논문 2023-1-8 http://dx.doi.org/10.29056/jsav.2023.3.08

임베디드 소프트웨어 구성용 C-언어의 효과적인 임무 분기 방법에 관한 고찰

강명훈*, 이현창**†

A Study on the Effective Task Branching Method of C-Language for Embedded Software Configuration

Myeong-Hoon Kang*, Hyun-Chang Lee**†

요 약

본 논문에서는 임베디드 시스템에서 사용되는 C-언어의 효과적인 분기 방법을 제시하였다. 이를 위해 단순 if문에 의한 분기, 트리 구조의 if문에 의한 분기, switch문에 의한 분기, 함수 포인터 배열에 의한 분기 및 look-up table에 의한 함수 포인터 분기들의 이론적인 분석을 진행하였다. 분석결과 2, 3개의 분기 시는 if문이 가장 유리하지만, 약 32개 이내의 분기에서는 트리 구조의 if문이 유리하고, 그 이상에서는 함수 포인터 배열이 유리함을 확인하였다. 특히 휘발성 메모리가 매우 작은 원-칩 마이크로컨트롤러의 경우에 있어서는 look-up table에 의한 함수 포인터 분기가 가장 유리하였다. 이러한 분석결과를 확인하기 위해 마이크로컨트롤러를 대상으로 실험을 진행한 결과 이론적 예상과 거의 유사한 결과를 확인하였다.

Abstract

In this paper, an effective branching method of C-language used in embedded systems was presented. For this purpose, a theoretical analysis of branching by simple if statement, branching by if statement of tree structure, branching by switch statement, branching by function pointer array, and function pointer branching by look-up table was conducted. As a result of the analysis, it was confirmed that the if statement is most advantageous for 2 or 3 branches, but the if statement in the tree structure is advantageous for branches within about 32 branches, and the function pointer array is advantageous for more than that. In particular, in the case of one-chip microcontrollers with very small volatile memory, function pointer branching by look-up table was most effective. In order to confirm these analysis results, an experiment was conducted on a microcontroller, and results almost similar to the theoretical expectations were confirmed.

한글키워드 : 임베디드, 소프트웨어, 분기, 함수 포인터, 조견표

keywords : embedded, software, branch, function pointer, look-up table

* 연암대학교 통합전산센터장
** 공주대학교 정보통신공학과
† 교신저자: 이현창(email: hclee@kongju.ac.kr)
접수일자: 2023.03.08. 심사완료: 2023.03.14.
게재확정: 2023.03.20.

1. 서 론

최근에 유행하는 IoT(Internet of Things)[1]를 비롯한 드론(drone)[2]이나 각종 전자제품들은 기

기의 소형화 및 고성능화를 위해 마이크로컨트롤러를 사용하고[3][4], 마이크로컨트롤러에 탑재할 소프트웨어의 구성은 간편성과 세부 제어가 가능한 C-언어를 주로 사용한다.[5][6] 그런데 각종 시스템에 부속으로 사용되는 마이크로컨트롤러는 메인 프로세서의 커맨드에 의한 동작이나 외부의 각종 이벤트들에 의한 동작이 많은데, 이러한 커맨드나 이벤트의 종류에 따라 해당 서비스를 진행하기 위해 각 루틴으로 분기를 해야 한다. 이 때 분기의 수가 많아지면 해당 서비스를 진행하기 위한 시간보다 분기 때 시간이 더 소모될 수 있으며, 또한 각 서비스 루틴에 도달하는 시간들이 프로그램의 구성 형태에 따라 차이가 발생할 수 있다. 특히 극히 제한적인 휘발성 메모리 용량을 가지는 마이크로컨트롤러의 경우 휘발성 메모리의 사용 또한 극소화해야 하는 제약점이 있다.

임베디드 시스템은 시스템 상황에 따라 요구조건이 다르지만 일반적으로 다음과 같은 특성이 요구된다.

