ARC labs handbook

Release 2018.09

Synopsys

2018

CONTENTS:

CHAPTER

ONE

OVERVIEW

1.1 Introduction

This is a handbook for ARC labs which is a part of ARC university courses. It's written to help students who attend the ARC university courses and anyone who is interested in DesignWare® ARC® processors to get started in DesignWare® ARC® processors processor development. It describes all the basic elements of ARC labs and how to finish the labs with step by step approach.

This handbook can be used as a Lab teaching material for ARC university courses at undergraduate or graduate level with majors in Commuter Science, Computer Engineering, Electrical Engineering; or for professional engineers.

This handbook includes 12 labs currently (more labs will be added in the future), which can be classified into 3 levels:

- *Level 1: ARC basic*

The labs in this level cover the basic topics about DesignWare® ARC® processors, e.g., the installation of tools, hello world, interrupts, timers and so on.

- *Level 2: ARC advance*

The labs in this level cover the advanced topics about DesignWare® ARC® processors, e.g., RTOS, customized linkage, ARC DSP and so on.

- *Level 3: ARC exploration*

The labs in this level will cover some complex applications about DesignWare® ARC® processors, e.g., IoT application, embedded machine learning and so on.

Most of labs are based on the embARC Open Software Platform (OSP) which is an open software platform to facilitate the development of embedded systems based on DesignWare® ARC® processors.

It is designed to provide a unified platform for DesignWare® ARC® processors users by defining consistent and simple software interfaces to the processor and peripherals, together with ports of several well known FOSS embedded software stacks to DesignWare® ARC® processors.

For more details about embARC OSP, please refer its online docs

1.2 Supported Hardware Platform

At present, the following hardware platforms are supported in this handbook.

- ARC EM Starter Kit
- ARC IoT Development Kit

You can go to the above specific link to get the board's data sheet and user manual as a reference.

1.3 Reference

Here is reference for this hand book.

Item	Name
1	ARC EM Databook
2	MetaWare docs
3	ARC EM Starter Kit User Guide
4	ARC GNU docs

CHAPTER

TWO

GETTING STARTED

Use this guide to get started with your ARC labs development.

2.1 Software Requirement

- **ARC Development Tools** Choose **MetaWare Toolkit** and/or **ARC GNU Toolchain** from the following list according to your requirement.
  - MetaWare Toolkit
    - Premium MetaWare Development Toolkit (2018.06) The DesignWare ARC MetaWare Development Toolkit builds upon a 25-year legacy of industry-leading compiler and debugger products. It is a complete solution that contains all the components needed to support the development, debugging and tuning of embedded applications for the DesignWare ARC processors.
    - DesignWare ARC MetaWare Toolkit Lite (2018.06) A demonstration/evaluation version of the MetaWare Development Toolkit is available for free from the Synopsys website. MetaWare Lite is a functioning demonstration of the MetaWare Development Toolkit, but has a number of restrictions, including a code-size limit of 32 Kilobytes and no runtime library sources. It is available for Microsoft Windows only.
  - ARC GNU Toolchain
    - Open Source ARC GNU IDE (2018.03) The ARC GNU Toolchain offers all of the benefits of open source tools, including complete source code and a large install base. The ARC GNU IDE Installer consists of Eclipse IDE with ARC GNU plugin for Eclipse, ARC GNU prebuilt toolchain and OpenOCD for ARC
- Digilent Adept Software for Digilent JTAG-USB cable driver. All the supported boards are equipped with on board USB-JTAG debugger, so just one USB cable is required, no need for external debugger.
- Tera Term or PuTTY for serial terminal connection, 115200 baud, 8 bits data, 1 stop bit and no parity (115200-8-N-1) by default.

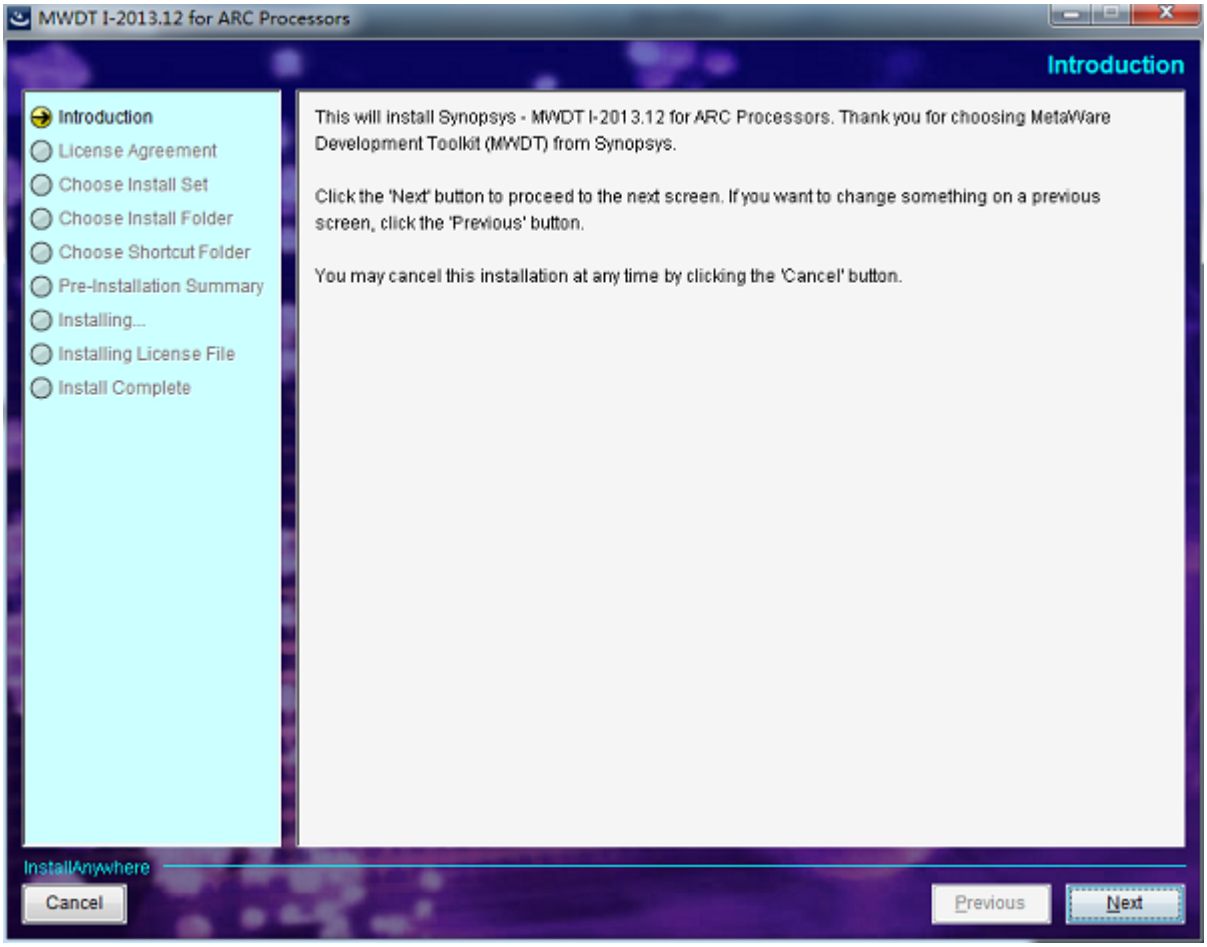
Note: If using embARC with GNU toolchain on Windows, install Zadig to replace FTDI driver with WinUSB driver. See How to Use OpenOCD on Windows for more information. If you wanto switch back to Metaware toolchain, you should also switch back the usb-jtag driver from WinUSB to FTDI driver.

2.2 Install Software Tools

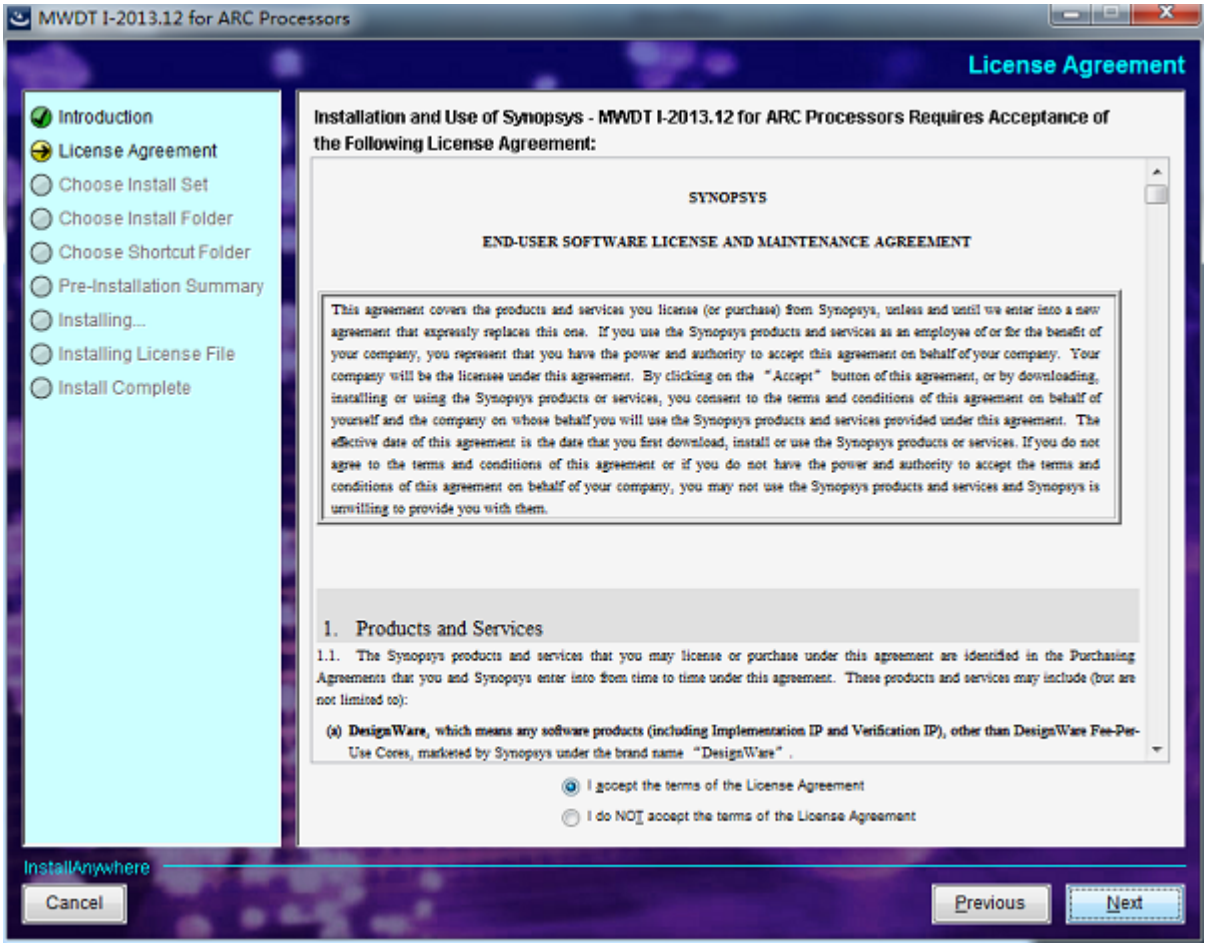
2.2.1 Install Metaware Toolkit

Here we will start install MetaWare Development Toolkit (2017.09).

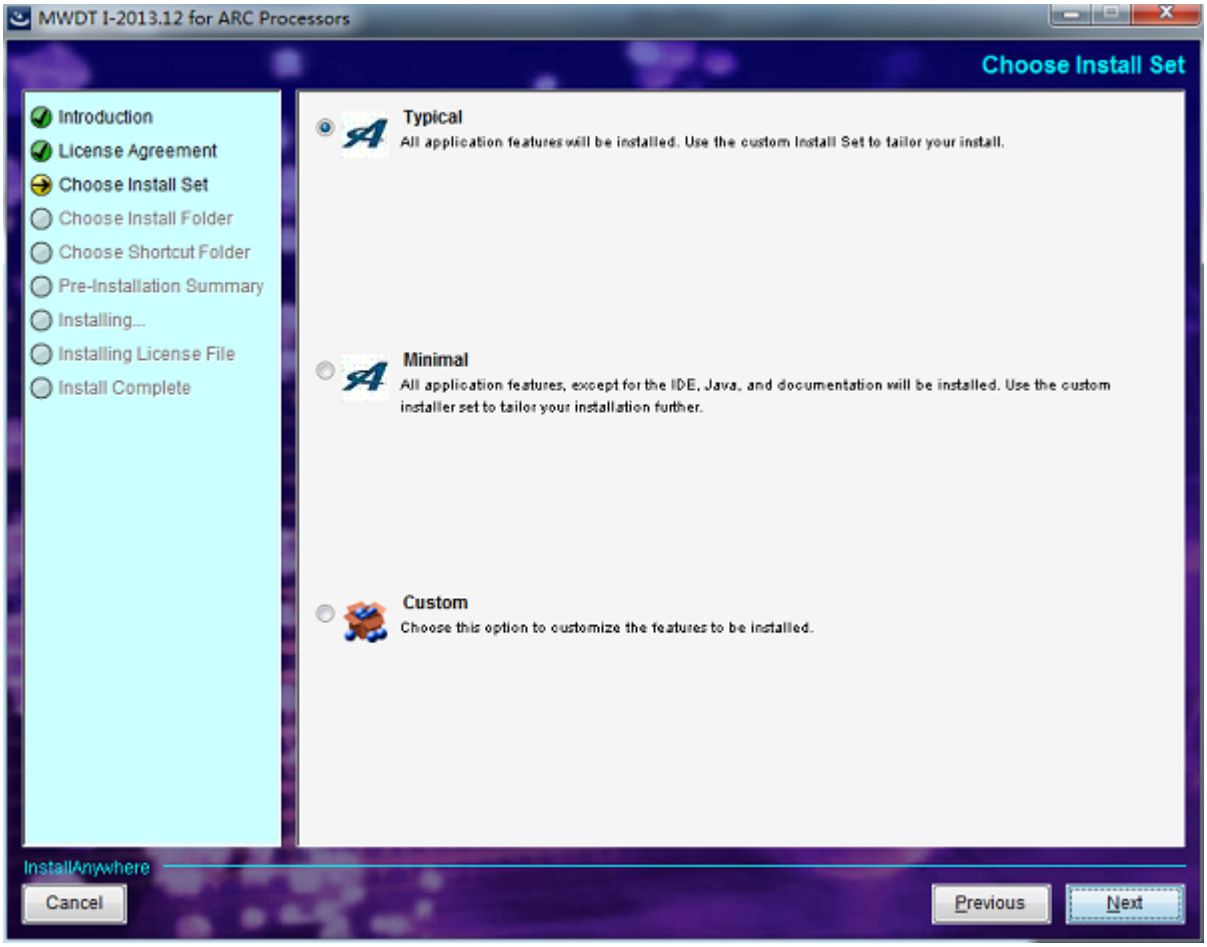
1. Double click the **mw_dekit_arc_i_2017_09_win_install.exe**, it will show



