 방법을 사용하고 있다. 하지만 이러한 방식은 검

* 학생회원, ** 정회원, 아주대학교 전자공학부

(Dept. Elecronics Engineering, Ajou Univ.)

접수일자 : 2003년1월17일, 수정완료일 : 2003년12월26일

지센서의 유지 보수에 필요한 비용과 인적 자원이 과다하고 여러 개의 검지센서를 포함하는 복잡한 현장 설비를 필요로 하는 단점을 가지고 있다.

하지만 영상을 이용한 차량 인식 시스템은 도로에 손상을 주지 않고서도 설치가 가능하다. 따라서 설치 시에는 교통 통제 등 불필요한 교통량의 부하를 주지 않을 수 있고, 설치가 검지센서 방식에 비해 매우 간단하므로 설치비 또한 절감할 수 있으며, 유지보수 또한 간편하며, 자체 설비의 가격 또한 저렴하다는 장점을 가지고 있다.

본 논문에서는 ARM7 마이크로 컨트롤러와 FPGA를 이용하여 저가의 고성능 차량 인식 시스템의 구현을 제안하고자 한다.

영상 데이터 저장용 메모리 2개를 사용함으로서 기존 시스템과 똑같은 CPU를 가지고도 보다 안정적인 시스템을 구축하며, 이를 통해 더 많은 데이터 처리하는데 보다 유연한 시스템을 개발하고자 한다.

본 시스템에서 FPGA는 마이크로컨트롤러가 연산을 안전하게 수행할 수 있도록 NTSC Decoder로부터의 데이터를 두개의 메모리에 분배시키는 역할과 함께, 데이터 버스의 제어와 OSD 메모리로부터 데이터를 읽어 그것을 모니터에 표시하는 역할까지 하게 된다.

호스트 컨트롤러 보드의 안정성을 높이고 어플리케이션의 개발을 편리하게 하기위해서 uC/OS-II를 포팅 하였다. uC/OS-II는 작은 용량의 커널을 가지고 있지만 선점형의 멀티태스킹이 가능한 운영체제이다. 또한 커널의 소스코드가 공개되어 있으며, 소스의 대부분이 ANSI-C로 쓰여 있기 때문에 각종 시스템에 적용하기가 쉽고, 롬화 할 수 있기 때문에 임베디드 시스템에 적용하기가 수월하다.

본 논문의 II장에서는 시스템의 하드웨어 구성을 살펴볼 것이며, III장에서는 시스템의 Firmware 구성 및 uC/OS-II의 시스템 포팅 과정을 자세하게 살펴보도록 할 것이다. IV장에서는 영상처리 알고리즘의 적용 과정을 알아보고, V장 결론에서는 개발된 시스템의 테스트 결과를 살펴보도록 할 것이다.

II. 하드웨어 구성

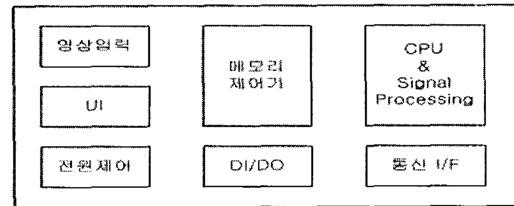


그림 1. 전체 하드웨어 블럭도
Fig 1. Total H/W block

본 연구에서 제작하려고 하는 하드웨어의 기본 블록도는 그림 1과 같다.

하드웨어는 II장의 PCB로 구성되어 있으며 전체 시스템을 제어하는 호스트 컨트롤 보드(이하 HST 보드)와 영상처리 알고리즘이 포팅되어 입력된 영상을 처리하는 영상처리 보드(이하 IMD 보드)로 나뉜다. 이는 영상처리 알고리즘의 처리와 전체 시스템 제어를 분리시킴으로서 CPU의 과부하로 인한 오동작을 막기 위함이며, 전체적인 시스템의 제어를 하나의 CPU에 맡김으로서 독립적으로 시스템의 제어를 수행하여 안정적인 시스템 구축에 목적이 있다.