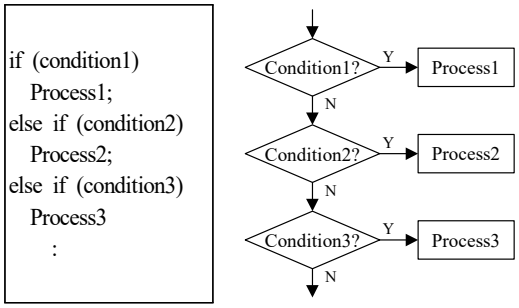
첫째, 커맨드 또는 이벤트에 따른 프로세스 도달시간이 일정해야 한다. 둘째, 실시간 시스템의 경우 커맨드 또는 이벤트 프로세스에 도달하는 시간이 짧아야 한다. 셋째, 시스템의 안정성 측면에서 유리하도록 최대한 저속으로 동작한다. 넷째, 휘발성 메모리 영역은 매우 작고, 비휘발성 메모리 영역은 비교적 큰 용량으로 구성된다.

본 논문에서는 C-언어에서 사용되는 다양한 분기 방법들에 관해 분석을 진행함으로써, 마이크로컨트롤러를 구성하는데 지침이 되는 데이터를 제공하고자 한다.

2. 분기 명령의 구성 분석

2.1 다중 if문에 의한 분기

분기에 가장 많이 사용되는 if문은 이 문장이 나타날 때 마다 비교 명령을 수행한다. 따라서 한 변수의 내용에 따라 많은 분기를 진행해야 할 경우 그림 1과 같은 구조를 형성한다.



(a) C-언어 원문 (b) 진행 과정

그림 1. 단순 if문 배열 구조

Fig. 1. If statement array structure

그림 1은 일렬로 if문을 배열해 조건별로 분기한 것으로서, 이러한 구성 방법은 각 조건마다 프로세스에 도달하는 시간이 달라지고, 특히 가장 마지막에 배치된 프로세스의 경우 첫 번째 배치된 프로세스에 비해 도달 시간에 매우 큰 차이가 발생한다. 프로세스에 도달하는데 소요되는 시간은 식 (1)과 같이 나타낼 수 있다.

$$T_n = T_{CMP} \cdot (n-1) \quad (1)$$

여기서, T_n은 n번째 프로세스에 도달하는 시간, T_{CMP}는 비교 명령 실행시간이다.

식 (1)에 의하면 만약 256 종류의 프로세스로 분기해야 하는 경우 첫 프로세스와 마지막 프로세스는 255배의 도달시간 차이가 발생한다.

2.2 트리 구조 if문에 의한 분기

프로세스가 여러 개인 경우 그림 2에 나타낸 바와 같이 트리 구조의 if문을 형성해 분기하면 프로세스에 도달하는 시간을 크게 줄일 수 있다[7][8].

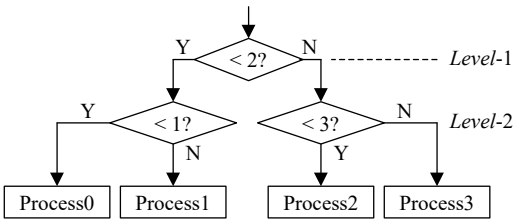


그림 2. 트리 구조 if문
Fig. 2. Tree structured if statement

이는 비교 레벨이 1 증가할 때 마다 분기 프로세스가 2배씩 증가하므로, 프로세스 수 n은 식 (2)와 같이 표시할 수 있다.

$$n = 2^L \qquad (2)$$

따라서 식 (2)로 부터 비교 레벨 수 L은 식 (3)과 같이 구할 수 있다.

$$L = \log_2 n \qquad (3)$$

트리 구조의 if문을 형성하면 예를 들어 256종류의 프로세스를 분기할 때 식 (3)에 의해 8번의 비교만으로 분기가 가능하다. 특히 앞서 고찰한 다중 if문에 비하면 각 프로세스들의 위치에 관계없이 모두 거의 유사한 지연시간을 가진다.

2.3 Switch문에 의한 분기

여러 경우로 분기할 때 자주 사용되는 switch문의 경우 이를 컴파일 하는 컴파일러에 따라 구조가 상당히 다를 수 있다. 그림 3에 임베디드 시스템을 구성할 때 자주 사용되는 gcc로 컴파일 했을 때의 어셈블 결과를 나타내었다.

그림 3(b)의 컴파일 결과에서, r24(24번 레지스터)에 판단할 숫자가 보관되고, 이를 cpi(compare immediate) 명령에 의해 판단해 breq(branch equal to zero) 명령으로 분기한다.

