Application Note 60

Title: **Frequently asked software questions for EM 8-bit Microcontrollers**

Product Family: **CoolRISC core architecture**

Part Number: EM6812, EM9550, EM9551, EM9553, EM4275

Keywords: CoolRISC, constant, initialized global variables, tables, programming, IntelHEX, sections, text, sections

Date: July 28, 2005

Table of contents:

A COMPANY OF THE SWATCH GROUP

1. How to implement constants table

CoolRISC architecture introduction

CoolRISC architecture is based on Harvard RISC.
Instructions are stored in the program memory and data are stored in a separate data memory. For some existing EM Microelectronic CoolRISC derivates there are no data ROM memories. It means constants, tables, initialized global variables can not be stored in a data section of the CoolRISC defined for the link (.data, .page0_data, .rodata). These sections are defined as null. Nevertheless, it is possible to define such types (constants, tables, initialized global variables). To realize this operation, several approaches have to be considered: Tables in RAM – Tables in C functions – Tables in C/assembler. This application Note will introduce three different ways.

Tables in C/assembler

It draws a constant table embedded in the program memory field. This solution is recommended for large tables which require constant access time and limited memory size. It also gives easy insertion in a C-program and keep good compact size.
Nb Instr. = V + D + (N*(D+1))
V = Overhead code for index decoding and range check
D = Data size (1= byte, 2 = int, …)
N = Data number in the table

Table implementation

The table has to be defined in a software routine able to return a constant value based on an index value given as parameter.

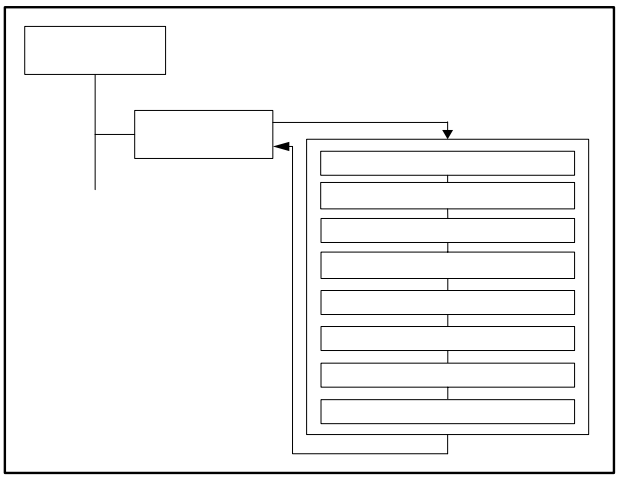

Figure 1: Principle

Each byte of data to be stored requires 2 instructions to restore the embedded byte value :

A move instruction : move reg, #table_value
A jump to the end of the table.

Copyright © 2006, EM Microelectronic-Marin SA

Examples of implementation

- One dimension char table

```
/* Example of table use */
if (GetConstTest(n)!= ExpectedArray[n])

/*-------------------------------------------------------------------------------------------------------------------
*/

/* realize the move operation */
#define   DataStored(val)  {asm ("move  %r2,#"#val"");\
asm("JUMP .Lab_End");}

unsigned char GetConstTest(unsigned char IndexConstTest) {
// optional check index range. Here range is 0 <-> 45
asm("CMP %r3,#46");
asm("JGE .Lab_ErrorIndex");
 /* compute the table defined below. Define where to find the value in the table.
  One value stored each (2*n); memory location */
asm("MOVE -(%i3,1),%ipl");                    // store current ip
asm("MOVE -(%i3,1),%iph");
asm("MOVE %iph,#HIWORD(.Lab_Table)");         // define table location start
asm("MOVE %ipl,#LOWORD(.Lab_Table)");
asm("MUL  %r3,#2");                           // find location from index
asm("ADD  %ipl,%a");
asm("ADDC %iph,%r3");
asm("JUMP %ip");;                             // table data definition
asm(".Lab_Table:");
DataStored(0xff);                             // index 0
DataStored(0xff);                             // index 1
DataStored(0xff);                             // index 2
...
DataStored(0xff);                             // index 44
DataStored(0x1f);                             // index 45
// finish ...
asm(".Lab_End:");
asm("MOVE %iph,(%i3,1)+");                    // restore ip for next jump
asm("MOVE %ipl,(%i3,1)+");
asm("JUMP %ip");
asm(".Lab_ErrorIndex:");
asm("Nop");                                   // 0 for index error
return 0;
}
```