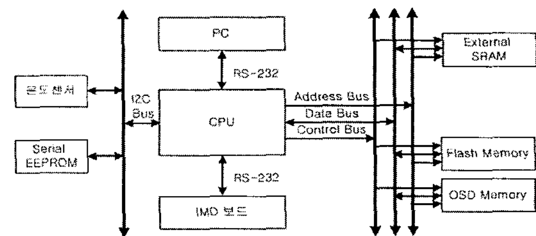


그림 2. HST 보드의 구조
Fig 2. Architecture of HST board

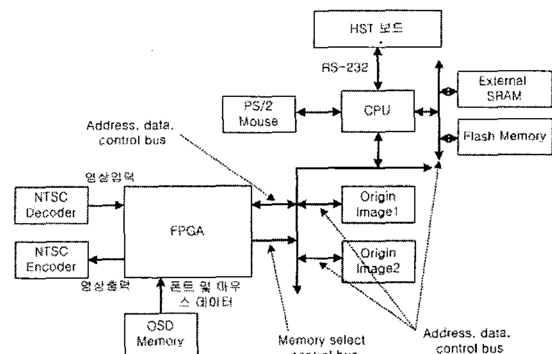


그림 3. IMD 보드의 구조
Fig 3. Architecture of IMD board

그림 2에서 두 보드간의 통신은 각 CPU간의 RS-232 통신을 이용하는 경로와 HST 보드의 CPU와 IMD보드의 FPGA의 Data Bus와 Control Bus를 통한 경로를 이용하여 통신이 가능하다.

외부와의 인터페이스는 HST 보드의 RS-232를 이용한 PC와의 통신 경로와 IMD 보드의 마우스를 이용한 인터페이스가 있다.

AT91R40008 마이크로컨트롤러는 리맵 명령을 지원하다. 리맵 전과 후의 가장 큰 차이점은 메모리의 0x0번지의 변화이다. AT91R40008은 리셋 후 처음 실행번지는 0x0번지의 변화이다. AT91R40008은 리셋후 처음 실행번지는 0x0번지이다. 따라서 0x0번지에는 CPU가 부팅을 할 수 있는 코드가 들어있어야만 한다. 대부분의 경우 0x0번지에 오는 외부 메모리는 플래시 메모리 같은 비휘발성 메모리가 위치하게 된다. 일단 부팅을 하고 나서는 플래시 메모리는 SRAM에 비해 속도가 느리므로 계속해서 플래시 메모리에서 코드를 실행하는 것은 전체적인 시스템의 성능을 저하시키는 결과를 초래하게 된다. 따라서 그림 4에서 보는 바와 같이 리맵 명령을 통해서 0x0번지를 내부 SRAM으로 바꾸어 주는 과정을 통해서 시스템의 전체적인 성능을 높일 수가 있게 된다.

① 리맵 전

그림 4에서 리셋 이후부터 리맵 명령어 전까지의 메모리 구조는 NCS0에 연결된 메모리가 주소공간의 최하단을 차지한다. ARM CPU의 리셋벡터는 0x00번지이므로 NCS0에는 부 팅에 필요한 메모리를 연결해야 하므로 대부분 플래시 메모리가 자리하게 된다. 그리고 wait state를 8로 설정하여 야만 한다. 그리고 내부의 SRAM은 0x300000번지부터 256 KB 의 공간을 가지고 있다.

② 리맵 후

리맵 명령은 EBI_RCR이라는 레지스터에 '1'을 써줌으로써 이루어진다. 이 명령은 리셋 이후의 메모리 구조에서 NCS0이 0x00번지로 되어 있는 것을 내부 SRAM에 0x00 번지를 재배치시킴으로써, 부팅하는 메모리와 실제 코드를 수행하는 부분을 분리시켜 더욱 빠른 수행속도를 가지게 하기 위함이다.[5]