그러나 각 case문마다 정확히 일치하는 breq가 아니라, 1be4 라인과 같이 중간에 brcc (branch if carry clear) 명령에 의해 분기하는데, 이는 숫자가 정확히 일치하는 것이 아니라 큰지 작은지에 따라 분기해 앞서 고찰한 트리구조와 유사한 구조를 형성하고 있음을 알 수 있다.

```
switch ( addr )
    {
    case 0 : cmd_G0C00(); break;
    case 1 : cmd_G0C01(); break;
     :              :       :
    case 6 : cmd_G0C06(); break;
    case 7 : cmd_G0C07(); break;
    }
```

(a) C-언어 원문

```
;      switch ( addr )
1bde: 83 30        cpi   r24, 0x03      ; 3
1be0: b1 f0        breq  .+44           ; 0x1c0e
1be2: 84 30        cpi   r24, 0x04      ; 4
1be4: 28 f4        brcc  .+10           ; 0x1bf0
1be6: 81 30        cpi   r24, 0x01      ; 1
  :    :            :     :
1bf8: 86 30        cpi   r24, 0x06      ; 6
1bfa: 79 f0        breq  .+30           ; 0x1c1a
1bfc: 87 30        cpi   r24, 0x07      ; 7
1bfe: 81 f4        brne  .+32           ; 0x1c20
1c00: 0e c0        rjmp  .+28           ; 0x1c1e
```

(b) 컴파일 결과

그림 3. Switch문의 컴파일 결과
Fig. 3. Compile result of switch statement

따라서 switch문을 이용하는 경우 일렬로 구성되는 if문에 비하면 더 빠른 시간에 프로세스에 도달할 수 있지만, 컴파일 알고리즘에 따라 프로그래머가 인위적으로 구성한 완전한 트리구조보다는 다소 느리거나 또는 각 프로세스에 도달하는 시간에 차이가 발생할 수 있어 각 프로세스마다 도달할 시간을 정확히 예측하기 어렵다.

2.4 함수 포인터 배열에 의한 분기

각 커맨드나 이벤트에 따라 처리되는 함수들의 포인터를 배열변수에 기록하고, 커맨드나 이

벤트 코드에 따라 직접 해당 함수를 호출하면 매우 짧은 시간에, 또한 모든 프로세스에서 동일한 시간에 도달할 수 있다. 그림 4에 함수 포인터 배열을 이용한 실행 예를 나타내었다.

```
void (*cmd_pnt[8])( void ); /* Define Var. array */

/* Initialize Vector Array */
void Set_Vector_Array( void )
  {
    cmd_pnt[0] = cmd_000; cmd_pnt[1] = cmd_001;
         :                          :
    cmd_pnt[6] = cmd_006; cmd_pnt[7] = cmd_007;
  }

void main( uint8_t command )
  {
   Set_Vector_Array( ); /* Initialize Vector */
   cmd_pnt[ command ]( ); /* Execute */
  }
```

(a) C-언어 원문

```
; cmd_pnt[0]=cmd_000; cmd_pnt[1]=cmd_001;
214: 82 e7          ldi   r24, 0x72      ; cmd_000
216: 9e e0          ldi   r25, 0x0E
218: 90 93 4c 03    sts   0x034C, r25    ; To SRAM
21c: 80 93 4b 03    sts   0x034B, r24
220: 8c e5          ldi   r24, 0x5C      ; cmd_001
222: 9e e0          ldi   r25, 0x0E
224: 90 93 4e 03    sts   0x034E, r25
228: 80 93 4d 03    sts   0x034D, r24    ; To SRAM
 :    :              :      :
; cmd_pnt[ command ](); /* Execute */
dc2: f0 e0          ldi   r31, 0x00      ; r30 - command
dc4: ee 0f          add   r30, r30       ; Calc. addr.
dc6: ff 1f          adc   r31, r31
dc8: e9 53          subi  r30, 0x39
dca: fd 4f          sbci  r31, 0xFD
dcc: 01 90          ld    r0, Z+         ; Fetch vector
dce: f0 81          ld    r31, Z
dd0: e0 2d          mov r30, r0
dd2: 09 95          icall                ; Indirect Jump
```

(b) 컴파일 결과

그림 4. 함수 포인터 배열의 컴파일 결과
Fig. 4. Compile result of array of function pointers

그림 4(a)의 리스트에서, "void (*cmd_pnt[8])(void);" 문장은 함수 포인터형으로 cmd_pnt라는 변수 배열을 8개 생성하고, "Set_Vector_Array( )" 함수에서 각 배열변수에 처리할 함수 포인터들을 대입하고. 'cmd_pnt[command]( )' 명령어 1개로 해당 함수를 직접 호출한다.

