Freescale Semiconductor
Application Note

Document Number: ANnnnn
Rev. 1, 9/2015

How to develop bare metal codes running on M4 core

By Freescale Semiconductor, Inc.

The purpose of this document is to describe how to develop bare metal codes running on M4 core for imx6SoloX.

history	
V1	• Create documentation

Contents

1 Introduction

Even though we have provided MQX OS support for imx6SX M4 core, we don’t need such complex operation system running on M4 core in some cases. For users who want to simplify and accelerate their M4 application development, bare metal codes without OS support is also a good choice. This chapter will describe how to develop bare metal codes running on M4 core through a demo application.

2 Development environment setup

For embedded system development, we always need to build cross compile development environment.

2.1 Development tool choose

These are many development tools (commercial or free) to assemble, compile or link bare metal codes. Now, some popular tools are as followed:

- GNU Tools for ARM Embedded Processors
- ARM Development Studio 5, DS-5
- RealView ICE

In this document, we choose Ubuntu 12.04 as host development environment and choose GNU Tools for ARM Embedded Processors as our development tools. You can download the tool at:

https://launchpad.net/gcc-arm-embedded

2.2 Install the gnu tools tarball

When you have your own Ubuntu host, you can execute the following commands to install GNU cross compile tools:

mkdir –p /opt/toolchain

cd /opt/toolchain

Please copy gnu tools tarball to this directory

tar -xvf gcc-arm-none-eabi-4_8-2013q4-20131204-linux.tar.tar

Version will be changed in future release

2.3 Create the build environment setup script

For every Linux shell user, you need to set your own shell environment variable. In order to speeding up your development, we provided a demo script to set these environment variable. Let’s name the script to be “m4-build-env”, and save it under ~/bin/

It contains:

```
#!/bin/bash

TOOLBIN=/opt/toolchain/gcc-arm-none-eabi-4_9-2015q3/bin

export CROSS_COMPILE=$TOOLBIN/arm-none-eabi-

export PATH=$PATH:$TOOLBIN
```

Now, your cross compile development environment is ready, you can use the tool to cross compile your application on Ubuntu host.

3 Basic workflow

As we all know, Cortex-A9 always boots as the primary core and Cortex-M4 does not have a boot ROM and is also not provided a clock at POR. So, Cortex-A9 is responsible for loading M4 binary image to TCM and launching the Cortex-M4 by enabling its clock.

In this demo, in order to simplify workflow, A9 will also run a simple bare metal application. This A9 application will basically arrange for necessary run-time environment for M4 and finally launch M4 application.

3.1 Tasks for A9

A9 will boot from any boot source (e.g. SD4), setup the environment for M4 and finally launch M4. The basic tasks are as follows:

1. Enable the peripheral access for M4
M4 core can access most of the peripherals, so the user should setup the AIPSTZ bridge access privilege for M4 master. Please note that A9 and M4 all can setup AIPSTZ bridge access privilege through Master Privilege Registers, but the memory map is different. For example:

AIPS1, AIPS2 & AIPS3 have +0x4000000 offset in M4

Please check memory map chapter from the Reference Manual.

2. Turn on M4 clock
3. Load the M4 binary image to TCM
The Cortex-M4 includes two tightly coupled memories:

TCML and TCMH (32K Byte + 32K Byte)

The A9 will load M4 binary image to TCML. Because the M4 binary image will include the vectors table for M4, what we should do is copying the image to 0x00000000(M4 mapped address).

Please note that:

TCML Address for A9 is 0x007f8000

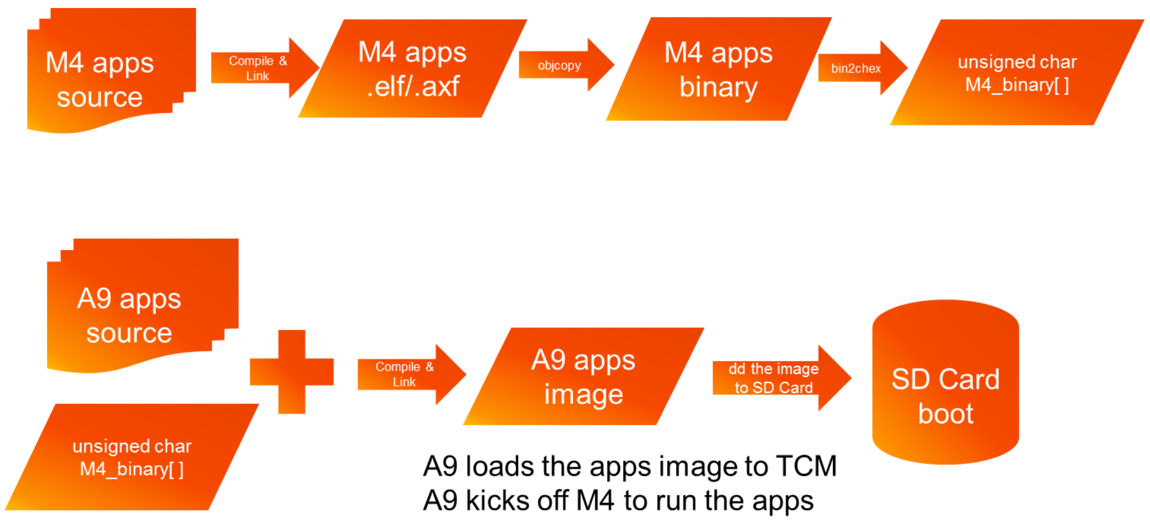
TCML Address for M4 is 0x1FFF8000 and alias at 0x00000000