본 시스템에서는 플래시 메모리의 코드는 1KB의 부트 코드이며, 이 부트 코드의 역할은 포팅된 uC/OS-II

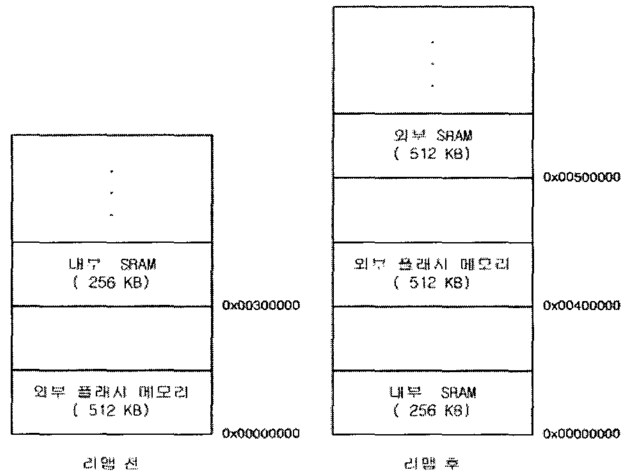


그림 4. 리맵전 , 후의 메모리 구조
Fig 4. Changing Memory Architecture After Remap Command

의 바이너리 이미지를 내부 SRAM으로 옮기고 나서 리맵 명령 과 함께, PC에 0x00을 넣어주는 것으로 끝이 난다. 이러한 동작으로 인하여 우리가 원하는 운영체제의 동작은 플래시에서가 아니라, 내부 SRAM에서 동작을 하게 되므로 훨씬 빠른 동작을 보이게 된다

영상신호의 입력은 NTSC Decoder에 의해서 이루어진다. 입력된 영상 신호는 NTSC Decoder를 통해 나가는 동시에 CPU에 의해 영상처리 과정을 거치기 위해서 Image Memory에 저장이 되어야 한다. 입력되는 영상 신호는 FPGA는 입력이 시작되는 순간부터 매 클록마다 어드레스를 생성하며 입력된 데이터를 메모리에 저장한다. 하나의 영상을 처리하기 위해서는 320*240 byte의 정보를 처리해야만 한다. 이러한 데이터가 초당 60프레임이 입력되기 때문에 1초당 처리해야 되는 데이터의 양은 240*60 = 4500Kbyte 이러한 데이터를 모두 실시간에 처리한다는 것은 위험부담이 있으므로 CPU의 처리 시간을 좀 더 안정적으로 가져가기 위해서 이미지 저장 메모리를 2개 설치하게 되었다. 따라서 1번 메모리에 FPGA에서 영상 신호를 저장하고 있는 동안 CPU는

2번 메모리에 저장되어 있는 영상신호를 처리하고, 반대로 FPGA가 2번 메모리에 영상신호를 저장하고 있는 동안에 CPU는 1번 메모리에 저장되어 있는 데이터를 처리하게 되면 보다 안정적으로 시스템을 구축할 수가 있게 된다.

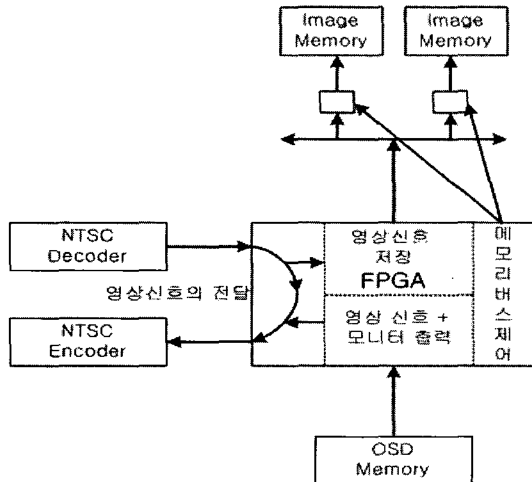


그림 5. FPGA 기능도
Fig 5. FPGA Function Diagram

본 시스템에서는 그림 5에서처럼 FPGA를 이용하여 출력 영상 신호를 적절히 가공하여 모니터를 통해 사용자와 통신할 수 있는 수단을 제공한다. 모니터에 글이나 그림을 인위적으로 표시하기 위해서는 각 글이나 그림의 좌표를 미리 파악하여 그 위치에 점을 찍음으로서 표시를 하게 된다.

본 시스템에서는 OSD 메모리를 따로 가지고 있다. OSD 메모리의 각 주소는 모니터의 각 좌표에 미리 매핑이 되어 있으므로 적당한 OSD 메모리의 주소에 0xff 값을 집어 넣음으로서 OSD 메모리의 내용을 영상신호로 출력을 하면 모니터에는 그 좌표에 하얀 점으로 나타나게 된다.

이 때 FPGA 영상 출력 신호에 OSD 메모리의 내용을 OR 연산을 통해서 중첩 시키는 역할을 함으로서 모니터에 각종 정보를 표시하게 된다.

Ⅲ. uC/OS-II 포팅

ARM7은 6가지의 예외처리가 가능하도록 설계되어 있다. 이들은 각각 Reset, Data Abort, Fetch Abort, Normal Interrupt, Fast Interrupt, Software Interrupt로 구성이 되어 있으며, 이러한 예외상황이 발생했을 때는 이미 정해져 있는 메모리 주소 공간에 있는 코드를 실행하도록 설계되어져 있다.[1] 표2에 예외벡터를 나타내었다.