이러한 C-언어 문장을 컴파일 한 경우 그림 4(b)와 같은 어셈블 리스트를 얻을 수 있는데, 이 리스트의 상반부는 배열변수에 각 함수 포인터들을 대입하는 준비과정이고, 하반부는 주어진 커맨드에 해당하는 함수 포인터를 추출해 최종적으로 icall (Indirect call) 명령에 의해 간접 호출한다. 이와 같은 구조의 프로그램은 프로세스의 수나 배치 순서에 관계없이 모두 동일한 지연시간을 가지므로 비교적 정확한 예측이 가능하다.

그러나 이 방법은 그림 4(b)의 상반부와 같이 배열변수로서 휘발성 메모리(SRAM)를 이용하는데, 주 메모리가 모두 휘발성 메모리로 구성되는 서버나 PC급에서는 큰 문제없이 사용할 수 있지만, 휘발성 메모리가 매우 작은 임베디드 시스템용 마이크로컨트롤러에는 적용하기에 다소 무리가 있다.

예를 들어 어떤 마이크로컨트롤러의 휘발성 메모리가 2Kbyte이고, 커맨드나 이벤트 코드의 종류가 256개인 경우 2byte의 포인터 주소를 가지면 2K 용량의 휘발성 메모리 중 25%에 해당하는 512byte를 배열변수용으로 사용하기 때문에 다른 변수나 스택 영역 등의 자유도가 크게 떨어지는 문제점이 발생할 수 있다.

2.5 PROGMEM 옵션의 활용

함수의 포인터들은 사실상 지속적으로 변화하는 숫자가 아니라 펌웨어로 구성되면 그 위치가 고정되는 일종의 상수에 해당하므로 이를 굳이 휘발성 메모리에 저장하지 않고 펌웨어가 저장되는 비휘발성 영역에 상수로 저장할 수 있다.

그림 5에 이와 같은 방법으로 구성한 프로그램의 예를 나타내었다.

```
void PROGMEM (*G0_pnt[8])( void ) =
  {cmd_G0C00, cmd_G0C01, cmd_G0C02, cmd_G0C03,
cmd_G0C04, cmd_G0C05, cmd_G0C06, cmd_G0C07};

void main( uint8_t command )
   { G0_pnt[ command ]( ); /* Execute */ }
```

(a) C-언어 원문

```
000000ac <G0_pnt>:
ac:  49 0a 47 0a 45 0a 3d 0a 35 0a 2d 0a 1e 0a 1c 0a

;   G0_pnt[addr]();
1c3e: f0 e0    ldi     r31, 0x00  ; 0
1c40: ee 0f    add     r30, r30
1c42: ff 1f    adc     r31, r31   ; Calc. addr.
1c44: e4 55    subi    r30, 0x54  ; Base addr(0xAC)
1c46: ff 4f    sbci    r31, 0xFF  ; 00
1c48: 01 90    ld      r0, Z+     ; Fetch vector
1c4a: f0 81    ld      r31, Z
1c4c: e0 2d    mov     r30, r0
1c4e: 09 95    icall              ; Indirect Jump
```

(b) 컴파일 결과

그림 5. PROGMEM을 활용한 컴파일 결과
Fig. 5. Compile result using the PROGMEM

그림 5(a)에 나타낸 바와 같이 배열변수를 선언할 때 이 배열변수를 비휘발성 메모리에 고정 배치하라는 의미로 PROGMEM 옵션을 사용한 경우, 그림 5(b)와 같이 컴파일된다.

컴파일 결과에서 프로그램 영역의 ac번지서부터 2바이트씩 각 함수의 위치가 저장되고, 이 함수를 실행하는 부분은 1c3e 이후와 같이 컴파일되어 일종의 look-up table로 함수의 주소를 추출해 이를 간접 호출한다.

이러한 구성 방법은 앞 절의 함수 포인터 배열을 이용한 것과 프로그램의 실행 사이클 수는 동일하지만 함수 포인터를 저장하기 위해 휘발성 메모리를 사용하지 않으므로 임베디드 시스템에서 더욱 유리하다.