4. Enable M4 and release the reset for M4
Now, we can turn on the M4 clock and reset M4. After M4 reset, M4 will jump to the address according to the reset vector in 0x00000004 and then branch to main ()

3.2 The binary image

In this demo, there is only one final image which is compiled and linked from A9 application source code and M4 binary image. This method will make us easy to flash image to boot device.

The following picture had described how to generate this final image.



Firstly, you need to compile your M4 application and generate M4 binary image. This image is executable binary file, but we will not load it into RAM directly. Instead, we transform it to be a char array (M4_binary []);

Secondly, we will compile and link A9 application source codes together with M4_binary array, and then generate a final A9 application binary image. The A9 codes will be responsible for loading M4_binary to M4 RAM.

4 M4 code skeleton

In order to make easy for M4 application development, we have provided a common M4 code skeleton for developer as follows:

- Link file

- Vector table
- Main function

4.1 Link file

As you know, the link file is important for cross-compile and it decides how to link your intermediate object together and then generate an executable binary file. For how to write your own link file, you can refer to following file:

```
ENTRY(_reset)
OUTPUT_FORMAT("elf32-littlearm", "elf32-littlearm", "elf32-littlearm")
OUTPUT_ARCH(arm)
MEMORY
{
        TCML_ALIAS (rwx) : ORIGIN = 0x00000000, LENGTH = 32K
}
SECTIONS
{
        .vectors :
        {
                *o(.vectors_)
        }>TCML_ALIAS
        .text :
        {
                . = ALIGN (4);
                *(.text)
        }
        .data :
        {
                . = ALIGN (4);
                *(.data)
        }
        .bss :
        {
                . = ALIGN (4);
                *(.bss)
        }
}
```

4.2 Vectors table

On system reset, the vector table is fixed at address 0x00000000. When an exception or interrupt occurs, the M4 processor sets the pc to a corresponding memory address. The corresponding memory address are entries that branch to specific routines designed to handle a particular exception or interrupt. For example, after M4 reset, M4 will jump to the address according to the reset vector in 0x00000004. When SysTick occurs, M4 will jump to the address according to the reset vector in 0x0000003C.

The customer can implement their own interrupt and exception handle function. In this demo, we have implemented SysTick and Reset handle function. The entry to SysTick interrupt is SysTickHandler label. The entry to Reset interrupt is _reset label.

The following table has described the vector table for Cortex M4.

Address	Vector
0x00000000	Initial SP value
0x00000004	Reset
0x00000008	NMI
0x0000000C	Hard fault
0x00000010	Memory management fault
0x00000014	Bus fault
0x00000018	Usage fault
	Reserved
0x0000002C	SVCall
	Reserved
0x00000038	PendSV
0x0000003C	SysTick
0x00000040	IRQ0
0x00000044	IRQ1
	.
	.
	.

```
/* vectors.s */
.thumb
.word   0x20007000  /* stack top address */
.word   _reset      /* 1 Reset */
.word   hang        /* 2 NMI */
.word   hang        /* 3 HardFault */
.word   hang        /* 4 MemManage */
.word   hang        /* 5 BusFault */
.word   hang        /* 6 UsageFault */
.word   hang        /* 7 RESERVED */
.word   hang        /* 8 RESERVED */
.word   hang        /* 9 RESERVED*/
.word   hang        /* 10 RESERVED */
.word   hang        /* 11 SVCall */
.word   hang        /* 12 Debug Monitor */
.word   hang        /* 13 RESERVED */
.word   hang        /* 14 PendSV */
.word   SysTickHandler  /* 15 SysTick */
.word   hang        /* 16 IRQ 0 */
.word   hang        /* 17 IRQ 1 */
.word   hang        /* 18 IRQ 2) */
.word   hang        /* 19 ...   */
.thumb_func
.global _reset
_reset:
    mov r0, #0
    ldr r1, [r0]
    mov sp, r1
    bl main
    b hang
.thumb_func
hang:   b .
```