표 2. Exception Vector
Table 2. Exception Vector

Address	Exception	Mode of Entry
0000.0000	Reset	Supervisor
0000.0004	Undefined Instruction	Undefined
0000.0008	Software Interrupt	Supervisor
0000.000C	Prefetch Abort	Abort
0000.0010	Data Abort	Abort
0000.0014	Reserved	
0000.0018	IRQ	IRQ
0000.001C	FIQ	FIQ

이것은 현재 pc에 있는 값이 0x18 이나 0x1C 이기 때문에 연산을 수행하게 되면, pc는 0xFFFFF100 번지나 0xFFFFF104 번지로 설정이 된다. 이 주소는 각각 AIC_IVR과 AIC_FVR 레지스터의 번지로 인터럽트의 종류에 따라서 각각의 인터럽트 핸들러가 저장되어 있는 주소(AIC_SVR)로 분기할 수 있게 해준다.[5]

CPU에서 C 언어로 작성한 코드가 동작하기 위해서는 함수의 호출과 함수 내부에서 사용하는 변수들을 저장할 수 있는 공간을 확보하는 것과 C 언어에서 사용하는 외부 변수들에 대한 초기화 과정이다.

C 언어에서는 함수 호출과 함수 내부에서 사용하는 변수들을 저장하는 공간으로 스택을 사용한다. 스택에 저장되는 내용들은 함수 호출시 되돌아올 주소와 각 함수에서 내부 변수로 잡았던 영역들이다. ARM CPU에서는 6개의 동작 모드를 지원하므로 각각의 모드에서도 C 언어를 사용하기 위해서는 각각의 모드에 대한 스택을 모두 만들어 주어야만 한다.

본 시스템에서는 CPU의 Supervisor 모드와, IRQ 모드만을 사용하므로 이 두 가지 모드에 대한 스택을 잡도록 한다.

uC/OS-II의 포팅을 위해서 필요한 하드웨어의 최소한의 조건은 10ms의 타이머 인터럽트이다. 운영체제는 이 타이머 인터럽트를 이용하여 스케줄러를 작동시킬 수 있다.

타이머 인터럽트 핸들러에서의 역할은 가장 빠른 시간 내에 인터럽트 모드를 빠져나와서 'OSTickISR' 함수를 호출하는 것이다. 표 3에 타이머 인터럽트 핸들러 코드를 나타내었다.

- 레지스터들을 저장
- OSIntEnter() 함수를 호출하거나, OSIntNesting 변수를 증가
- OSTimeTick()함수를 호출
- OSIntExit()함수를 호출
- 레지스터 복구
- 인터럽트 종료[7]

uC/OS-II에서는 컨텍스트 스위칭을 위하여 OSStartHighRdy(), OSCtxSw(), 그리고 OSIntCtxSw() 3가지의 함수를 사용한다. 이러한 함수들은 인라인 어셈블러를 이용하여 C 함수로 만들거나, 아니면 어셈블러로 직접 만들어야 한다. 표 4에 OSStartHighRdy() 함수 어셈블러 코드를 나타내었다.

OSStartHighRdy() 함수는 OSStart()함수 내부 에서 호출이 된다. 이 함수는 생성된 태스크를 uC/OS-II가 관리할 수 있도록 해주는 멀티태스킹 프로세스를 시작하는데 사용된다.

OSStartHighRdy() 함수가 실행되는 시점은 현재 아무런 태스크가 실행되고 있지 않은 상태에서의 첫 번째 태스크를 수행하는 시점이다. 따라서 현재 상태의 저장 과정이 생략된 컨텍스트 스위칭의 동작을 하게 된다.

- 사용자 정의 가능한 OSTaskWHook() 함수의 호출
- 복원할 태스크의 스택 포인터 가져오기
- OSRunnig = TRUE
- 새로운 태스크의 스택으로부터 CPU의 모든 레지 스터를 복원
- 함수에서 복귀[7]