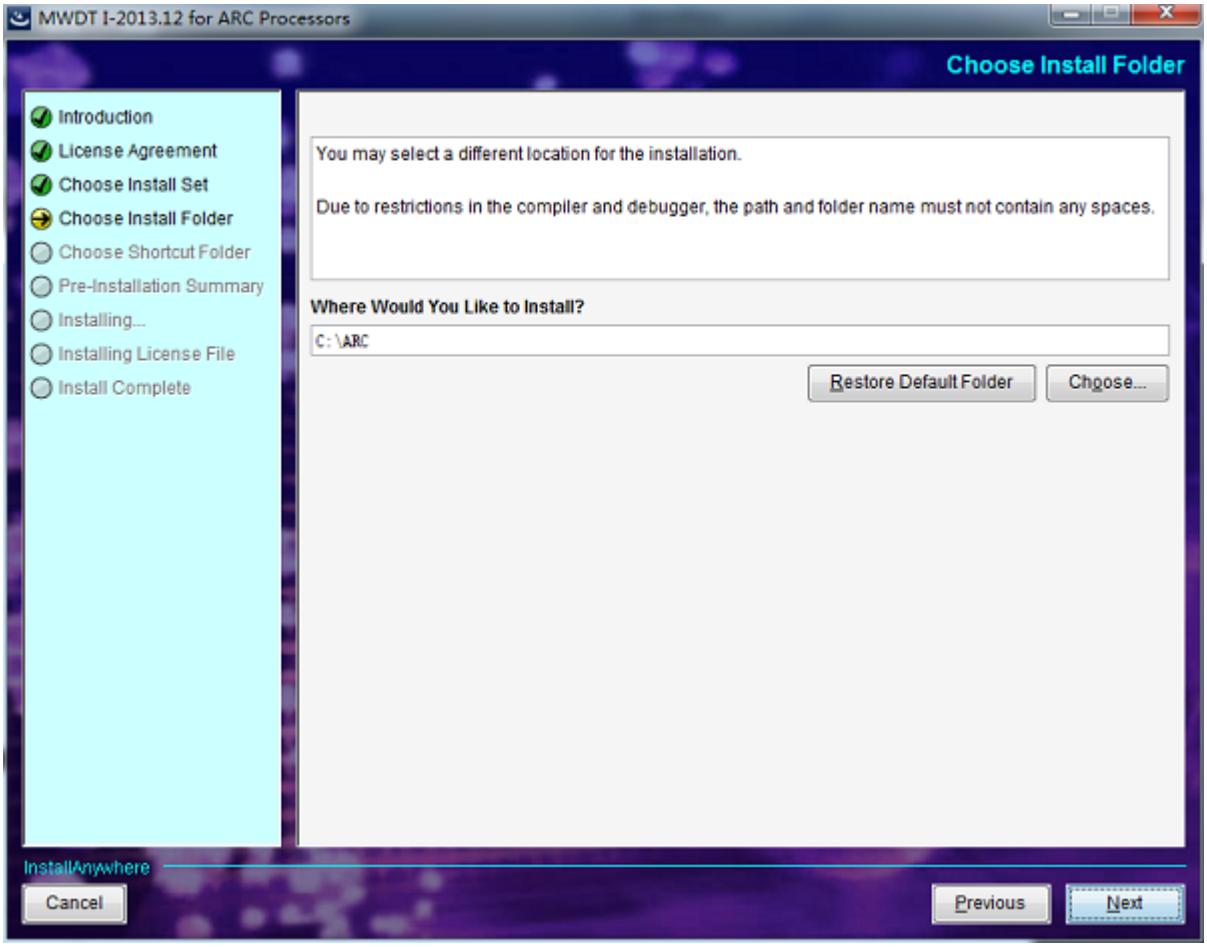
2. Click next, choose I accept, continue to click next



3. Choose Typical installation, then click next

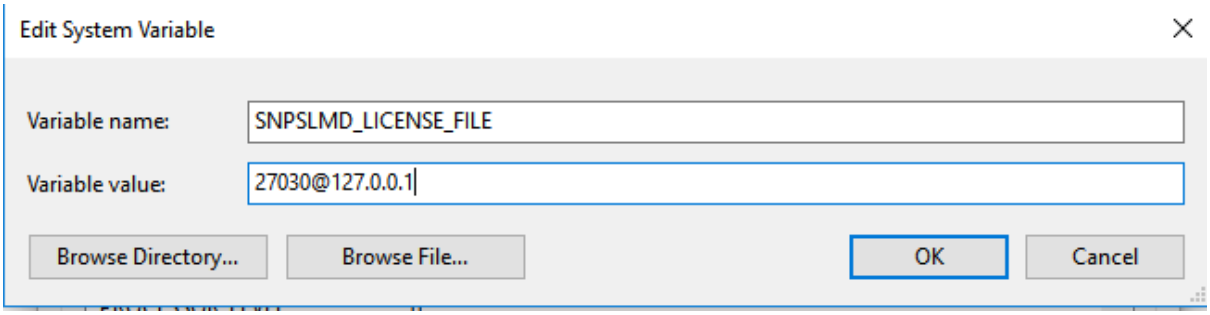


4. Set the install path (please use English letters only and no space), then click next until the installation is finished



5. Set the license file (SNPSLMD_LICENSE_FILE) for MetaWare Development Toolkit. It can be a real file containing license, also can be a license server

- For Windows, go to Computer->property->Advanced->Environment Variables->System Variables->New to Set



- For Linux, please add SNPSLMD_LICENSE_FILE into your system variables

6. Test the MetaWare Development Toolkit and its license

Open cmd.exe in Windows and find the queens.c in the installation folder of MetaWare Development Toolkit, e.g., **C:/ARC/MetaWare/arc/demos/queen.c**. Type the following commands in cmd

```
# On Windows
cd  C:\ARC\MetaWare\arc\demos
ccac queens.c
```

If you get the following message and no error, it means MetaWare Development Toolkit is successfully installed and license is ok.

```
MetaWare C Compiler N-2017.09 (build 005)       Serial 1-799999.
(c) Copyright 1987-2017, Synopsys, Inc.
MetaWare ARC Assembler N-2017.09 (build 005)
(c) Copyright 1996-2017, Synopsys, Inc.
MetaWare Linker (ELF/ARCompact) N-2017.09 (build 005)
(c) Copyright 1995-2017, Synopsys, Inc.
```

2.2.2 Install ARC GNU Toolchain

Please go here (https://github.com/foss-for-synopsys-dwc-arc-processors/toolchain/releases) to get the latest version of ARC GNU toolchain.

To use and install ARC GNU toolchain, you can refer this (http://embarc.org/toolchain/ide/index.html), where has detailed instructions.

It's recommended to install ARC GNU toolchain in the path (windows: `C:\arc_gnu\`, linux: `~/arc_gnu/`) and add `arc_gnu/bin` into $PATH variable.

2.2.3 Install embARC OSP

The embARC OSP source code is hosted in a GitHub repository that supports cloning via git. There are scripts and such in this repo that you'll need to set up your development environment, and we'll be using Git to get this repo. If you don't have Git installed, see the beginning of the OS-specific instructions below for help.

We'll begin by using Git to clone the repository anonymously.

```
# On Windows
cd %userprofile%
# On Linux
cd ~

git clone https://github.com/foss-for-synopsys-dwc-arc-processors/embarc_osp.git␣
↪embarc_osp
```

You have successfully checked out a copy of the source code to your local machine.

2.2.4 Install ARC labs code

The source codes of ARC labs are assumed to work with embARC OSP. You need to use git to clone or download the ARC labs first to **arc_labs** folder, then copy the `arc_labs` folder to the root folder of embARC OSP. You will get the following folder structure:

```
cd path/to/embarc_osp
git clone https://github.com/foss-for-synopsys-dwc-arc-processors/arc_labs.git arc_
↪labs
```

```
embarc_osp
├── arc
├── board
├── device
├── doc
├── example
├── arc_labs
├── inc
├── library
├── middleware
├── options
└── os
```

2.3 Final Check

Check the following items and set development environment.

- Make sure the paths of the above required tools for the MetaWare toolkit and ARC GNU toolchain are added to the system variable **PATH** in your environment variables.
- We recommend users to install ARC GNU IDE to default location. Otherwise you need to make additional changes as below.
  - If running and debugging embARC applications using **arc-elf32-gdb** and OpenOCD for ARC, make sure 1) the path of **OpenOCD** is added to the **PATH** in your environment variables, and 2) modify **OPENOCD_SCRIPT_ROOT variable** in *<embARC>/options/toolchain/toolchain_gnu.mk* according to your **OpenOCD** root path.
  - If running GNU program with using the GNU toolchain on Linux, modify the **OpenOCD** configuration file as Linux format with LF line terminators. **dos2unix** can be used to convert it.

Note: Check the version of your toolchain. The embARC OSP software build system is purely makefile-based. *make/gmake* is provided in the MetaWare toolkit (gmake) and ARC GNU toolchain (make)

CHAPTER

THREE

LABS

3.1 Overview

3.2 Labs

3.2.1 Level 1 Labs

How to use ARC IDE

MetaWare ToolKit

Purpose

- To learn the MetaWare IDE integration interface
- To get familiar with the use of the MetaWare IDE interface and command line
- To get familiar with the features and usage of the MetaWare Debugger debugger

Equipment

The following hardware and software tools are required:

- PC host
- MetaWare Development Toolkit
- nSIM simulator or ARC board (EM Starter kit/IoT Development Kit)
- `embarc_osp/arc_labs/labs/lab1_core_test`

PC, MetaWare Development Toolkit, nSIM simulator, core_test source file in embAR OSP

Content

Create a C project using the Metaware IDE graphical interface, import the code CoreTest.c, configure compilation options to compile, and generate executable files.

Start the debugger of MetaWare IDE and enter debug mode. From the different angles of C language and assembly language, use the functions of setting breakpoint, single step execution, full speed execution, etc., combined with observing PC address, register status, global variable status and Profiling performance to analyze the debug target program.

Principles

Use the MetaWare IDE integrated development environment to create projects and load code. In the engineering unit, configure the compile option to compilation code, debug and analyze the compiled executable file.

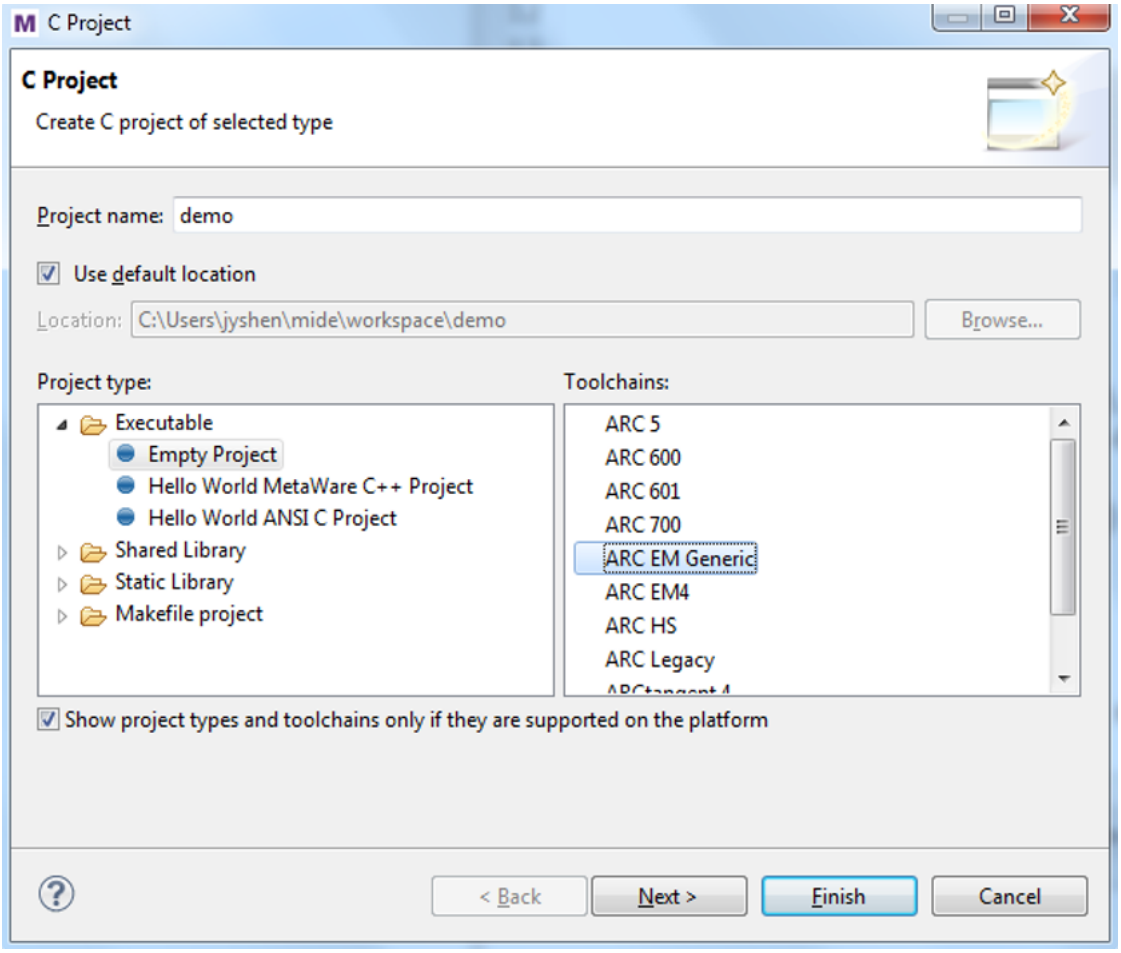
Routine code CoreTest.c:

```
//////////////////////////////////////////////////////////////////////////
// This small demo program finds the data point that is the
// minimal distance from x and y [here arbitrarily defined to be (4,5)]
//
// #define/undefine '_DEBUG' precompiler variable to obtain
// desired functionality. Including _DEBUG will bring in the
// I/O library to print results of the search.
//
// For purposes of simplicity, the data points used in the computations
// are hardcoded into the POINTX and POINTY constant values below
//////////////////////////////////////////////////////////////////////////


#ifdef _DEBUG
#include "stdio.h"
#endif

#define POINTX {1, 2, 3, 4,  5,  6,  7,  8,  9, 10}
#define POINTY {2, 4, 6, 8, 10, 12, 14, 16, 18, 20}
#define POINTS 10

#define GetError(x, y, Px, Py) \
    ( (x-Px)*(x-Px) + (y-Py)*(y-Py) )

int main(int argc, char* argv[]) {
    int pPointX[] = POINTX;
    int pPointY[] = POINTY;

    int x, y;
    int index, error, minindex, minerror;

    x = 4;
    y = 5;

    minerror = GetError(x, y, pPointX[0], pPointY[0]);
    minindex = 0;

    for(index = 1; index < POINTS; index++) {
            error = GetError(x, y, pPointX[index], pPointY[index]);

            if (error < minerror) {
                    minerror = error;
                    minindex = index;
            }
        }

#ifdef _DEBUG
    printf("minindex = %d, minerror = %d.\n", minindex, minerror);
    printf("The point is (%d, %d).\n", pPointX[minindex], pPointY[minindex]);
    getchar();
#endif

    return 0;
    }
```

Steps

Establishing a project

Open the MetaWare IDE, create an empty project called `demo`, and select the ARC EM series processor, as shown below (figure 1).