- Other dimensions table or data size

The example above can be implemented as well for all possible needs (multi-array, int, long, …).
Some examples are shown below.

Int table:

```
#define   DataStored(val1, val2)  {asm ("move  %r3,#"#val1"");\
asm ("move  %r2,#"#val2"");\
asm("JUMP .L_End");}

int GetValue(unsigned char nIndex) {
  asm("       ; checking index range..
    CMP    %r3,#16
    JGE    .L__IndexError
  ");
  asm("       ; index range is ok, computing table entry address...
      MOVE   -(%i3,1),%ipl
      MOVE   -(%i3,1),%iph
      MOVE   %iph,#HIWORD(.L__Start)
      MOVE   %ipl,#LOWORD(.L__Start)
      MUL    %r3,#3
      ADD    %ipl,%a
      ADDC   %iph,%r3
             ; jumping to table entry...
      JUMP   %ip
   ");
  asm("       ; data table
      .L__Start:
             ; index is 0
      DataStored(0x00, 0x00);
            ...
             ; index is 15
      DataStored(0x00, 0x0F);
             ; end of table values
  ");
  asm("       ; end of the processing, exiting..
      .L__End:
     MOVE   %iph,(%i3,1)+
     MOVE   %ipl,(%i3,1)+
    JUMP   %ip
    .L__IndexError:
             ; default return value for out-of-range index is ...
  ");
  return 0;
```

Long table:

```
#define   DataStored(val1, val2, val3, val4)  {asm ("move  %r3,#"#val1"");\
asm ("move  %r2,#"#val2"");\
asm ("move  %r1,#"#val3"");\
asm ("move  %r0,#"#val4"");\
asm("JUMP .L_End");}

long GetValue(unsigned char nIndex) {
  asm("    ; checking index range..
    CMP    %r3,#16
    JGE    .L__IndexError
  ");
  asm("    ; index range is ok, computing table entry address...
      MOVE   -(%i3,1),%ipl
      MOVE   -(%i3,1),%iph
      MOVE   %iph,#HIWORD(.L__Start)
      MOVE   %ipl,#LOWORD(.L__Start)
      MUL    %r3,#5
      ADD    %ipl,%a
      ADDC   %iph,%r3
      ; jumping to table entry...
      JUMP   %ip
   ");
  asm("    ; data table
      .L__Start:
             ; index is 0
     DataStored(0x00, 0x0F, 0x04, 0x23);
       ...       ; index is 15
     DataStored(0x10, 0x2F, 0x34, 0x29);
             ; end of table values
  ");
  asm("    ; end of the processing, exiting..
      .L__End:
  MOVE   %iph,(%i3,1)+
  MOVE   %ipl,(%i3,1)+
  JUMP   %ip
      .L__IndexError:
  ; default return value for out-of-range index values is given by the next C
code...
  ");
  return 0;
  }
```

Copyright © 2006, EM Microelectronic-Marin SA

Tables in RAM

It draws a constant table by coding initialization RAM. It is recommended for short access time and size limited tables. Moreover, user has to ensure data integrity no overwrite) and handle data location.
Nb Instr. = v+ 2*N*D
Data RAM = D*N
V = Overhead code
D = Data size (1= byte, 2 = int, …)
N = Data number in the table

```
DataInit:
        move    -(i3),      ipl
        move    -(i3),      iph
        move    i0h, #HIWORD(_spage0data)
        move    i0l, #LOWORD(_spage0data)
        move    a, #0x01
        move    (i0)+, a    ; Addr = 0x61
        move    a, #0x04
        move    (i0)+, a    ; Addr = 0x62
        …
        move    iph,        (i3)+
        move    ipl,        (i3)+
        rets
```