표 3. 타이머 인터럽트 핸들러 코드
Table 3. Timer Interrupt handler source code

```
TimerIRQ
        stmfd   sp!,{r0-r3}
        ldr             r2, =0xfffe0000         ;Timer interrupt clear
        ldr             r2, [r2, #0x20]
        mov             r2, sp          ;copy IRQ's Stak pointer
        add             sp,sp,#16       ;recover IRQ's stack pointer
        sub             r3, lr, #4      ;copy return address ->r3

        ldr             r0, =0xfffff000         ;AIC Base Address
        str             r0, [r0, #0x130]        ;End of Interrupt

        ldr             r0, =IRQ_2
        movs    pc, r0                          ;IRQ mode->SVC mode
;--------Stack operation for context switching-------------
IRQ_2
        stmfd   sp!,{r3}
        stmfd   sp!,{r4-r12, lr}
        mov             r4, r2
        ldmfd   r4!,{r0-r3}
        stmfd   sp!,{r0-r3}
        mrs             r5, cpsr
        stmfd   sp!,{r5}
        b               OSTickISR
;--------------------------------------------------------
OSTickISR
        LDR     r0,=OSIntNesting        ; Notify uC/OS-II of ISR
        LDRB    r1,[r0]
        ADD     r1,r1,#1
        STRB    r1,[r0]
        BL      OSTimeTick              ; Process system tick
        BL      OSIntExit               ; Notify uC/OS-II of end of ISR
        LDMFD sp!,{r0} .
        MSR     CPSR_xsf,r0
        LDMFD   sp!,{r0 - r12, lr , pc}
```

표 4. OSStartHighRdy의 어셈블러 코드
Table 4. OSStartHighRdy Assembly Code

```
OSStartHighRdy
    BL        OSTaskSwHook
    LDR        r0, =OSRunning
    MOV        r1, #1
    STRB r1, [r0]
    LDR        r0, =OSTCBHighRdy
    LDR        r0, [r0]
    LDR        sp, [r0]
    LDMFD sp!; {r0}
    MSR        CPSR_xsf, r0
    LDMFD sp!,{r0-r12,lr, pc}
```

스케쥴링에는 태스크 레벨의 스케쥴링과 ISR 레벨의 스케쥴링이 있다. 태스크 레벨의 스케쥴링은 OSSched() 함수를 실행시킴으로서 이루어지는데, 이 함수 내에서 컨텍스트 스위칭을 시켜주는 것이 OSCtxSw() 함수이다. 따라서 OSCtxSw() 함수의 역할은 표 5의 OSCtxSw 코드에서 보는 것처럼 현재 태스크의 레지스터 내용을 스택에 저장하고 새

로운 태스크의 정보를 레지스터에 복원시키는 작업을 하게 된다.

- 현재 작업 레지스터 저장
- OS_TCB에 현재 태스크의 스택 포인터 저장
- 사용자 정의 가능한 OSTaskSwHook() 함수 호출
- OSTCBCur = OSTCBHighRdy
- OSPrioCur = OsPrioHighRdy
- 복원할 태스크의 스택 포인터 가져오기
- 새로운 태스크의 스택으로부터 모든 레지스터 복원
- 함수에서 복귀[7]

이 함수는 OSIntExit() 함수에 의해 호출된다. 이 함수는 항상 타이머 인터럽트가 발생할 때마다 호출이 되며, OSIntExit() 함수는 호출된 시점에서 태스크 스위칭이 필요한 경우에 OSIntCtxSw() 함수를 호출하여 컨텍스트 스위칭을 하게 된다. 따라서 OSIntCtxSw() 함수는 OSCtxSw() 함수와는 다르게 이미 그 전의 상태가 저장이 되어 있으므로 현재 태스크에 대한 레지스터 저장 등의 동작이 필요가 없게 된다. 표 6에 OSIntCtxSw 어셈블러 코드를 나타내었다.

표 5. OSCtxSw의 어셈블러 코드
Table 5. OSCtxSw Assembly Code

```
OSCtxSw
        STMFD sp!,{lr}
        SEMFD sp!,{ro-r12,lr}
        MRS    ro, CPSR
        STMFD sp!,{r0}
        LDR    r0,=OSTCBCur
        LDR    r0,{r0}
        STR    ro,{r0}
        BL     OSTaskSwHook
        LDR    r0,=OSTCBCur
        LDR    r1,OSTCBHighRdy
        LDR    r2,[r1]
        STR    r2,[r0]
        LDR    r0,=OSPrioCur
        LDR    r1,=OSPrioHighRdy
        LDRB   r3,[r1]
        STRB   r3,[r0]
        LDR    sp,{r2}
        LDMFD sp!{r0}
        MSR    CPSR_xsf, r0
        LDMFD sp!,{ro-r12,pc}
```