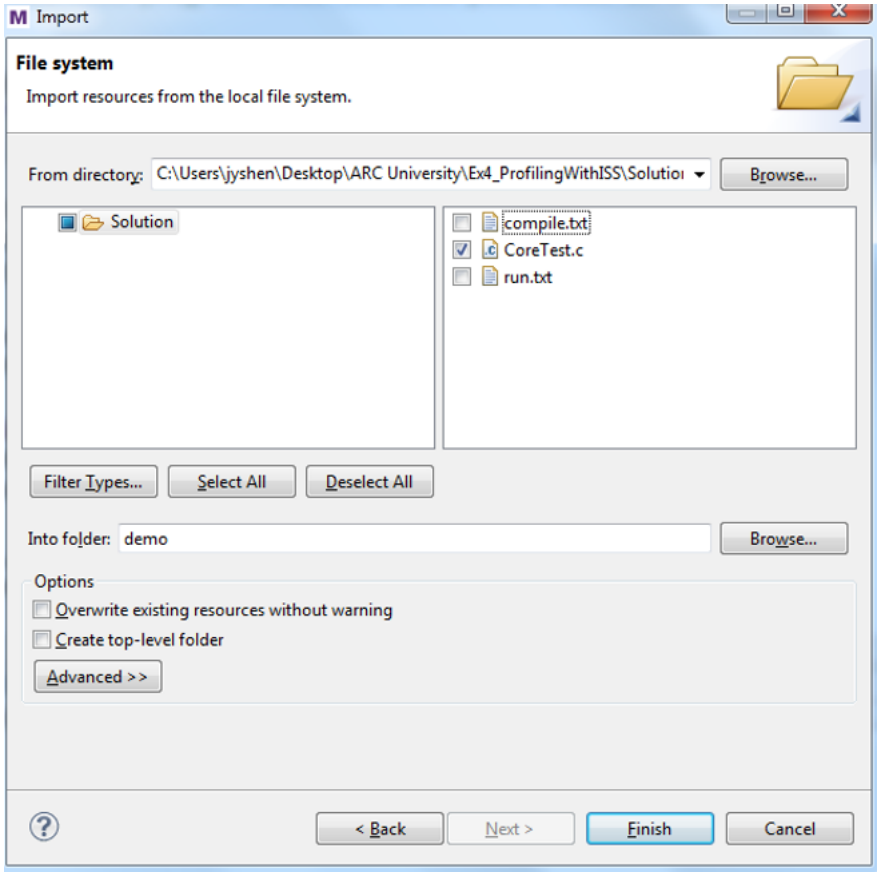


Import the code file CoreTest.c to the project demo.

In the Project Explorer on the left side of the MetaWare IDE main interface, click the icon 

and select Import from the pop-up menu.

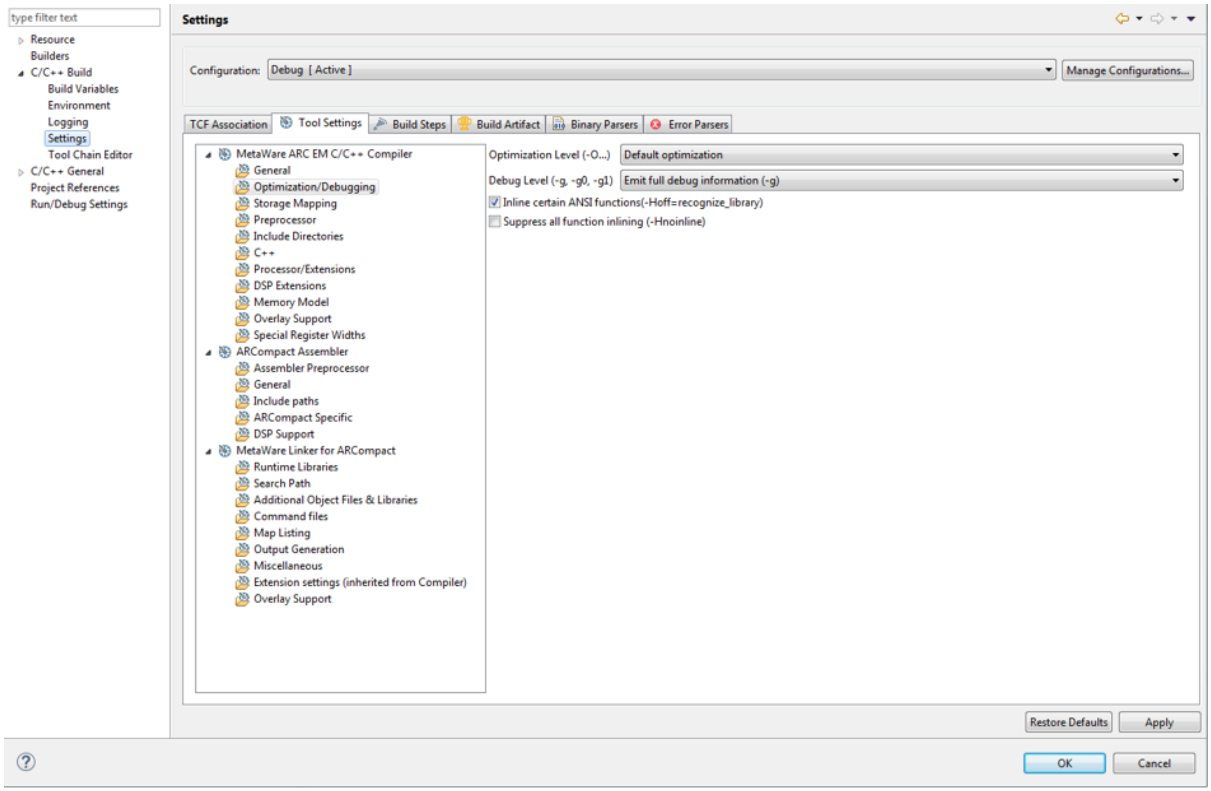
At this point, a dialog called Import appears, select the File System item in the General tab, and then click next. As shown in the figure below, add the file directory where the source code CoreTest.c is located. The dialog box will automatically display the name of the directory and the file name of the file contained in the directory. Select the file to be added, CoreTest.c, and click Finish to complete the entire import process (figure2).



After the import is complete, you can see the code file CoreTest.c you just added in the Project Explorer on the left side of the MetaWare IDE main interface.

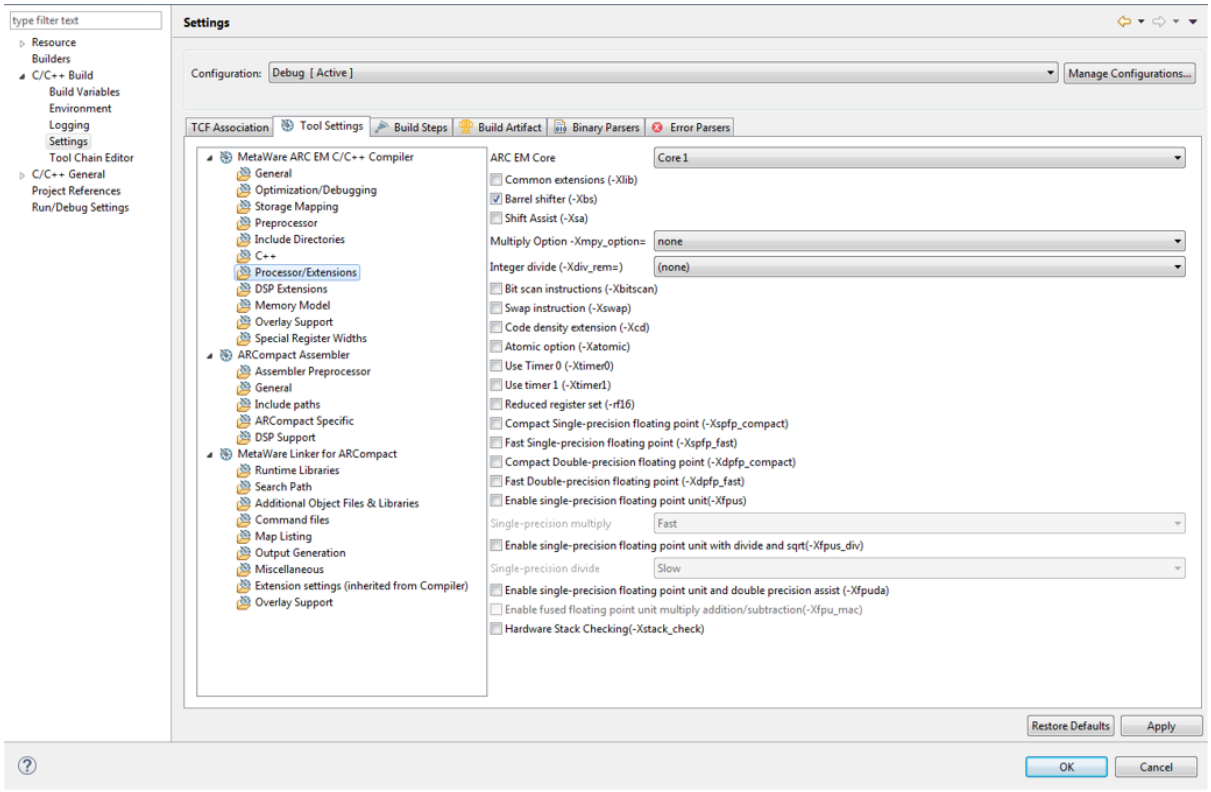
Set compilation options

Right click on the current project demo and select Properties in the popup tab. Click C/C++ Build, settings, Tool Settings to open the compile option settings page, as shown below (figure3).



In the current interface, select Optimization/Debugging to set the compiler optimization and debugging level. For example, set the optimization level to turn off optimization, and set the debugging level to load all debugging information.

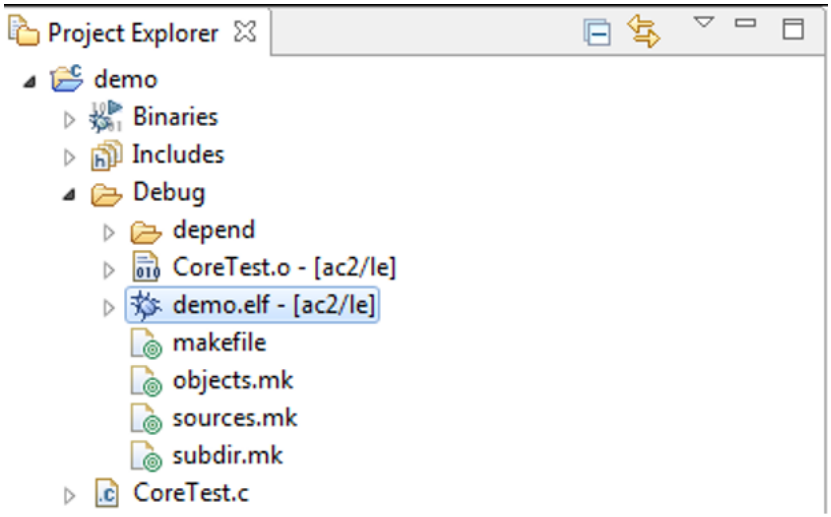
Select Processor/Extensions on the current interface to set the compile options corresponding to the target processor hardware properties, such as the version of the processor, whether to support extended instructions such as shift, multiplication, floating-point operations, etc., whether to include Timer0/1. As shown in the figure below, this setting indicates that the target processor supports normal extended instructions (figure4).



Finally select MetaWare ARC EM C/C++ and check the settings compile options in the All options column on the right. Then click OK to close the Properties dialog.

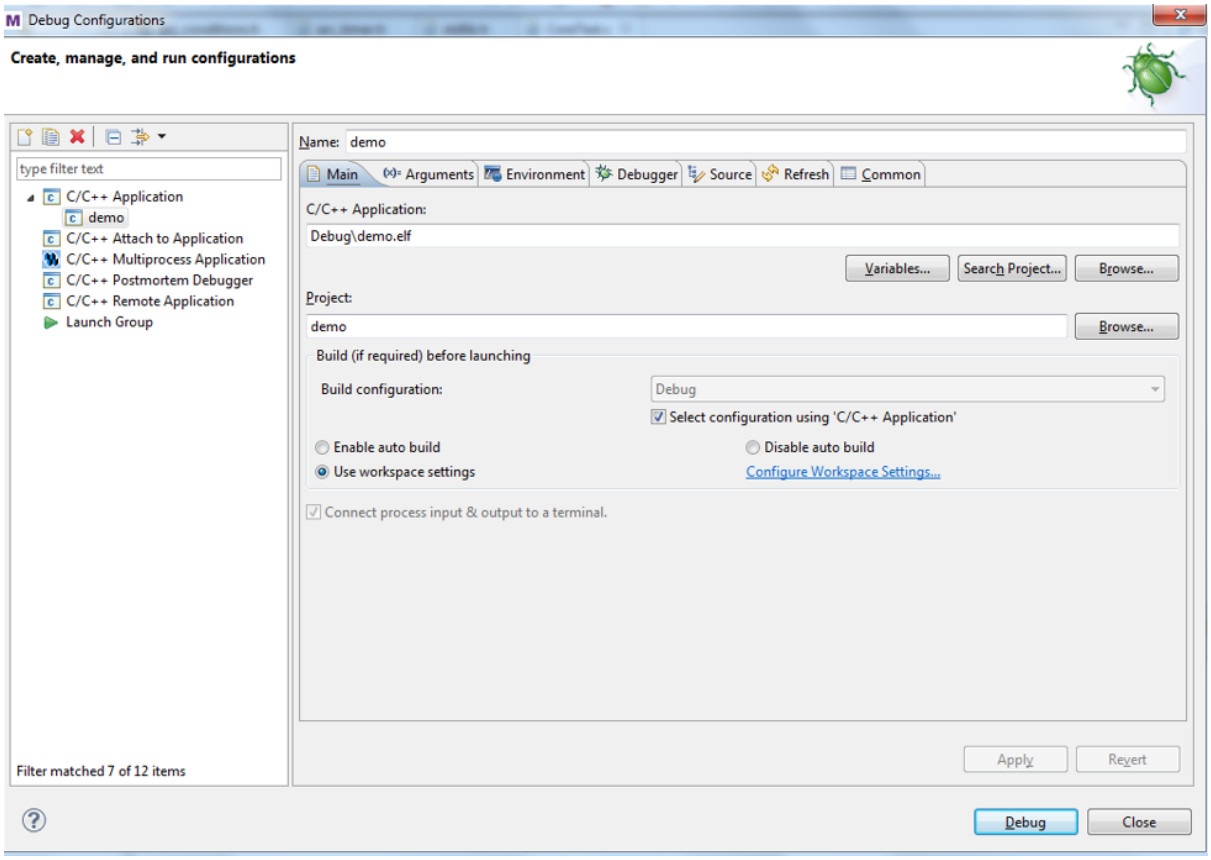
Compile project demo

Select Build Project from the Project drop-down menu in the MetaWare IDE main menu or click the icon  . In the middle of the MetaWare IDE main interface, select the Console tab to view the logs during compilation. When the message 'Finished building target: demo.elf' appears, the compilation is successful, and the compiled executable file demo.elf can be seen in the Project Explorer on the left side of the MetaWare IDE main interface, as shown in the following figure (figure5).