Data initialization

It is possible to implement a mechanism which automatically generate DataInit code. By using binary tools an assembler file can be fulfilled with init code and linked with the others source project in order to generate a DataInit piece of code. By using this mechanism, user can writes its code with no restriction in term of coding (C constant declaration, initialized global variables, …). This solution is suitable as long as Tables in RAM can be used in the project (size limited tables).
This solution is provided since EM CoolRISC Environment 2 V3.0.

The corresponding project example is available (contact EM Microelectronic).

Copyright © 2006, EM Microelectronic-Marin SA

2. How to generate several text sections

Introduction

This application note shows how to write code using several code areas. It is useful when you want to constraint a piece of code to be loaded in a specific memory location.
This Application Note introduces a general way to define the different sections used to store assembler and/or C-code.
There are various ways to define the mapping: one section for each memory area, one memory for several sections, etc …

Example

In this example we create three code sections for an 8K instructions code memory size stored in two memory areas.
The goal is to reserve and store four instructions in the .text1 section and four other instructions in the .text2 instructions.
The content of .text1 is generated from assembler code and .text2 from C-code.

MCU address	Linker Mapping	Memory	Section
0x0	0x100000	main_rom	.text
0x1FF7	0x107FDF		
0x1FF8	0x107FE0	special_rom	.text1
0x1FFB	0x107FEF		
0x1FFC	0x107FF0		.text2
0x1FFF	0x107FFF		

Table 1: Memory organization summary

Note: CoolRISC Mapping rule

- *In the mapping of the linker script a positive 0x100000 offset is added for the code.*
- *Each instruction word takes four bytes (whereas one instruction is 22-bit wide).*

Linker script

The linker script (crt0.ld) must contain the following information:

```
...
MEMORY
{
  main_rom       : ORIGIN = 0x00100000, LENGTH = 4*8K - 32
  special_rom    : ORIGIN = 0x00107FE0, LENGTH = 32
 …
}
...
SECTIONS
{
.text :                  /* THE MAIN CODE IS STORED IN THE .TEXT SECTION */
   {
      _stext = .;
      *(.text )
      _etext = .;
      _eprom = .;
   } > main_rom
.text1 0x107FE0 :   /* THE CODE 1 IS STORED IN THE .TEXT1 SECTION */
   {
   *(.text1)
     _eprom = .;
   } > special_rom
.text2 0x107FF0 :   /* THE CODE 2 IS STORED IN THE .TEXT2 SECTION */
   {
   *(.text2)
     _eprom = .;
   } > special_rom
```

Copyright © 2006, EM Microelectronic-Marin SA

Once, this linker definition is done, it is necessary to explicitly constraint the code to its section destination.

Assembler

For the assembler code, It's required to add specific directives to give information to the linker and loader that the new section (.text1) has to be taken in account.

In our example (special_code_1.s):

```
.section .text1,"ax",@progbits
.global _code_1

_code_1:
   nop
   nop
   nop
   ret
```

Note: section directive

A summary of the most common options are described here.
The directive section to assemble assembler code into a section is defined as follow (we have to consider ELF format version) :

.section name, "flags", @type

The quoted "flags" argument may contain a combination of the following characters:
a (allocatable) / w (writable) / x (executable) / M (mergeable)
The optional type may contain one of the following constants:
@progbits (contains data) / @nobits (only occupies space) / @note (data but not used by program)
In our case the "ax", @progbits tells the assembler that the section is allocatable (a"), executable ("x") and contains data (@progbits).