4.3 Main function

According to vector table, after M4 reset, M4 will jump to the address according to the reset vector in 0x00000004, and then branch to main(). So, you must implement your own main function. This function should your C entry to your M4 application.

```
int main(int argc, char ** argv)
{
…
}

void SysTickHandler(void)
{
…
}
```

5 Demo application

The purpose of this demo is to print “Hello World” on UART2 which controlled by M4 core. It is a bare metal code which can build under GNU toolchain under Linux host.

The M4 application sources include:

Vector Table: vectors.s

Link Script: cortex_m4.ld

C code: main() …

5.1 Compile M4 binary image

You need to cross-compile your M4 application on Ubuntu host. The commands are as follows:

```
m4-build-env

cd m4_apps

make
```

The compile has 3 steps:

1. Compile with “-mcpu=cortex-m4 -mthumb” options

```
arm-none-eabi-gcc -mcpu=cortex-m4 -mthumb -c -g vectors.s -o vectors.o

arm-none-eabi-gcc -mcpu=cortex-m4 -mthumb -c -g main.c -o main.o

arm-none-eabi-gcc -mcpu=cortex-m4 -mthumb -c -g iomux-v3.c -o iomux-v3.o
```

```
arm-none-eabi-gcc -mcpu=cortex-m4 -mthumb -c -g timer.c -o timer.o
```

```
arm-none-eabi-gcc -mcpu=cortex-m4 -mthumb -c -g uart.c -o uart.o
```

2. Link

```
arm-none-eabi-ld -g -T cortex_m4.ld vectors.o main.o iomux-v3.o timer.o uart.o -o cortex_m4.elf
```

3. Extract the binary from elf

```
arm-none-eabi-objcopy -O binary cortex_m4.elf cortex_m4.bin
```

4. Convert cortex_m4.bin to m4_binary[ ]

```
echo "unsigned char M4_binary[] = {" > m4_bin.h

bin2chex cortex_m4.bin >> m4_bin.h

echo "};" >> m4_bin.h
```

The m4_binary[ ] is C char array and will be loaded to TCM by A9.

5.2 Compile A9 binary image

You need to cross-compile your A9 iamge on Ubuntu host. The commands are as follows:

```
m4-build-env

cd a9_launch_m4

make
```

Please note you should copy the new m4_bin.h to this directory.

When make process completed, you can find a9_launch_m4.bin. This is the final A9 binary image.

5.3 Running the Demo

The demo will use SD card as boot device.

1. Transfer the A9 binary to a SD card

```
sudo dd if=a9_launch_m4.bin of=/dev/sda bs=1K skip=1 seek=1 && sync
```

2. Connect “UART to USB” port to PC

The port contains 2 devices:

UART1 for A9

UART2 for M4

3. Boot the card from SD4 port

You should see “Hello World from Cortex-M4!” in UART2 port

How to Reach Us:

Home Page:
www.freescale.com

Web Support:
http://www.freescale.com/support

USA/Europe or Locations Not Listed:
Freescale Semiconductor
Technical Information Center, EL516
2100 East Elliot Road
Tempe, Arizona 85284
+1-800-521-6274 or +1-480-768-2130
www.freescale.com/support

Europe, Middle East, and Africa:
Freescale Halbleiter Deutschland GmbH
Technical Information Center
Schatzbogen 7
81829 Muenchen, Germany
+44 1296 380 456 (English)
+46 8 52200080 (English)
+49 89 92103 559 (German)
+33 1 69 35 48 48 (French)
www.freescale.com/support

Japan:
Freescale Semiconductor Japan Ltd.
Headquarters
ARCO Tower 15F
1-8-1, Shimo-Meguro, Meguro-ku,
Tokyo 153-0064, Japan
0120 191014 or +81 3 5437 9125
support.japan@freescale.com

Asia/Pacific:
Freescale Semiconductor China Ltd.
Exchange Building 23F
No. 118 Jianguo Road
Chaoyang District
Beijing 100022
China
+86 010 5879 8000
support.asia@freescale.com

For Literature Requests Only:
Freescale Semiconductor Literature
Distribution Center
P.O. Box 5405
Denver, Colorado 80217
1-800-441-2447 or 303-675-2140
Fax: 303-675-2150
LDCForFreescaleSemiconductor@hibbertgroup.com

Document Number: ANnnnn
09/2013