Set debug options

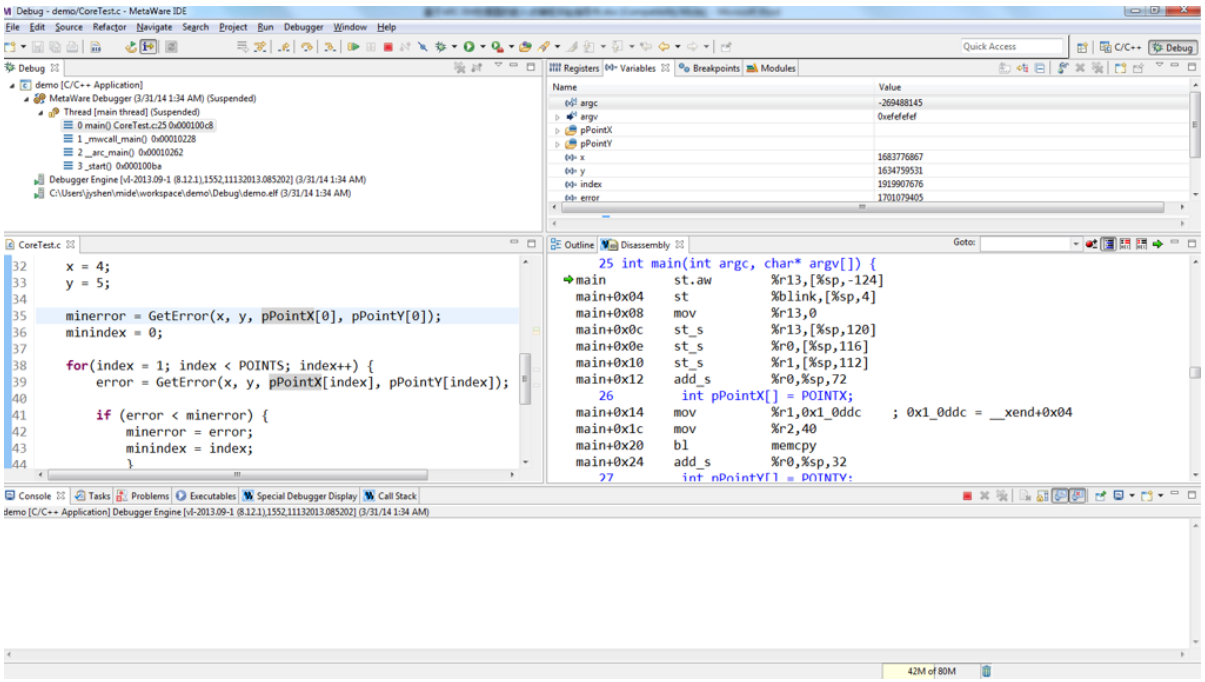
Select Debug Configurations from the Run drop-down menu in the MetaWare IDE main menu. Then double-click on C/C++ Application or right-click on New to get a dialog similar to the one below (figure6).



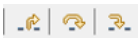
Click Debugger in the right tab, generally do not need to make any changes, finally check the contents of the bottom Debugger Options, click Debug to enter the debugging interface.

Debug executable file demo.elf

First, select the required debug window in the pull-down menu Debugger in the main menu of the debug interface, such as source code window, assembly code window, register window, global variable window, breakpoint window, function window, etc., as shown in the following figure (figure7).

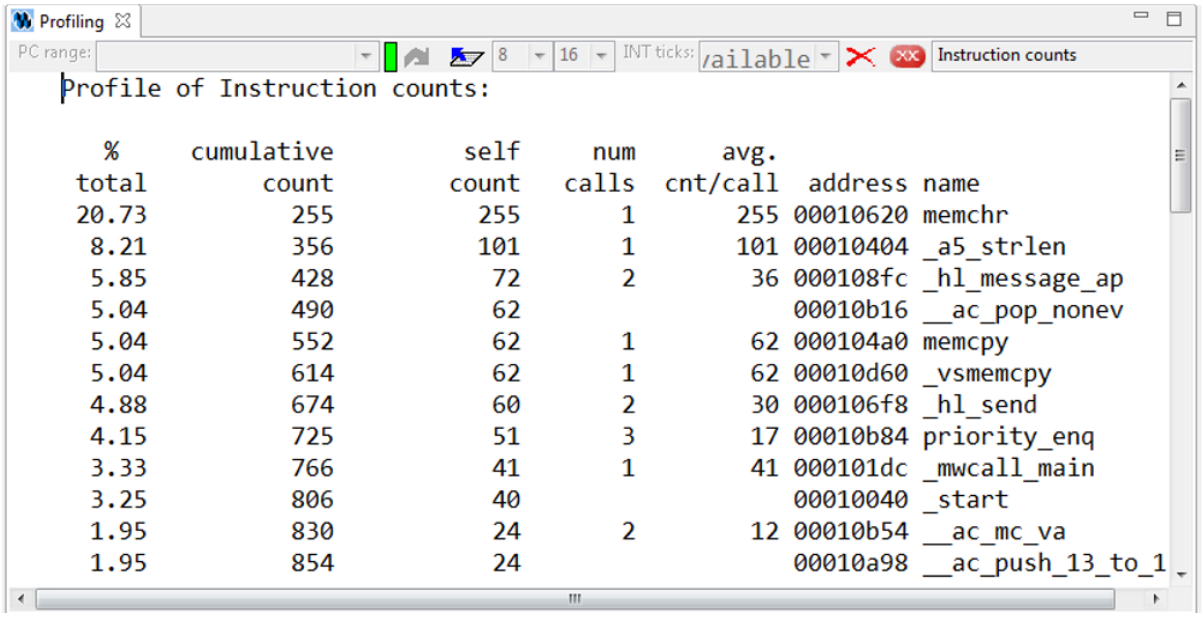


In the C code window, right-click the code line number on the left side of the window, select Toggle Breakpoint in the pop-up menu or double-click the line number to set a breakpoint on the current line. In the assembly code window, double-click a line of code to set a breakpoint on the current line.

Once the breakpoint is set, click the icon  to run the program. After that, the program will run directly to the nearest breakpoint. At this point, you can observe the current program execution and the relevant status information of the processor through the various windows called in the previous step. If you want to know more about the details of program execution and the instruction behavior of the processor, you can use the following three execution commands  to perform single-step debugging. The icon  can choose to step through a C language statement or an assembly instruction to match the status information of each window. It can be very convenient for program debugging. If you want to end the current debugging process, click the icon  . And if you want to return to the main MetaWare IDE page, click C/C++ in the upper right corner icon  .

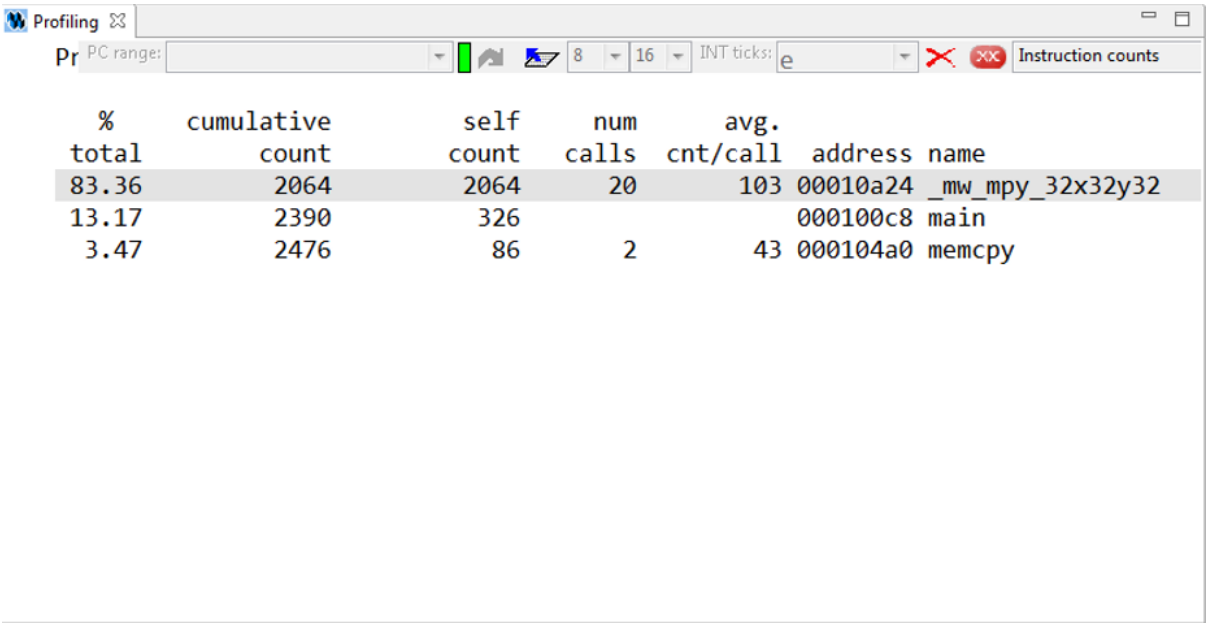
Code performance analysis using the debugger

Based on the previous project demo, open the Compile Options dialog in step 3 and set the Optimization Level to -O0 in the Optimization/Debugging column. Then click  to recompile the project, then click  to enter the debugging interface. Click Debugger in the main menu of the debugging interface, select Disassembly in the pop-up drop-down menu, open the disassembly code window, and you can see that the program is paused at the entrance of the main() function. In the same way, select Profiling in the Debugger drop-down menu, open the performance analysis window and click the icon  in the window, as shown below (figure8).



The Profiling window displays the corresponding of the number of executed instructions of the program with each function under the current debug window. From left to right, the total number of executions of function instructions in the total number of executions of the entire program instruction, the total number of executions of the accumulated instructions, the total number of executions of the functions, the number of times the function is called, the number of including functions, the address of the function, and the name of the function. Through the relationship between the instruction information and the function in the Profiling window, it is very convenient to analyze the program efficiency and find the shortcoming of the program performance.

Let's take this project as an example to continue to introduce the use of the Profiling window. At this point, the program is paused at the entrance of the main() function and the Profiling window opens as shown above. The main() function is the main object of performance analysis optimization. At this time, the content displayed in the Profiling window is actually some function information initialized by the processor before the main() function is executed. Click the icon  in the Profiling window to clear the current information. If you click the icon  again, nothing will be displayed, And it indicate that the cleaning is successful. Then, set a breakpoint at the last statement of the main() function (either C statement or assembly statement), and click the icon  in the toolbar above the debug interface to let the program execute to the breakpoint. Next, click on the icon  in the Profiling window again, and only the information related to the main() function will be displayed, as shown below. Therefore, flexible setting of breakpoints, combined with the clear function, can perform performance analysis on the concerned blocks (figure9).



It can be seen that the multiplication library function _mw_mpy_32x32y32 in the main() function is called 20 times, and a total of 2064 instructions are executed, while the main() function itself executes only 326 instructions, and the memcpy function executes 86 instructions.It can be seen that the implementation of the multiplication function of the program consumes a large number of instructions, and the large number of instructions means that the processor will spend a large number of computation cycles to perform multiplication operations. Therefore, multiplication is the shortcoming of current program performance. If you want to improve the performance of the program, you should first consider how you can use fewer instructions and implement multiplication more efficiently.

Exercises

How can I implement multiplication more efficiently with fewer instructions? Apply this method to the project demo of the fifth part, analyze it with the debugger's Profiling function, observe the total number of instructions consumed by the main function, and compare it with the previous Profiling result of Figure 8.

Note: The expand multiply instruction

ARC GNU TOOLCHAIN

Purpose

- Learn the ARC GNU IDE integration interface
- Familiar with the ARC GNU IDE interface and command line usage
- Familiar with the functions and usage of the ARC GNU IDE debugger

Equipment

PC, ARC GNU IDE software, nSIM simulator, core_test source code in embAR OSP package

Content

Create a C project using the ARC GNU IDE graphical interface, import the routine code CoreTest.c, configure compilation options to compile, and generate executable files.

Start the ARC GNU IDE debugger to enter the debug mode, from the C language and assembly language different perspectives, use set breakpoints, single-step execution, full-speed execution and other functions, combined with observation of PC address, register status, global variable status and Profiling Performance analysis window, analysis of the debug target program.

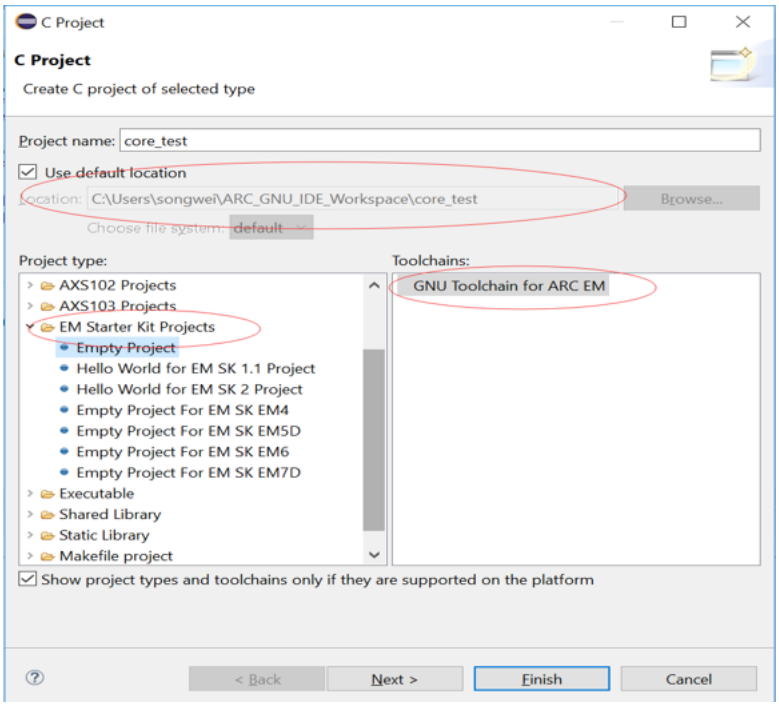
Principles

Use the ARC GNU IDE integrated development environment to create projects and load routine code. In the engineering unit, configure the compile option compilation routine code to debug and analyze the compiled executable file.