C-code

For the C-code, it is necessary to define the section where the code must be located. It is done by adding an attribute which define the section.

In our example (special_code_2.c):

```
void code_2(void)
{
  asm("nop");
  asm("nop");
  asm("ret");    // will be added one more instruction for jump ip
};
```

With the following function prototype (special_code_2.h):

```
void code_2 (void) __attribute__ ((section (".text2")));
```

Results

The disassembly code is:

```
1FF4  000000              move    FF, #FF
1FF5  000000              move    FF, #FF
1FF6  000000              move    FF, #FF
1FF7  000000              move    FF, #FF
                    _code_1
1FF8  3FFFFF               nop
1FF9  3FFFFF               nop
1FFA  3FFFFF               nop
1FFB  3F3FFF               ret
                    code_2
1FFC  3FFFFF               nop
1FFD  3FFFFF               nop
1FFE  3F3FFF               ret
1FFF  23FFFF               jump    ip
```

Copyright © 2006, EM Microelectronic-Marin SA

The output ELF file contain the different information:

```
Section Headers:
  [Nr] Name              Type            Addr     Off    Size   ES Flg Lk Inf Al
  [ 0]                   NULL            00000000 000000 000000 00      0   0  0
  [ 1] .text             PROGBITS        00100000 0000d4 00035c 00  AX  0   0  1
  [ 2] .text1            PROGBITS        00107fe0 000430 000010 00  AX  0   0  1
  [ 3] .text2            PROGBITS        00107ff0 000440 000010 00  AX  0   0  1
             ...            ...             ...      ...
Key to Flags:
  W (write), A (alloc), X (execute), M (merge), S (strings)
  I (info), L (link order), G (group), x (unknown)
  O (extra OS processing required) o (OS specific), p (processor specific)

Program Headers:
  Type           Offset   VirtAddr   PhysAddr   FileSiz MemSiz  Flg Align
                   ...      ...        ...        ...
  LOAD           0x0000d4 0x00100000 0x00100000 0x0035c 0x0035c R E 0x1
  LOAD           0x000430 0x00107fe0 0x00107fe0 0x00020 0x00020 R E 0x1

 Section to Segment mapping:
  Segment Sections...
                   ...      ...        ...        ...
   03     .text
   04     .text1 .text2
```

The corresponding project example is available (contact EM Microelectronic).

Copyright © 2006, EM Microelectronic-Marin SA

3. What is the hex file format

Introduction

This application note introduces the object IntelHEX file Format used for CoolRISC loader and/or programmer.
IntelHEX format is representing binary object file in hexadecimal ASCII. It describes the CoolRISC code/data of the microcontroller to store/load with corresponding address to be loaded.
For most cases it only contains program code but it can also enclose data field in case data ROM exist (sections .data, .rodata, page0_data).
IntelHex file is generated from the output ELF linker file with a format tool converter (e.g c816-objcopy, c816-srec2rom …)

Note: For detailed and general IntexHEX specification please refer to the Intel literature.

IntelHEX overview

An IntelHEX format is made of records. Each record contains:
- ✓ record type,
- ✓ length,
- ✓ memory load address,
- ✓ checksum.

Each record is based on:

:	XX	XXXX	XX	XX…XX	XX
Record mark	**Record length** (nb of bytes of information/data which follow the Record type)	**Load offset** (gives the 16-bit starting load offset of data bytes -> Data records. For other record it is 0x0000)	**Record Type** 00 Data 01 End of file 02 Extended Segment address 03 Start Segment address 04 Extended Linear Address 05 Start Linear address	**Data / Information**	**Cheksum CRC** Sum of Bytes + CRC = 0

Notes:
I. X is a nibble
II. Main record Types used for CoolRISC are:
- ✓ Extended Linear address ":**02000004**X…X"
- ✓ End of file ":**00000001FF**"
- ✓ Data ":XXXXXX**00**X…X"
- ✓ Start Linear address ":**04000005**X…X"

III. Extended Linear address record is used before data record when an offset is greater than 16-bits (offset > 0xFFFF -> 32-bits mode).