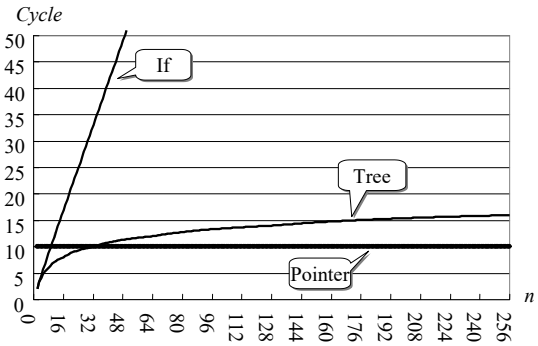
2.6 이론적 분석 결과

이상에서 살펴본 다양한 분기방법들의 효과를 분석하기 위해 시뮬레이션을 진행하여 그림 6과 같은 결과를 얻었다.

n	If	Tree	Pointer
2	2	2	10
:	:	:	:
8	8.75	6	10
:	:	:	:
16	16.88	8	10
:	:	:	:
32	32.94	10	10
:	:	:	:

n	If	Tree	Pointer
64	64.97	12	10
:	:	:	:
128	128.98	14	10
:	:	:	:
171	171.99	15	10
:	:	:	:
256	256.99	16	10
:	:	:	:

(a) 분기 시간 (평균)



(b) 시뮬레이션 그래프

그림 6. 시뮬레이션 결과
Fig. 6. Simulation result

그림 6(a)는 분기할 가지 수(*n*)에 도달하기 위한 평균 사이클 수를 나타낸 것으로서, 프로세서가 RISC 구조인 경우 일반적으로 한 명령 당 한 사이클이 소요되므로 비교 명령(compare) 1사이클과 분기 명령(branch) 1사이클로 분기 시 2사이클이 소요된다고 가정한 것이다. 그림 6(b)는 그림 6(a)를 그래프로 나타낸 것으로서, 그래프에서 가로축은 분기 수를, 세로축은 이러한 분기를 진행하기 위한 평균 사이클 수이다.

이 결과에 따르면 일렬로 구성된 if문의 경우 분기 수가 늘어나면 그에 따라 분기 실행 사이클이 비례적으로 늘어나는데 비해, 트리 구조의 if

문인 경우 로그적으로 증가한다. 함수 포인터 배열을 이용한 경우에는 분기 수에 관계없이 항상 동일한 사이클을 유지한다. 그림 6(a)의 결과에 의하면 2종류의 분기에서는 if문이 단연 유리하고, 32종류 이하의 분기에서는 트리구조의 if문이 유리하며, 이 이상의 경우에는 함수 포인터 배열이 유리하다는 결과를 얻었다. 특히 단순 if문의 경우 256종류 분기에서는 가장 빠른 경우가 2사이클, 가장 느린 경우가 510사이클로서 매우 큰 차이가 발생한다.

3. 실험 및 고찰

이상에서 고찰한 사항을 실험으로 확인하기 위해 AVR GCC[9]를 이용해 그림 7의 프로그램을 구성해 마이크로컨트롤러 ATmega128A[10]에서 실행하였다.

마이크로컨트롤러의 하드웨어 카운터에서는 프로세서의 1개 클록신호마다 증가동작 하도록 설정하였으며, 따라서 이 카운터에는 프로세서의 실제 실행시간이 클록 단위로 저장된다. 해당 if문이나 switch문, 그리고 포인터 등에 의해 분기해 해당 부분에 도달하면 그 부분에서 카운터 값을 읽으면 분기 동작에 소요된 클록 수를 카운트할 수 있다.

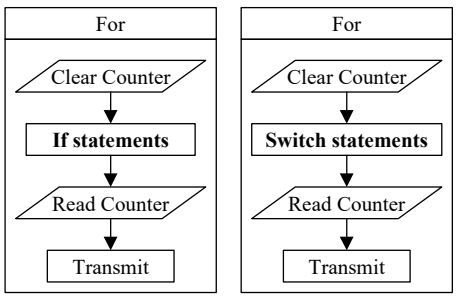


그림 7. 실험용 소프트웨어의 구성
Fig. 7. Configuration of experimental software

이와 같은 방법을 이용해 각 분기방법별로, 또한 각 분기 숫자별로 실험한 결과를 표 1에서 표 4까지에 나타내었고, 그림 8에 이 결과들을 그래프로 나타내었다. 결과표들에서 'Result [CLK]'는 각 분기 단계별로 도달한 클록 수이다.