Steps

Establishing a project

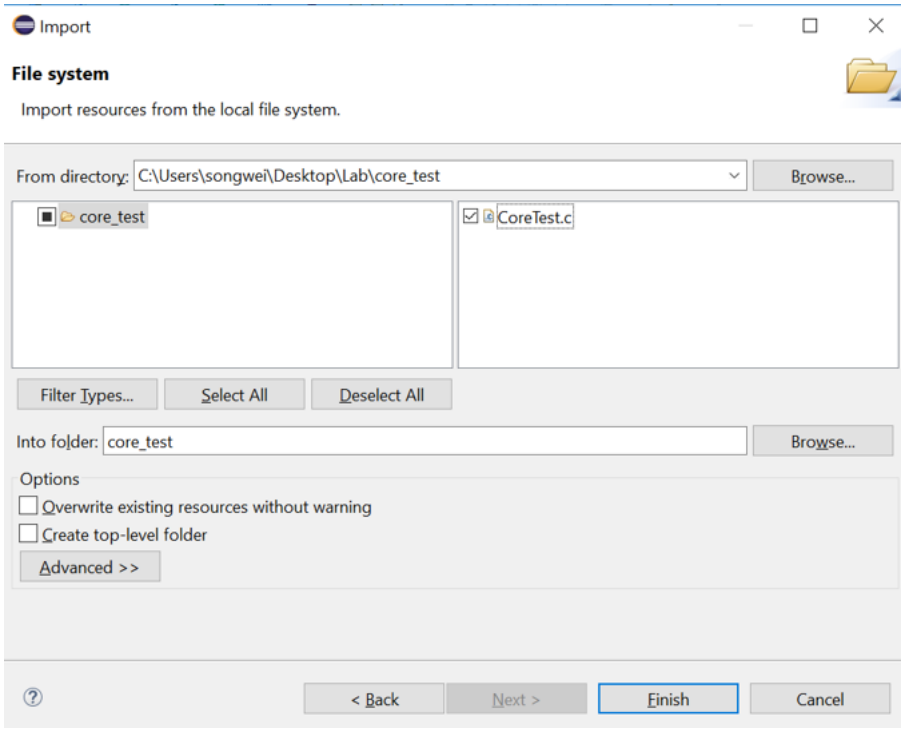
Open the ARC GNU IDE, create an empty project called core_test, and select the ARC EM series processor, as shown below (figure10).



Import the code file CoreTest.c to the project demo

Right click on the icon core_test in the Project Explorer on the left side of the ARC GNU IDE main interface, then select Import from the popup menu.

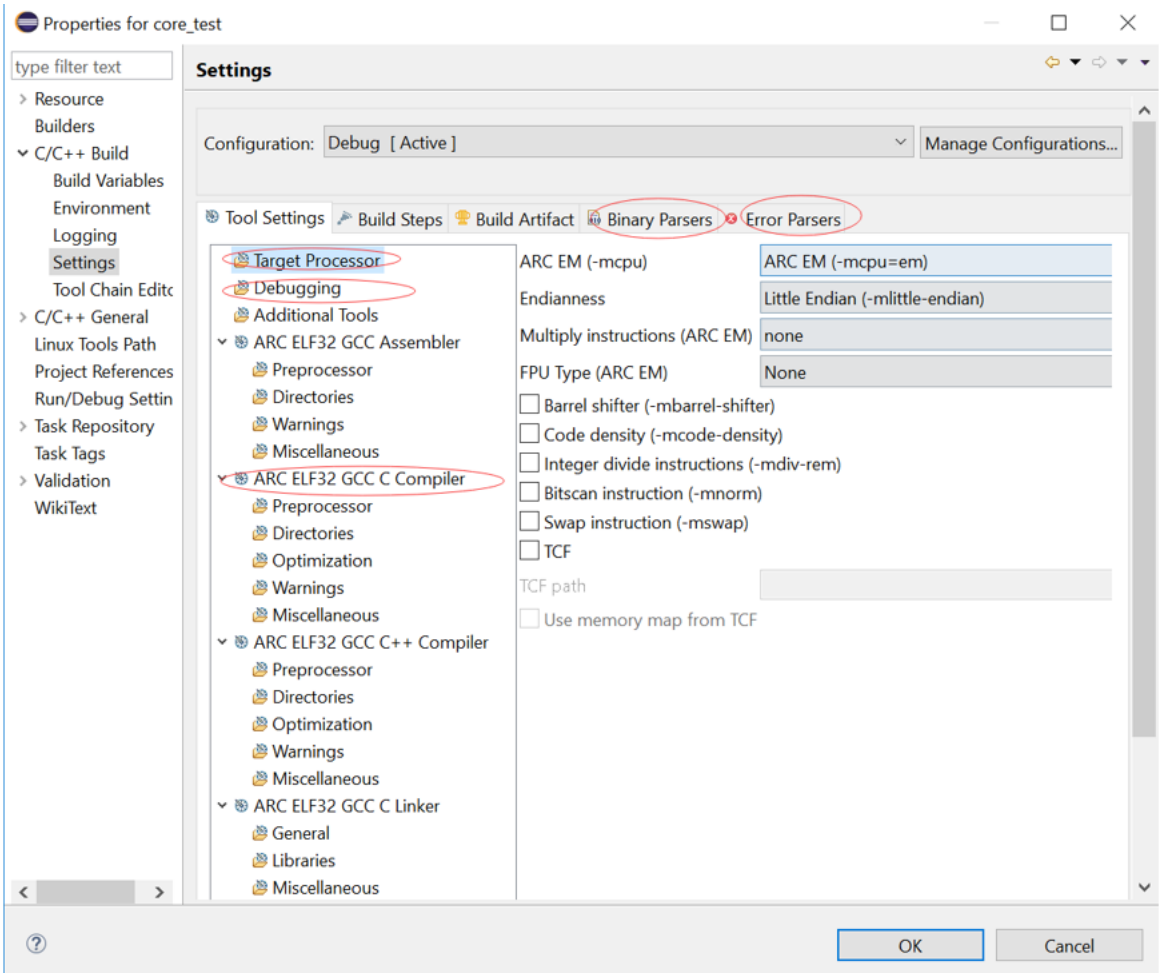
At this point, a dialog called Import appears, select the File System item in the General tab, and then click next. As shown in the figure below (figure11), add the file directory where the source code CoreTest.c is located. The dialog box will automatically display the name of the directory and the file name of the file contained in the directory. Select the file to be added, CoreTest.c, and click Finish to complete the entire import process.



After the import is complete, you can see the code file CoreTest.c that you just added in the Project Explorer on the left side of the ARC GNU IDE main interface.

Set compilation options

Right click on the current project core_test and select Properties in the popup tab. Click C/C++ Build, settings, Tool Settings to open the compile option settings page, as shown below (figure12).



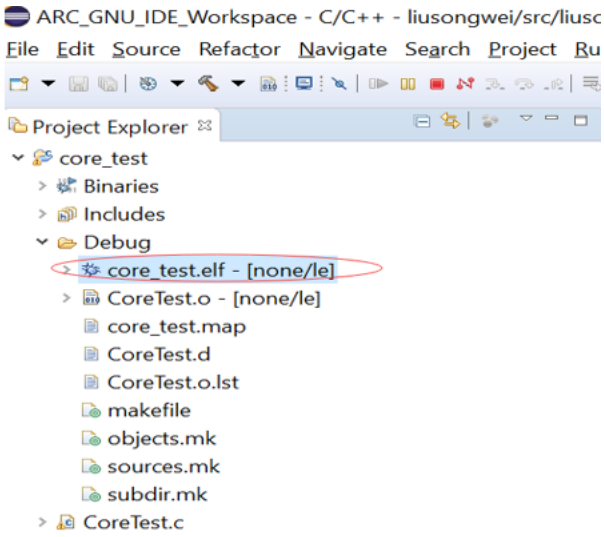
In the current interface, select Debugging to set the compiler optimization and debugging level. For example, set the optimization level to off optimization, and the debugging level is to load all debugging information.

Select Processor in the current interface to set the compile options corresponding to the target processor hardware attributes, such as the version of the processor, whether to support extended instructions such as shift, multiplication, floating-point operations, etc., whether to include Timer0/1.

In step 1, we have already built the project using the engineering template of EMSK, so the corresponding necessary options have been set by default. If there is no special requirement, check the setting compile options in the All options column on the right. Then click OK to close the Properties dialog.

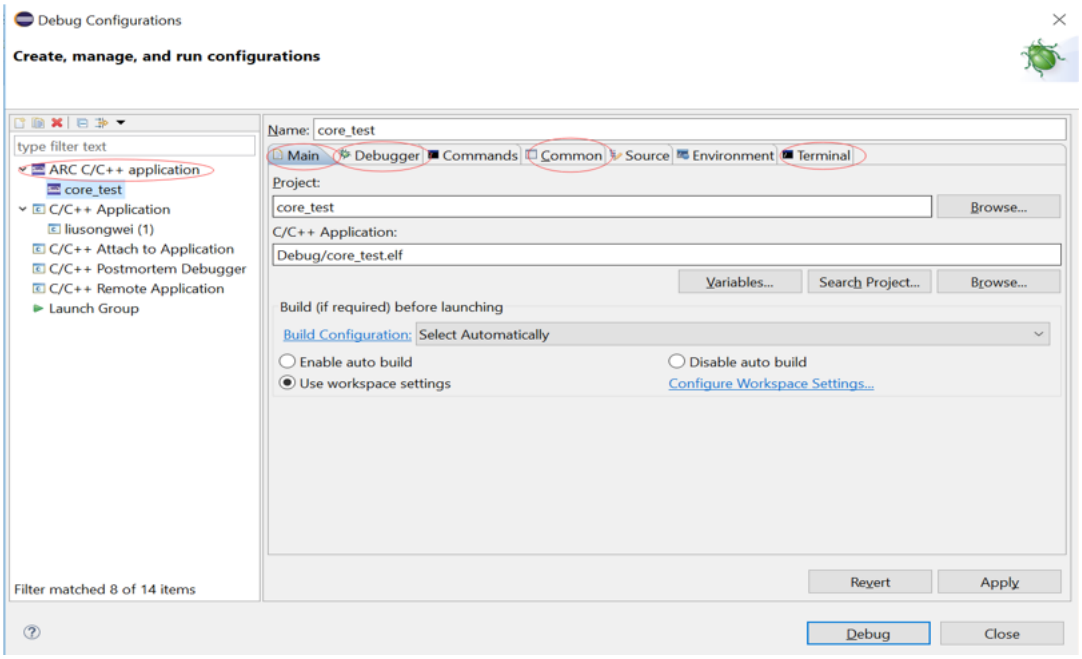
Compile the project core_test

Select Build Project from the Project drop-down menu in the ARC GNU IDE main menu or click the icon . In the middle of the ARC GNU IDE main interface, select the Console tab to view the logs during the compilation process. When the message 'Finished building target: Core_test.elf' appears, the compilation is successful, and the compiled executable file Core_test.elf can be seen in the Project Exporer on the left side of the main interface of the ARC GNU IDE, as shown in the following figure (figure13).



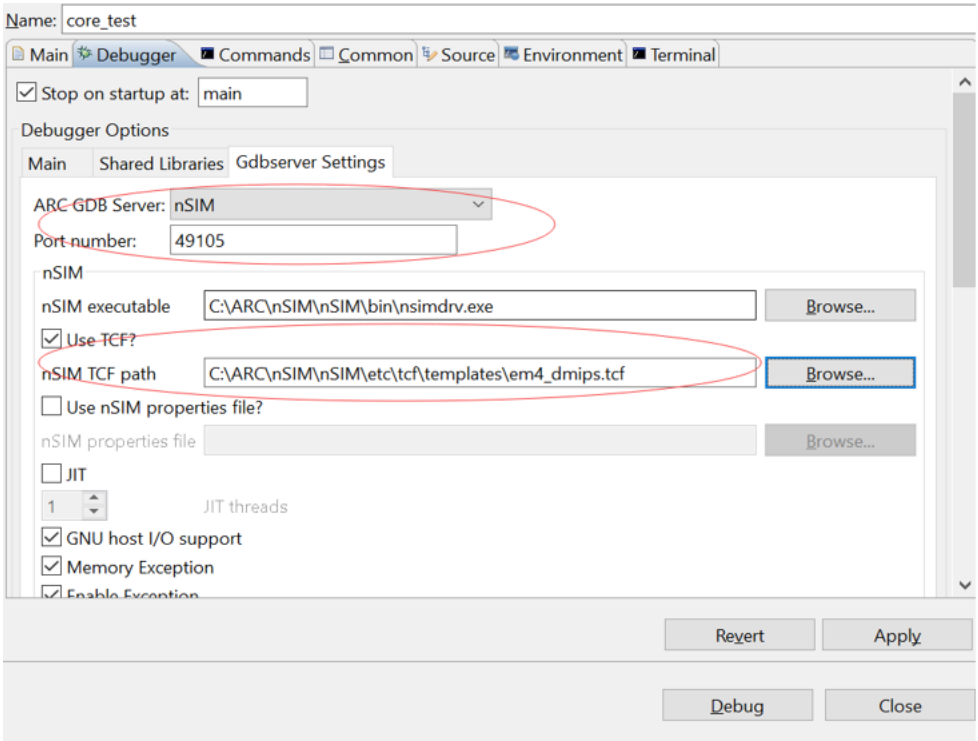
Set debug options

Select Debug Configurations from the Run drop-down menu in the main menu. Then double-click on C/C++ Application or right-click on New to get a dialog similar to the one below (figure14).



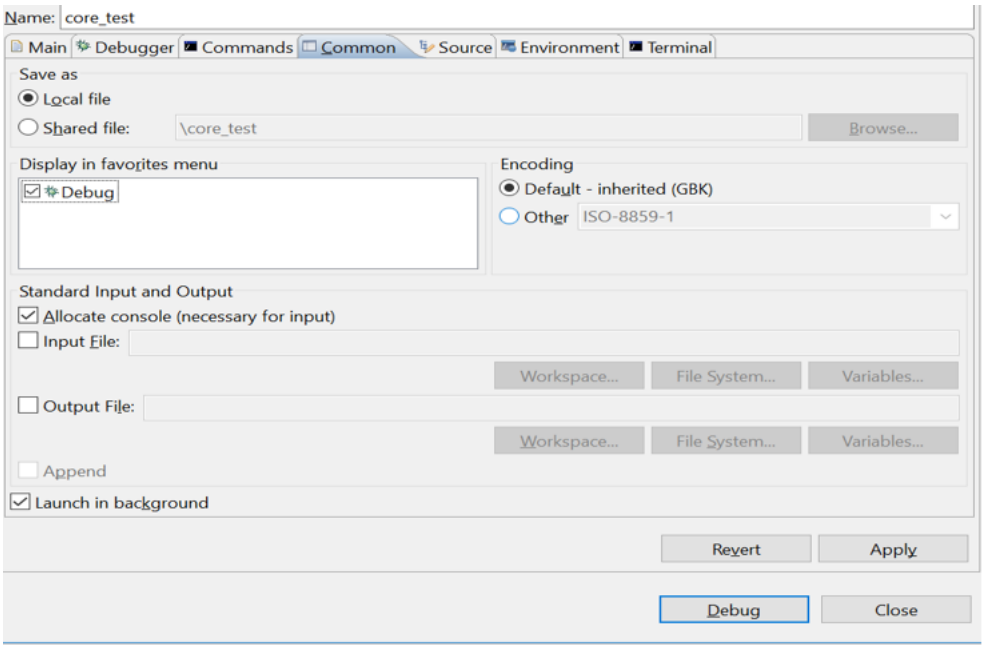
As shown in the figure above, check if the information in Main is correct. Since we use nSIM simulator to simulate EMSK development board, we need to modify the settings of Debugger, Common, and Terminal (this is because nSIM cannot be called directly in GNU IDE. Still need GDB Server for indirect calls). The specific settings are as follows:

- Set Debugger->Gdbsrever Settings



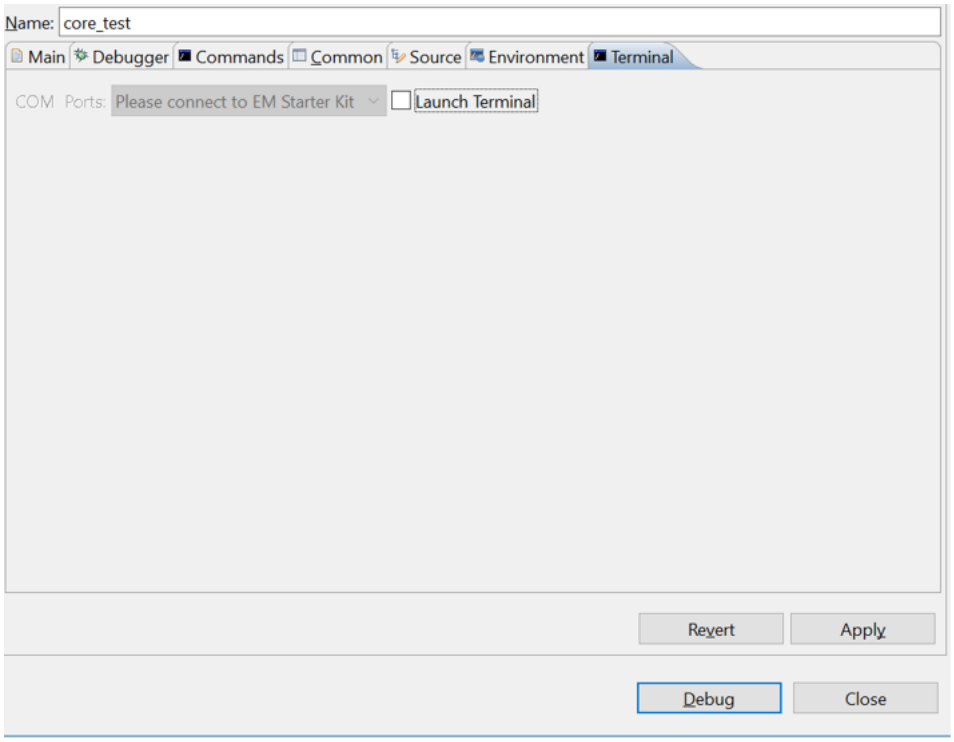
As shown in the above figure (figure15), the ARC GDB Server should select nSIM. At this time, the port number default setting is 49105. It is important to check the Use TCF. Otherwise, the nSIM cannot work normally. The TCF boot file is under *nSIM/nSIM/etc/tcf/templates* (the default installation path). If you have downloaded the Metaware IDE, its own nSIM path is *C:/ARC/nSIM/nSIM/etc/tcf/templates*, and you can select a TCF file in this folder (depending on the version of the board you are simulating and the kernel model), as shown above.

- Pay attention to Debug in Common (figure16)



- Terminal settings

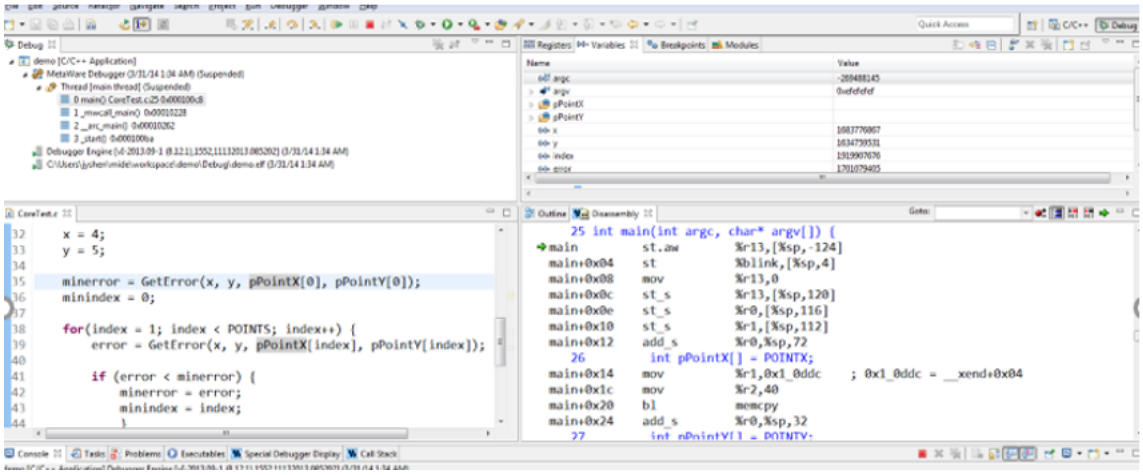
If you are using the EMSK development board, the terminal will automatically select the correct port number, and we are using the emulator without a port, so uncheck it, as show bellow (figure17).



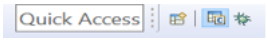
After all settings are completed, click Debug to enter the debugging interface.

Debug executable file core_test.elf

First, select the required debug window in the pull-down menu Debugger in the main menu of the debug interface, such as source code window, assembly code window, register window, global variable window, breakpoint window, function window, etc., as shown in the following figure (figure18).



In the C code window, right-click the code line number on the left side of the window, select Toggle Breakpoint in the pop-up menu or double-click the line number to set a breakpoint on the current line. In the assembly code window, double-click a line of code to set a breakpoint on the current line.

Once the breakpoint is set, click the icon  to run the program. After that, the program will run directly to the nearest breakpoint. At this point, you can observe the current program execution and the relevant status information of the processor through the various windows called in the previous step. If you want to know more about the details of program execution and the instruction behavior of the processor, you can use the following three execution commands  to perform single-step debugging. The icon  can choose to step through a C language statement or an assembly instruction to match the status information of each window and it is very convenient for program debugging. If you want to end the current debugging process, click the icon  . If you want to return to the IDE main page, click C/C++ in the upper right corner icon  .

Code performance analysis using the debugger

Same as the code performance analysis method of MetaWare IDE. For details, please refer to the first part of Experiment 1.

For the use of these two IDEs, you can refer to the Help documentation in the respective IDE, or you can view the online documentation provided by the company.

How to use embARC OSP

Purpose

- To know what embARC OSP is
- To know how to run the provided examples in embARC OSP
- To know how to debug the provided examples in embARC OSP
- To know how to do application development in embARC OSP

Equipment

The following hardware and software tools are required:

- PC host
- ARC GNU toolchain/MetaWare Development Toolkit
- ARC board (EM Starter kit/IoT Development Kit)
- embARC OSP packages

The detailed software requirements of embARC OSP can be found here

Content

First, a brief introduction of embARC OSP will be made and also how to get it. Then, you will get to how to run the provided examples and debug them. Finally, you can try to create your own embARC applications.

Principles

1. IoT OS/Platform

As more and more devices are connected and become more complex, the software running in them are becoming more and more complex.

An IoT OS is an operating system that is designed to perform within the constraints that are particular to Internet of Things devices, including restrictions on memory, size, power and processing capacity. IoT operating systems are a type of embedded OS but by definition are designed to enable data transfer over the internet and more other features.

2. embARC OSP

The embARC Open Software Platform (OSP) is an open software platform to facilitate the development of embedded systems based on DesignWare® ARC® processors.

It is designed to provide a unified platform for DesignWare® ARC® processors users by defining consistent and simple software interfaces to the processor and peripherals, together with ports of several well known FOSS embedded software stacks to DesignWare® ARC® processors.

For more details, please go to embARC OSP on-line documentation

3. Other platforms

Besides embARC OSP, there are also other IoT platforms:

- Zephyr
- Amazon FreeRTOS

Steps

Get embARC OSP

- git

The embARC OSP source code is hosted in a GitHub repository. There are scripts and such in this repo that you'll need to set up your development environment, and we'll be using Git to get this repo. If you don't have Git installed, see the beginning of the OS-specific instructions below for help.

We'll begin by using Git to clone the repository anonymously.

```
# On Windows
cd %userprofile%
# On Linux
cd ~

git clone https://github.com/foss-for-synopsys-dwc-arc-processors/embarc_osp.git␣
↪embarc_osp
```

Then you will have checked out a copy of the source code to your local machine.

- http download

If you are not familiar with git, you can also try to get the latest release of embarc as a zip from the repository, see release page.

Run the examples

The command line interface is the default interface to use embARC OSP. After getting the embARC OSP package, you need to open a **cmd** console in Winodws / a **terminal** in Linux and cd to the root of embARC osp.

Here, take the **blinky** as an example.

1. go to the **blinky** example folder

```
cd example\baremetal\blinky
```

2. connect your board to PC host, and open the UART terminal with putty/tera term/minicom
3. build and run it with command showing below, here ARC GNU toolchain is selected

```
# For EMSK 2.3
make TOOLCHAIN=gnu BOARD=emsk BD_VER=23 CUR_CORE=arcem11d run
# For EMSK 2.2
make TOOLCHAIN=gnu BOARD=emsk BD_VER=22 CUR_CORE=arcem7d run
# For IoTDK
make TOOLCHAIN=gnu BOARD=iotdk run
```

Note: for EMSK, please make sure the board version (BD_VER) and core configuration (CUR_CORE) match your hardware. You could press configure button (located above the letter "C" of the ARC logo) when bit 3 and bit 4 of SW1 switch is off to run a self-test. By doing so, board infomation will be sent by uart and shown on your uart terminal.

---

4. Get the results

- For EMSK, you can see the on-board leds start to blink when the download is successful.
- For IoTDK, as it doesn't have usable leds except some status leds, you will the output log from UART instead.

```
led out: ff, ff
led out: ff00, ff
led out: ff, ff
led out: ff00, ff
led out: ff, ff
led out: ff00, ff
led out: ff, ff
led out: ff00, ff
led out: ff, ff
led out: ff00, ff
led out: ff, ff
led out: ff00, ff
led out: ff, ff
led out: ff00, ff
led out: ff, ff
.................
```

Debug the examples

Still take the **blinky** as example, to debug it, you need to run the following commands:

```
# For emsk 2.3
make TOOLCHAIN=gnu BOARD=emsk BD_VER=23 CUR_CORE=arcem11d gui
# For emsk 2.2
make TOOLCHAIN=gnu BOARD=emsk BD_VER=22 CUR_CORE=arcem7d gui
# For IoTDK
make TOOLCHAIN=gnu BOARD=iotdk gui
```

For Metaware toolchain, the mdb (MetaWare debugger) is used and it's a GUI interface. You can refer the Metaware toolchain use manual for details.

For ARC GNU toolchain, the command line based gdb is used. You need to have some basic knowledge of gdb debug.

Create your own application

Here, it's your turn to create your own application in embARC OSP, taking a well-known simple `hello world` as an example.

- Goals
  - Baremetal application based on embARC OSP
  - Hardware: EMSK 2.2 - ARC EM7D Configuration / IoTDK
  - Print "Hello world from embARC" through UART at 115200 bps
  - Use GNU toolchain to running and debugging in the command line

1. Creat a folder named `hello_world` under `embarc/example/baremetal`.
2. Copy the makefile template `example/example.makefile` and `main.c.tmpl` into `hello_world` folder and rename `example.makefile` to `makefile`, rename `main.c.tmpl` to `main.c`.
3. Change the configurations in makefile according to your hardware configuration and application.
   - Change the application name: change the value of `APPL` to `helloworld`.

- Change the board name: change the value of BOARD to emsk / iotdk. This option can also be given in cmd line. If not specified, the default value will be emsk
- Change the board version: change the value of BD_VER to 22 (for emsk) or 10 (for iotdk).This option can also be given in cmd line. If not specified, the default value will be 22 for board emsk.
- Change the core configuration: change the value of **CUR_CORE** to **arcem7d** This option can also be given in cmd line. If not specified, the default value will arcem7d for board emsk and version 22. For iotdk, **CUR_CORE** can be bypassed as iotdk only has one core configuration.
- Change the embARC OSP root: change the value of EMBARC_ROOT to ../../... EMBARC_ROOT can be relative path or an absolute path.
- Add the middleware that you need for this application: Change the value of MID_SEL.
  - The value of MID_SEL must be the folder name in <embARC>/middleware, such as common or lwip.
  - If using lwip, ntshell, fatfs, and common, set MID_SEL to lwip ntshell fatfs common.
  - Set it to common in the "HelloWorld" application.
- Change your toolchain: change the value of TOOLCHAIN to gnu.
- Update source folders and include folder settings.
  - Update the C code folder settings: change the value of APPL_CSRC_DIR to . APPL_CSRC_DIR is the C code relative path to the application folder
  - Update the assembly source-folder settings: change the value of APPL_ASMSRC_DIR.
  - Update the include-folders settings: change the value of APPL_INC_DIR which is the application include path to the application folder (-I).
  - If more than one directory is needed, use whitespace between the folder paths.
- Set your application defined macros: Change the value of APPL_DEFINES.
  - For example, if define APPLICATION=1, set APPL_DEFINES to -DAPPLICATION=1.

Then makefile for hello world application will be like this

```
## embARC application makefile template ##
### You can copy this file to your application folder
### and rename it to makefile.
##

##
# Application name
##
APPL ?= helloworld

##
# Extended device list
##
EXT_DEV_LIST +=

# Optimization level
# Please refer to toolchain_xxx.mk for this option
OLEVEL ?= O2

##
# Current board and core (for emsk)
##
BOARD ?= emsk
BD_VER ?= 22
CUR_CORE ?= arcem7d
```

(continues on next page)

(continued from previous page)

```
##
# Current board and core (for iotdk)
BOARD ?= iotdk
BD_VER ?= 10


##
# Debugging JTAG
##
JTAG ?= usb

##
# Toolchain
##
TOOLCHAIN ?= gnu

##
# Uncomment following options
# if you want to set your own heap and stack size
# Default settings see options.mk
##
#HEAPSZ ?= 8192
#STACKSZ ?= 8192

##
# Uncomment following options
# if you want to add your own library into link process
# For example:
# If you want link math lib for gnu toolchain,
# you need to set the option to -lm
##
#APPL_LIBS ?=

##
# Root path of embARC
##
EMBARC_ROOT = ../..

##
# Middleware
##
MID_SEL = common

##
# Application source path
##
APPL_CSRC_DIR = .
APPL_ASMSRC_DIR = .

##
# Application include path
##
APPL_INC_DIR = .

##
# Application defines
##
APPL_DEFINES =

##
```

(continues on next page)

(continued from previous page)

```
# Include current project makefile
##
COMMON_COMPILE_PREREQUISITES += makefile

### Options above must be added before include options.mk ###
# Include key embARC build system makefile
override EMBARC_ROOT := $(strip $(subst \,/,$(EMBARC_ROOT)))
include $(EMBARC_ROOT)/options/options.mk
```

4. run

- Set your EMSK 2.2 hardware configuration to ARC EM7D (no need to set for iotdk), and connect it to your PC. Open `PuTTY` or `Tera-term`, and connect to the right COM port. Set the baudrate to **115200 bps**.
- Enter `make run` in the command line to run this application.

Exercises

Create your application which is different with **blinky** and **hello_world** in embARC OSP.