CoolRISC example

The following example shown describes a program section. For all CoolRISC targets an offset of 0x100000 must be set (.text section start at 0x100000).

intelHex file example:

```
:02000004 0010EA                                  -> See note A
:100000000033FFFB0033FFDF0033FFAC0033FF7D25       -> See note B
:10001000000EA7FD000EA6A0000EA3FF000EA29FDB       -> See note C
:10002000000EA5FF000EA49E0033FFF300035E0147
…
:100330000023FFFF0023FFFF0023FFFF0023FFFF39
:080340000023FFFF0023FFFF73                       -> See note D
:04000005 00100000E7                              -> See note E
:00000001FF                                       -> See note F
```

Copyright © 2006, EM Microelectronic-Marin SA

Note A:

:02000004	0010	EA
Type = Extended Linear address definition	LBA = Define the Linear Base Address. 0x0010 is used for the ULBA (upper bits): - ULBA= [31:16] bits. - [15:0] are always 0x0000 -> Gives the 0x100000 offset for the code area	CRC

This record introduce code. Due to the extended value offset, it is necessary to set extended mode. All IntexHex files for CoolRISC always start with :020000040010EA. In case code a piece of code is located to a lower address (16-bit range), this extended Linear address is not necessary and code could be defined within a Data record with the offset (e.g. with offset 0x900 : :10900000033FFFB0033FFDF0033FFAC0033FF7DXX)

Note B:

:10	0000	00	0033FFFB, 0033FFDF, 0033FFAC, 0033FF7D	25
Number of pairs (=2*n bytes)	Address (start address for the current line)	00 Data Record	words	CRC

Each instruction is four Bytes. Number of bytes for each line is 16d (10h) (so 4 instructions per line)

Note C:

:10	0010	00	000EA7FD, 000EA6A0, 000EA3FF, 000EA29F	DB
Number of pairs (=2*n bytes)	Address (start address for the current line)	00 Data Record	words	CRC

Note D:

:08	0340	00	0023FFFF, 0023FFFF	73
Number of pairs (=2*n bytes)	Address (start address for the current line)	00 Data Record	words	CRC

End of code data record.

Note E:

:04000005	00100000	E7
Type = 32-bits Start Linear address definition	EIP= Linear address for EIP register CRC	CRC

It is used to define the execution start address for the object file. Force the address for the loader.
All IntexHex files for CoolRISC finish with :020000040010EA followed by the end of file record. In fact this line is not necessary (execution is always 0x0000).

Note F:

:00000001FF
Type= End of File

The end of file record specifies the end of the hexadecimal object file.

Data ROM and Initialized data:

In case the target has data ROM some additional informations have to be defined for the ROM data memory. If ROM data is mapped on a 16-bit range adressing, it can simply be defined within a simple Data Record with offset definition. If it is on an extended (32-bit range addressing), an extended linear address record is required first.

intelHex file sample example:

:02006500010494 -> Data record - 2 bytes (0x01, 0x04) to store with offset 0x65
:10010000080700010203040506A0B0C0D0E0F14155D -> Data record - 16 bytes (0x 08,…0x 15) - Offset 0x100
:04011000161718198D -> Data record - 4 bytes (0x16, …0x 19) - Offset 0x110

EM Microelectronic-Marin SA (EM) makes no warranty for the use of its products, other than those expressly contained in the Company's standard warranty which is detailed in EM's General Terms of Sale located on the Company's web site. EM assumes no responsibility for any errors which may appear in this document, reserves the right to change devices or specifications detailed herein at any time without notice, and does not make any commitment to update the information contained herein. No licenses to patents or other intellectual property of EM are granted in connection with the sale of EM products, expressly or by implications. EM's products are not authorized for use as components in life support devices or systems.