표 1. 일렬 If문의 결과.
Table 1. Result of serial if statement

n	*Result [CLK]*	*Tot.*	*Avrg.*
2	2,3	5	2.5
3	2,5,6	13	4.33
4	2,5,8,9	24	6
5	2,5,8,11,12	38	7.6
6	2,5,8,11,14,15	55	9.17
7	2,5,8,11,14,17,18	75	10.71
8	2,5,8,11,14,17,20,21	98	12.25
:	:	:	:

표 2. 트리구조 If문의 결과
Table 2. Result of tree structed if statement

n	*Result [CLK]*	*Tot.*	*Avrg.*
2	2,3	5	2.5
3	4,5,3	12	4
4	4,5,5,6	20	5
5	6,7,5,5,6	29	5.8
6	6,7,7,8,5,6	39	6.5
7	6,7,7,8,7,8,6	49	7
:	:	:	:
36		465	12.92
37		481	13
:		:	:

표 3. Switch문의 결과
Table 3. Result of switch statement

n	*Result [CLK]*	*Tot.*	*Avrg.*
2	3,6	9	4.5
3	5,3,8	16	5.33
4	5,3,7,10	25	6.25
5	7,10,3,8,11	39	7.8
6	7,10,3,10,8,13	51	8.5
7	10,7,9,3,10,8,13	60	8.57
:	:	:	:
12	11,15,7,12,16,3,12,...	147	12.25
13	12,16,8,16,13,15,4,...	173	13.31
:	:	:	:

표 4. 함수 포인터 배열의 결과
Table 4. Result of array of function pointers

n	*Result [CLK]*	*Tot.*	*Avrg.*
1	13	13	13
2	13,13	26	13
3	13,13,13	39	13
4	13,13,13,13	52	13
5	13,13,13,13,13	65	13
:	:	:	:

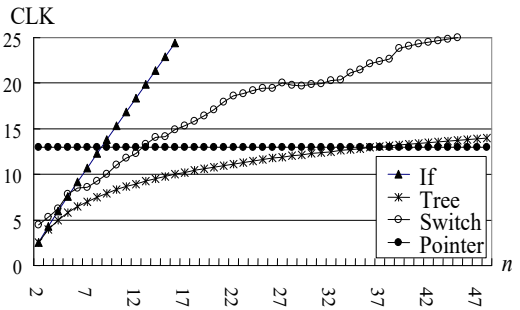


그림 8. 실험 결과
Fig. 8. Experimental result

그림 8의 그래프를 살펴보면, 일렬로 구성된 if문과 트리 구조의 if문, 그리고 포인터 배열에 의한 분기들은 그림 6에서 예상한 바와 거의 유사한 형태로 나타났다. 단, 이론 부분에서 예상한 것은 프로그램의 사이클 수에 의한 것이며, 실재 실험에서는 함수 포인터 배열에 의한 방법에서 해당 함수를 call 하는 과정에 스택에 보관하는 과정 등이 추가되어 13클록이 지연되며, 이에 따라 트리구조 if문과 함수 포인터 방법을 선택하는 기준은 37개 분기점임을 확인할 수 있다.

이에 비해 switch문의 경우에는 트리 구조에 비해 대략적으로 2배 정도의 지연시간이, 특히 부드러운 곡선의 형태가 아닌 것으로 나타났다. 이는 표 3에 의하면 switch문의 컴파일 알고리즘이 모든 case의 중앙 점을 가장 먼저 판단하고, 이를 기준으로 상, 하 쪽으로 판단을 확장하는데, 아무래도 인위적으로 정확하게 구성한 트리 구조에 비하면 완전하지 못한 점이 확인된다. 이와 같이 switch문에 의한 결과는 컴파일러마다 다르기 때문에 이 데이터가 모든 컴파일러와 동일하지 않다는 것, 즉 예측하기 어렵다는 것이다.