ARC features: timer and auxiliary registers

Purpose

- To learn the timer resource of ARC EM processor
- To learn how to use the auxiliary registers to control the timer
- Read the count value of the timer, and implement a time clock by the timer

Equipment

The following hardware and software tools are required:

- PC host
- ARC GNU toolchain/MetaWare Development Toolkit
- ARC board (EM Starter kit/IoT Development Kit)
- embARC OSP package
- `embarc_osp/arc_labs/labs/lab3_timer`

Content

Read the auxiliary registers of ARC EM to get the version and other setting information of the timer resource. As all the EM processors have **Timer0**, we use the **Timer0** in this lab, and write the auxiliary registers to initialize, start and stop the timer. By reading the count value of the timer, we can calculate the execution time of a code block with the count value and the clock frequency.

Principles

Introduction of timer and auxiliary registers

The timers in ARC EM processor

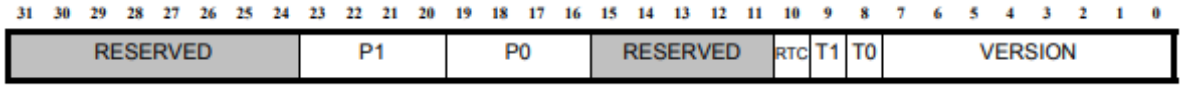
- Two 32-bits programmable timers **Timer0** and **Timer1**
- One 64-bits **RTC**(Real-Time Counter)

All the times are configurable, for example, there are four EM processor cores in **ARC EMSK1.1**, the configuration information are as follow.

Timer	EM4	EM4_16CR	EM4	EM4_16CR
HAS_TIMER0	1	1	1	1
HAS_TIMER1	1	0	1	0
RTC_OPTION	0	0	0	0

The auxiliary register Timer BCR storaged the timer resource information of an EM processor core, the register address of **TIMER_BUILD** is *0x75*.

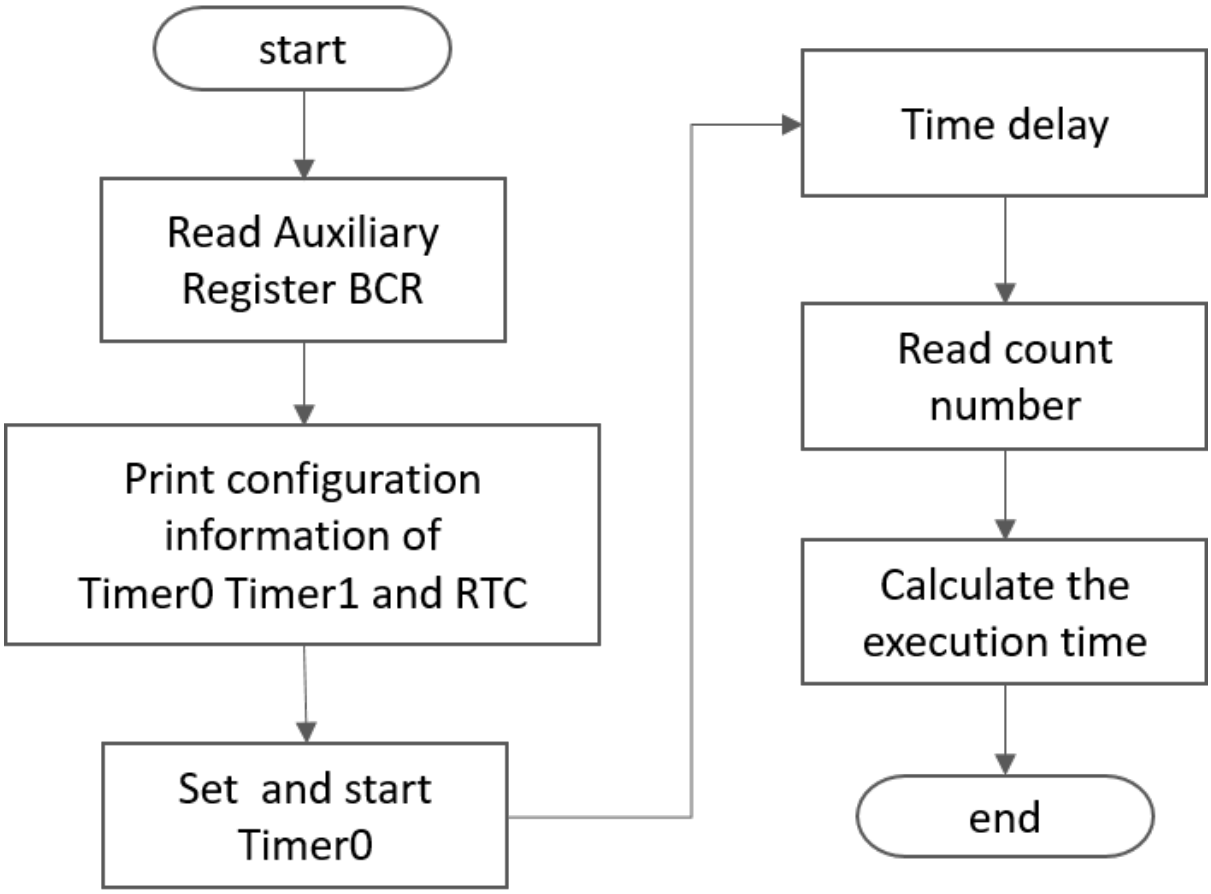
TIMER_BUILD



As we know the timer resources the EM processor configured, we can get the timer's configuration information and control the timers by writing and reading the auxiliary register of that timer. For example, there are the related auxiliary registers of the **Timer0**.

Auxiliary Register	Name	Permission	Description
0x21	COUNT0	RW	Processor timer 0 count value
0x22	CONTROL0	RW	Processor timer 0 control value
0x23	LIMIT0	RW	Processor timer 0 limit value

Program flow chart



Steps

Makefile configuration

There are two ways to do the configuration.

First, configured by compile command, for example:

```
make BOARD=iotdk BD_VER=10 CUR_CORE=arcem9d -j4 TOOLCHAIN=gnu run
```

Second, configured by modifying the makefile. At here, the compile command will be very simple, for example:

```
make -j4 run
```

Open the folder `embarc_osp/arc_labs/labs/lab3_timer`, and open the `makefile`, here is the default configuration.

```
# Application name
APPL ?= lab3_timer

##
# Current Board And Core
##
BOARD ?= iotdk
BD_VER ?= 10
CUR_CORE ?= arcem9d
```

(continues on next page)

(continued from previous page)

```
##
# Set toolchain
##
TOOLCHAIN ?= gnu

#
# root dir of embARC
#
EMBARC_ROOT = ../../..

MID_SEL = common

# application source dirs
APPL_CSRC_DIR = .
APPL_ASMSRC_DIR = .

# application include dirs
APPL_INC_DIR = .
```

- Reconfigure **BOARD** and **CUR_CORE**, in this lab, we use the launch board *iotdk*

```
##
# Current Board And Core
##
BOARD ?= iotdk
BD_VER ?= 10
CUR_CORE ?= arcem9d
```

- Reconfigure **TOOLCHAIN**, select the toolchain *gnu* or *metaware* you used

```
##
# Set toolchain
##
TOOLCHAIN ?= gnu
```

- Reconfigure **EMBARC_ROOT**, make sure the relative path between *embARC OSP* root folder and the *timer* folder is correct.

```
#
# root dir of embARC
#
EMBARC_ROOT = ../../..
```

Main code

Read auxiliary register BCR_BUILD

We can use the function `_arc_aux_read()` to read the auxiliary register for the timer resource information.

Read auxiliary register **TIMER_BUILD**. In the register **TIMER_BUILD** The lower 8 bits indicate the core version information, the bit 9 indicate the **Timer0**, the bit 10 indicate the **Timer1**, the bit 11 indicate the **RTC**. Here is the code:

```
uint32_t bcr = _arc_aux_read(AUX_BCR_TIMERS);
int timer0_flag=(bcr >> 8) & 1;
int timer1_flag=(bcr >> 9) & 1;
int RTC_flag=(bcr >> 10) & 1;
```

Read timer related auxiliary registers, for example, the **Timer0**. Here is the code:

```
EMBARC_PRINTF("Does this timer0 exist?  YES\r\n");
/*Read auxiliary register configuration information*/
EMBARC_PRINTF("timer0's operating mode:0x%08x\r\n",_arc_aux_read(AUX_TIMER0_CTRL));
EMBARC_PRINTF("timer0's limit value :0x%08x\r\n",_arc_aux_read(AUX_TIMER0_LIMIT));
EMBARC_PRINTF("timer0's current cnt_number:0x%08x\r\n",_arc_aux_read(AUX_TIMER0_
↪CNT));
```

Stop-Set-Start the Timer0

We can use the function `_arc_aux_write()` to write the auxiliary register.

To control the **Timer0** with the related auxiliary registers.

- **COUNT0**: write this register to set the initial value of the **Timer0**. It will increase from the set value at anytime you write this register.
- **CONTROL0**: write this register to update the control modes of the **Timer0**.
- **LIMIT0**: write this register to set the limit value of the **Timer0**, the limit value is the value after which an interrupt or a reset must be generated.

In this lab, we should stop timer before setting and starting it, the function `timer_stop()` is already encapsulated in embARC OSP, you can use this function or directly write the register. And then set the timer work mode, enable interrupt or not and set the limit value. At last start the timer. Here is the code:

```
/* Stop it first since it might be enabled before */
_arc_aux_write(AUX_TIMER0_CTRL, 0);
_arc_aux_write(AUX_TIMER0_LIMIT,0);
_arc_aux_write(AUX_TIMER0_CNT, 0);
/* This is a example about timer0's timer function. */
uint32_t mode = TIMER_CTRL_NH;/*Timing without triggering interruption.*/
uint32_t val = MAX_COUNT;
_arc_aux_write(AUX_TIMER0_CNT, 0);
_arc_aux_write(AUX_TIMER0_LIMIT,val);
/* start the specific timer */
_arc_aux_write(AUX_TIMER0_CTRL,mode);
```

When the timer is running, we can read the count value of the timer,and calculate the execution time of a code block. Here is the code:

```
uint32_t start_cnt=_arc_aux_read(AUX_TIMER0_CNT);
/**
 * code block
 */
uint32_t end_cnt=_arc_aux_read(AUX_TIMER0_CNT);
uint32_t time=(end_cnt-start_cnt)/(BOARD_CPU_CLOCK/1000);
```

Compile and debug

- Compile and download

Open `cmd` under the folder *embarc_osp/arc_labs/labs/lab3_timer*, input the compile command as follow:

```
make -j4 run
```

Note: If your toolchain is meteware, you should use `gmake`. If you don't use core configuration specified in makefile, you need to pass all the make options to trigger make command

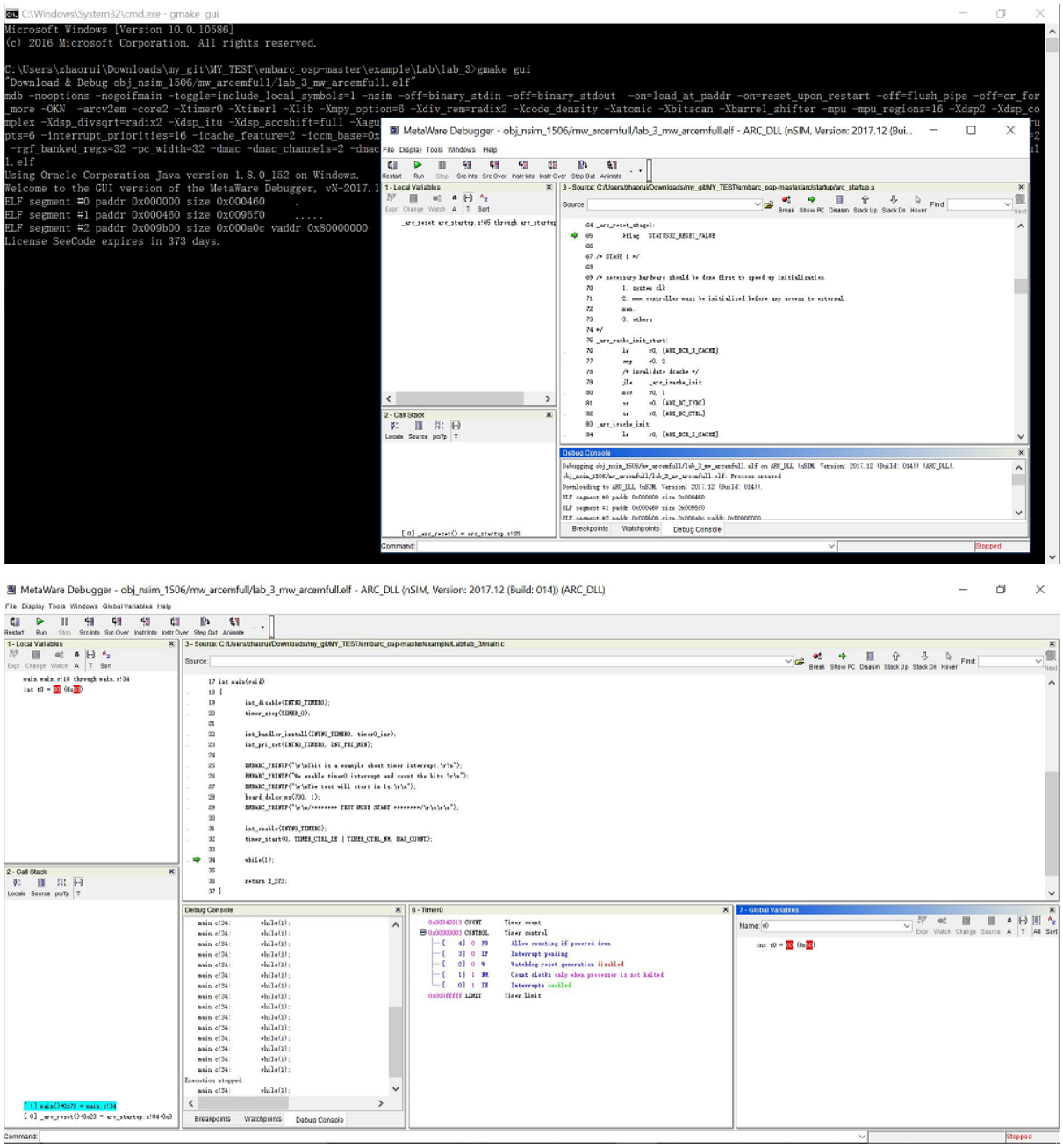
- Output

```
------------------------------------------------------------
      ____                                    _ ____
     |  _ \ ______      ______ _ __ ___  __| | __ ) _   _
     | |_) / _ \ \ /\ / / _ \ '__/ _ \/ _` |  _ \| | | |
     |  __/ (_) \ V  V /  __/ | |  __/ (_| | |_) | |_| |
     |_|   \___/ \_/\_/ \___|_|  \___|\__,_|____/ \__, |
                                                   |___/
                          _       _    ____   ____
           ___ _ __ ___ | |__   / \  |  _ \ / ___|
          / _ \ '_ ` _ \| '_ \ / _ \ | |_) | |
         |  __/ | | | | | |_) / ___ \|  _ <| |___
          \___|_| |_| |_|_.__/_/   \_\_| \_\\____|
     ------------------------------------------------------------

     embARC Build Time: Aug 22 2018, 15:32:54
     Compiler Version: Metaware, 4.2.1 Compatible Clang 4.0.1 (branches/release_40)
     Does this timer0 exist?  YES
     timer0's operating mode:0x00000003
     timer0's limit value :0x00023280
     timer0's current cnt_number:0x0000c236

     Does this timer1 exist?  YES
     timer1's operating mode:0x00000000
     timer1's limit value :0x00000000
     timer1's current cnt_number:0x00000000

     Does this RTC_timer exist?   NO

     The start_cnt number is:2
     /******** TEST MODE START ********/

     This is TEST CODE.

     This is TEST CODE.

     This is TEST CODE.

     /******** TEST MODE END ********/
     The end_cnt number is:16785931
     The board cpu clock is:144000000

     Total time of TEST CODE BLOCK operation:116
```

- Debug

Open `cmd` under the folder *embarc_osp/arc_labs/labs/lab3_timer*, input the command as follow:

```
make gui
```



The debug view will pop up automatically, we can watch the variables and registers.