- OSIntExit() 함수, OSIntCtxSw() 함수의 호출등에 의해 스택 포인터를 지우면서 정리
- OS_TCB에 현재 태스크의 스택 포인터 저장
- 사용자 정의 가능한 OSTaskSwHook() 함수 호출
- OSTCBCur = OSTCBHighRdy
- OSPrioCur = OSPrioHighRdy
- 복원할 태스크의 스택 포인터 가져오기
- 새로운 태스크의 스택으로부터 모든 레지스터 복원
- 함수에의 복귀[7]

표 6. OSIntCtxSw의 어셈블러 코드
Table 6. OSIntCtxSw Assembly Code

```
OSIntCtxSw
        ADD    sp, sp,#4
        LDR    r0,=OSTCBCur
        LDR    ro,[r0]
        STR    sp,[r0]
        BL     OSTaskSwaHook
        LDR    ro,=OSTCBCur
        LDR    r1,=OSTCBHighRdy
        LDR    r2,[r1]
        STR    r2,[r0]
        LDR    r0,=OSPrioCur
        LDR    r1,=OSPrioHighRdy
        LDRB   r3,[r1]
        STRB   r3,[r0]
        LDR    sp,[r2]
        LDMFD sp!,{r0}
        MSR    CPSR_xsf, r0
        LDFD   sp!,{r0~r12,lr, pc}
```

VI. 결 론

본 논문에서는 높은 성능을 얻기 위하여 하나의 CPU에 영상 처리 알고리즘 연산을 전담 시켰으며 영상 디코더로부터의 신호를 FPGA를 통해서 따로 확보해 둔 영상 데이터 메모리에 저장을 하고 CPU는 영상 디코더부터의 신호가 아닌 메모리의 데이터를 처리하게 된다. 영상 처리 CPU는 FPGA가 접근하지 않는 영상 데이터 메모리에 접근하여 처리를 하게 됨으로써, 영상 데이터 처리에 있어서 보다 안정적인 동작을 보장 받을 수 있게 된다. 또한 더욱 복잡한 영상 처리 알고리즘을 필요로 할 경우에는 FPGA를 통하여 그 영상 테이터의 전처리 알고리즘을 수행할 수 있으므로 시스템의 성능을 또 다시 비약적으로 발전시킬 수 있는 여지를 가질 수 있으므로 임베디드 환경에서 보다 안정적인 동작을 보장 받을 수 있으며 또한 시스템에 대한 업그레이드가 FPGA의 코드 변경을 통해 이루어 질 수 있으므로 근본적인 하드웨어의 수정 없이도 좀 더 높은 사양

의 시스템으로 업그레이드를 시킬 수 있는 장점이 있다.

참 고 문 헌

[1] ARM, "ARM Architecture Reference Manual". February 2000

[2] ARM, "ARM7TDMI Data Sheet". Aug 1995

[3] Jean, J. Labrosse, "Micro C/OS-II The Real-Time Kernel", R&D Books, 1999

[4] Ed Sutter, "Embedded Systems Firmware Demystified", CMP Books, 2002

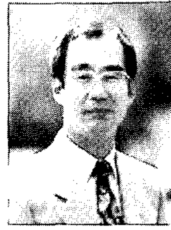
[5] ATMEL, " AT91 ARM Thumb Micro controllers Data Sheet ", 2002

[6] 김효준. "uC/OS-II EP7209(ARM7) 포팅" 2001

저 자 소 개

김 병 철(학생회원)
2001년 아주대학교 공과대학 전자공학전공(학사), 2003년 아주대학교 대학원 전자공학전공 (석사), ~ 현재 LG전자 DAV 사업부

하 동 문(학생회원)
1997년 아주대학교 공과대학 전자공학전공(학사), 2001년 아주대학교 대학원 전자공학전공 (석박사 통합과정 수료)



김 용 득(정회원)
1971년 연세대학교 전자공학전공 (학사), 1973년 연세대학교 전자공학전공 (석사), 1978년 연세대학교 전자공학전공 (박사), 1979년~현재 아주대학교 전자공학부 정교수 , 1973년~1974년 불란서 E. S. E 전자공학 연구실, 1973년~1974년 미국 STANFORD 대학교 연구교수, 1981년~1982년 한국전자통신연구소 위촉연구위원, 1994년~1998년 ITS 연구기획단 연구 위원, 전자부문 총괄