4. 결 론

본 논문에서는 임베디드 시스템에서 사용되는 C-언어의 효과적인 분기 방법을 제시하였다. 이를 위해 단순 if문에 의한 분기, 트리 구조의 if문에 의한 분기, switch문에 의한 분기, 함수 포인터 배열에 의한 분기 및 look-up table에 의한 함수 포인터 분기들의 이론적인 분석을 진행하였다. 분석결과 2, 3개의 분기 시는 if문이 가장 유리하지만, 약 32개 이내의 분기에서는 트리 구조의 if문이 유리하고, 그 이상에서는 함수 포인터 배열이 유리함을 확인하였다. 특히 휘발성 메모리가 매우 작은 원-칩 마이크로컨트롤러의 경우에 있어서는 look-up table에 의한 함수 포인터 분기가 가장 유리하였다. 이러한 분석결과를 확인하기 위해 마이크로컨트롤러를 대상으로 실험을 진행한 결과 이론적 예상과 거의 유사한 경향을 보이지만, 실험 결과 실제 프로세서의 실행 과정에 의해 분기 방법의 선택 포인트가 37개임을 확인하였다.

참 고 문 헌

[1] Quangang Wen, Xinzheng Dong, Ronggao Zhang, “Application of dynamic variable cipher security certificate in Internet of Things”, 2012 IEEE 2nd International Conference on Cloud Computing and Intelligence Systems, pp.1062 - 1066, Oct, 2012. DOI: 10.1109/CCIS.2012.6664544

[2] Zhu, Rongjiao, “Drone swarm navigation

system for new media", Thesis - The Hong Kong university of science and technology, 2017.
DOI: 10.14711/thesis-991012555559103412
[3] P Sihombing, T P Astuti, Herriyance, D Sitompul, "Microcontroller based automatic temperature control for oyster mushroom plants", 2nd International Conference on Computing and Applied Informatics, 2017.
DOI : 10.1088/1742-6596/978/1/012031
[4] Yei Wang, Yiling Liu, Li Wang, "Research on Grain Drying Control System based on Microcontroller", World Scientific Research Journal, Vol.6, No.11, pp.368-375, Nov.2020.
DOI : 10.6911/WSRJ.202011_6(11).0051
[5] Heng Wang, Xinrui Chen, "An embedded C language Target Code level Unit Test method base on CPU Simulator", Journal of Physics: Conference Series, Vol.1345, No.5, 2019.
DOI : 10.1088/1742-6596/1345/5/052068
[6] Ling Ji, "Control System Design Based on MSP430 Microcontroller", Advanced Materials Research, Vol.1030-1032, pp.1438-1441, Sep. 2014. DOI:10.4028/www.scientific.net/AMR.1030-1032.1438
[7] Yan-yan Song, Ying Lu, "Dicision tree methods: applications for classification and prediction", Shanghai Archives of Psychiatry, Vol.27, No.2, Apr. 2015.
DOI : 10.11919/j.issn.1002-0829.215044
[8] Yong Zhang, Xiuxing Li, Jin Wang, Ying Zhang, Chunxiao Xing, Xiaojie Yuan, "An Efficient Framework for Exact Set Similarity Search Using Tree Structure Indexes", 2017 IEEE 33rd International Conference on Data Engineering(ICDE), pp.759 - 770, Apr. 2017.
DOI : 10.1109/ICDE.2017.127
[9] Microchip, GCC Compilers for AVR and Arm-Based MCUs and MPUs, May 2022.
https://www.microchip.com/en-us/tools-resources/develop/microchip-studio/gcc-compilers
[10] Microchip, 8-bit AVR Microcontroller ATmega128A Datasheet Complete, 2015.
http://ww1.microchip.com/downloads/en/devicedoc/atmel-8151-8-bit-avr-atmega128a_datasheet.pdf

저 자 소 개

강명훈
(Myeong-Hoon Kang)

1996.2 연암공과대학교 전산학과 졸업
1995.7~2014.2 LG CNS IT 컨설팅외
2020.2 공주대학교 IT융합학과 석사
2022.2~현재 공주대학교 컴퓨터공학과 박사과정
2014.3~현재 연암대학교 통합전산센터장
<주관심분야> 시스템 소프트웨어, 개인정보보호, Database

이현창(Hyun-Chang Lee)

1986.2 단국대 전자공학과 학사
1989.8 단국대 전자공학과 석사
1996.2 단국대 전자공학과 박사
1996.3~2004 국립 천안공업대학 정보통신과 부교수.
2005.3~현재 국립 공주대학교 천안공과대학 정보통신공학과 교수.
<주관심분야> 멀티미디어 회로, 모터제어, 마이크로프로세서, 임베디드 소프트웨어