Exercises

In the debug view, observe and understand the contents of the interrupt vector table.

Note: Click the Memory button in the debug view Debugger drop-down menu to see the contents of the memory in real time.

ARC features: interrupts

Purpose

- To introduce the interrupt module in the ARC embedded processor
- To demonstrate how to use the interrupt and timer APIs already defined in the embARC processor in the program

Equipment

The following hardware and software tools are required:

- PC host
- ARC GNU toolchain/MetaWare Development Toolkit
- ARC board (EM Starter kit/IoT Development Kit)
- embARC OSP package
- `embarc_osp/arc_labs/labs/lab4_interrupt`

Content

This lab and lab 3 are both introductions to the internal characteristics of the ARC processor. Lab 3 introduces the timer. This lab aims to introduce the interrupt of embARC through `embarc_osp/arc_labs/labs/lab4_interrupt` in the embARC OSP package. The two routines give you a preliminary understanding of the ARC interrupt resources.

Principles

The ARC EM processor uses vector interrupts to handle interrupt events. When the interrupt occurs, the processor stops the execution of the current program, and queries the corresponding interrupt vector in the predefined interrupt vector table according to the current interrupt type. In other words, to find the entry address of the interrupt service program. Then program jumps to the address to execute the interrupt service routine. After the execution is completed, return to the interrupted program and complete the response of the interrupt event.

In embARC OSP, we use the `int_handler_install()` function to bind our interrupt function name to the interrupt vector of the corresponding interrupt, and then we can achieve the above functions.

Steps

Open and browse lab one

Go to the `embarc_osp/arc_labs/labs/lab4_interrupt`, where there are two folders, `lab_4_1` and `lab_4_2`.

The `lab_4_1` is more fundamental compared to `lab_4_2`. So we first enter folder `lab_4_1`, in which the precise timing function is implemented through the timer interrupt.

Open main.c and browse the entire program.

```
#include "embARC.h"
#include "embARC_debug.h"

#define COUNT (BOARD_CPU_CLOCK/1000)
```

(continues on next page)

(continued from previous page)

```
    volatile static int t0 = 0;
    volatile static int second = 0;

    /** arc timer 0 interrupt routine */
    static void timer0_isr(void *ptr)
    {
      timer_int_clear(TIMER_0);
      t0++;
    }

    /** arc timer 0 interrupt delay */
    void timer0_delay_ms(int ms)
    {
      t0 = 0;
      while(t0<ms);
    }

    /** main entry for testing arc fiq interrupt */
    int main(void)
    {
      int_disable(INTNO_TIMER0);
      timer_stop(TIMER_0);

      int_handler_install(INTNO_TIMER0, timer0_isr);
      int_pri_set(INTNO_TIMER0, INT_PRI_MIN);

      EMBARC_PRINTF("\r\nThis is a example about timer interrupt.\r\n");
      EMBARC_PRINTF("\r\n/******** TEST MODE START ********/\r\n\r\n");

      int_enable(INTNO_TIMER0);
      timer_start(TIMER_0, TIMER_CTRL_IE | TIMER_CTRL_NH, COUNT);

      while(1)
      {
            timer0_delay_ms(1000);
            EMBARC_PRINTF("\r\n %ds.\r\n",second);
            second ++;
      }
      return E_SYS;
}
```

Sub-module analysis lab one code

The code can be roughly divided into three parts: interrupt service function, main function, and delay function.

Let's analyze each one below:

- Interrupt service function:

```
static void timer0_isr(void *ptr)
{
  timer_int_clear(TIMER_0);
  t0++;
}
```

This code is a standard example of an interrupt service function, enters the service function, clears the interrupt flag bit, and then performs the processing that needs to be done in the interrupt service function. Other interrupt service functions can also be written using this template.

In this function, we incremented the count variable t0 by one.

- Main function

```
    int main(void)
    {
      int_disable(INTNO_TIMER0);
      timer_stop(TIMER_0);

      int_handler_install(INTNO_TIMER0, timer0_isr);
      int_pri_set(INTNO_TIMER0, INT_PRI_MIN);

      EMBARC_PRINTF("\r\nThis is a example about timer interrupt.\r\n");
      EMBARC_PRINTF("\r\n/******** TEST MODE START ********/\r\n\r\n");

      int_enable(INTNO_TIMER0);
      timer_start(TIMER_0, TIMER_CTRL_IE | TIMER_CTRL_NH, COUNT);

      while(1)
      {
            timer0_delay_ms(1000);
            EMBARC_PRINTF("\r\n %ds.\r\n",second);
            second ++;
      }
      return E_SYS;
}
```

The `EMBARC_PRINTF` function in this code is only used to send information to the computer, which can be ignored during analysis.

This code is divided into two parts: initialization and looping.

In the initialization section, we configured the timer and timer interrupts.

Unlike Lab 3, this code uses the embARC OSP API to implement it. In fact, in essence, these two methods are the same. The API just encapsulates the read and write operations of the auxiliary registers for convenience.

First, in order to configure **Timer0** and its interrupts, we need to turn them off first. This work is done by the functions `int_disable` and `timer_stop`.

Then we configure the interrupt service function and priority for our interrupts. This work is done by the functions `int_handler_install` and `int_pri_set`.

Finally, after the interrupt configuration is complete, we need to enable the **Timer0** and interrupts that we previously turned off. This work is done by the functions `int_enable` and `timer_start`. The implementation of the `timer_start` function is basically the same as the reading and writing of the auxiliary registers in our lab_3. Interested students can view them in the file arc_timer.c. One point to note in this step is the configuration of `timer_limit` (the last parameter of `timer_start`). We need to configure the interrupt time to 1ms , so we need to do a simple calculation (the formula is the expression after COUNT).

In this example, the loop body only serves as an effect display. We call our own delay function in the loop body to print the time per second.

Note: Since nSIM is only simulated by computer, there may be time inaccuracy when using this function. Interested students can use the EMSK to program the program in the development board. In this case, the time will be much higher than that in nSIM.

- Delay function

```
static void timer0_isr(void *ptr)
{
  t0 = 0;
  while(t0<ms);
}
```

This code is very simple and the idea is clear. When we enter the function, we clear the global variable t0. Since we have set the interrupt interval to 1ms in the above timer_start, we can think that every time t0 is incremented, the time has passed 1ms.

Then, we wait through the while(t0<ms) sentence, so that we can get the full ms delay with higher precision.

Lab one Labal phenomenon

After the lab one program is successfully downloaded, the serial output is as follows:

```
embARC Build Time: Mar 16 2018, 09:58:46
Compiler Version: Metaware, 4.2.1 Compatible Clang 4.0.1

This is an example about timer interrupt
/********TEST MODE START********/
0s

1s

2s

3s

4s

5s

...
```

Open and browse the lab two

We then enter `lab_4_2`, which mainly shows the working state of priority and interrupt nesting.

Open main.c and browse through the entire program.

```
#include "embARC.h"
#include "embARC_debug.h"

#define MAX_COUNT 0xfffff

volatile static uint8_t timer_flag = 0;
volatile static uint8_t hits = 0;

volatile static uint8_t nesting_flag = 1;

/** arc timer 0 interrupt routine */
static void timer0_isr(void *ptr)
{
    timer_int_clear(TIMER_0);

    timer_flag = 0;

    board_delay_ms(10, 1);

    if(timer_flag)
    {
            EMBARC_PRINTF("Interrupt nesting!\r\n");
    }
    else
    {
```

(continues on next page)

(continued from previous page)

```
            EMBARC_PRINTF("Interrupt\r\n");
    }

    hits++;
}

/** arc timer 1 interrupt routine */
static void timer1_isr(void *ptr)
{
    timer_int_clear(TIMER_1);

    timer_flag = 1;
}

/** main entry for testing arc fiq interrupt */
int main(void)
{
    timer_stop(TIMER_0);
    timer_stop(TIMER_1);

    int_disable(INTNO_TIMER0);
    int_disable(INTNO_TIMER1);

    int_handler_install(INTNO_TIMER0, timer0_isr);
    int_pri_set(INTNO_TIMER0, INT_PRI_MAX);

    int_handler_install(INTNO_TIMER1, timer1_isr);
    int_pri_set(INTNO_TIMER1, INT_PRI_MIN);

    EMBARC_PRINTF("\r\nThe test will start in 1s.\r\n");
    EMBARC_PRINTF("\r\n/******** TEST MODE START ********/\r\n\r\n");

    int_enable(INTNO_TIMER0);
    int_enable(INTNO_TIMER1);

    timer_start(TIMER_0, TIMER_CTRL_IE | TIMER_CTRL_NH, MAX_COUNT);
    timer_start(TIMER_1, TIMER_CTRL_IE | TIMER_CTRL_NH, MAX_COUNT/100);

    while(1)
    {
            if((hits >= 5)&&(nesting_flag == 1))
            {
                    timer_stop(TIMER_0);
                    timer_stop(TIMER_1);

                    int_disable(INTNO_TIMER0);
                    int_disable(INTNO_TIMER1);

                    int_pri_set(INTNO_TIMER0, INT_PRI_MIN);
                    int_pri_set(INTNO_TIMER1, INT_PRI_MAX);

                    nesting_flag = 0;

                    int_enable(INTNO_TIMER0);
                    int_enable(INTNO_TIMER1);

                    timer_start(TIMER_0, TIMER_CTRL_IE | TIMER_CTRL_NH, MAX_COUNT);
                    timer_start(TIMER_1, TIMER_CTRL_IE | TIMER_CTRL_NH, MAX_COUNT/
↪100);
            }
            else if((hits >= 10)&&(nesting_flag == 0))
```

(continues on next page)

(continued from previous page)

```
            {
                    timer_stop(TIMER_0);
                    timer_stop(TIMER_1);

                    int_disable(INTNO_TIMER0);
                    int_disable(INTNO_TIMER1);

                    int_pri_set(INTNO_TIMER0, INT_PRI_MAX);
                    int_pri_set(INTNO_TIMER1, INT_PRI_MIN);

                    hits = 0;
                    nesting_flag = 1;

                    int_enable(INTNO_TIMER0);
                    int_enable(INTNO_TIMER1);

                    timer_start(TIMER_0, TIMER_CTRL_IE | TIMER_CTRL_NH, MAX_COUNT);
                    timer_start(TIMER_1, TIMER_CTRL_IE | TIMER_CTRL_NH, MAX_COUNT/
↪100);
            }
    }
    return E_SYS;
}
```

Sub-module analysis lab two code

Lab two seems complicated, but it is very simple. The code for Lab two only needs to be divided into two parts: the interrupt service function and the main function.

- Interrupt service function

```
static void timer0_isr(void *ptr)
{
  timer_int_clear(TIMER_0);

  timer_flag = 0;

  board_delay_ms(10, 1);

  if(timer_flag)
  {
          EMBARC_PRINTF("Interrupt nesting!\r\n");
  }
  else
  {
          EMBARC_PRINTF("Interrupt\r\n");
  }

  hits++;
}

static void timer1_isr(void *ptr)
{
  timer_int_clear(TIMER_1);

  timer_flag = 1;
}
```

First, in order to analyze the code, we first ignore the extraneous parts (such as EMBARC_PRINTF, delay and hits in if).

In this case, we can find that for the interrupt service function timer0_isr, it is impossible to have the timer_flag of 1 only when it is itself. The only way to do this is to have another higher priority interrupt between timer_flag=0 and if statement set it.

Following this line of thought, let's look at timer1_isr again, and sure enough.

Regarding EMBARC_PRINTF, it is used to indicate the status.

Regarding the delay, its role is to lengthen this period of time, making nesting more likely.

Regarding hits, it will be mentioned in the main function module.

- main function

```
int main(void)
{
timer_stop(TIMER_0);
timer_stop(TIMER_1);

int_disable(INTNO_TIMER0);
int_disable(INTNO_TIMER1);

int_handler_install(INTNO_TIMER0, timer0_isr);
int_pri_set(INTNO_TIMER0, INT_PRI_MAX);

int_handler_install(INTNO_TIMER1, timer1_isr);
int_pri_set(INTNO_TIMER1, INT_PRI_MIN);

EMBARC_PRINTF("\r\nThe test will start in 1s.\r\n");
EMBARC_PRINTF("\r\n/******** TEST MODE START ********/\r\n\r\n");

int_enable(INTNO_TIMER0);
int_enable(INTNO_TIMER1);

timer_start(TIMER_0, TIMER_CTRL_IE | TIMER_CTRL_NH, MAX_COUNT);
timer_start(TIMER_1, TIMER_CTRL_IE | TIMER_CTRL_NH, MAX_COUNT/100);

while(1)
{
        if((hits >= 5)&&(nesting_flag == 1))
        {
                timer_stop(TIMER_0);
                timer_stop(TIMER_1);

                int_disable(INTNO_TIMER0);
                int_disable(INTNO_TIMER1);

                int_pri_set(INTNO_TIMER0, INT_PRI_MIN);
                int_pri_set(INTNO_TIMER1, INT_PRI_MAX);

                nesting_flag = 0;

                int_enable(INTNO_TIMER0);
                int_enable(INTNO_TIMER1);

                timer_start(TIMER_0, TIMER_CTRL_IE | TIMER_CTRL_NH, MAX_COUNT);
                timer_start(TIMER_1, TIMER_CTRL_IE | TIMER_CTRL_NH, MAX_COUNT/100);
        }
        else if((hits >= 10)&&(nesting_flag == 0))
        {
                timer_stop(TIMER_0);
                timer_stop(TIMER_1);

                int_disable(INTNO_TIMER0);
```

(continues on next page)

(continued from previous page)

```
                int_disable(INTNO_TIMER1);

                int_pri_set(INTNO_TIMER0, INT_PRI_MAX);
                int_pri_set(INTNO_TIMER1, INT_PRI_MIN);

                hits = 0;
                nesting_flag = 1;

                int_enable(INTNO_TIMER0);
                int_enable(INTNO_TIMER1);

                timer_start(TIMER_0, TIMER_CTRL_IE | TIMER_CTRL_NH, MAX_COUNT);
                timer_start(TIMER_1, TIMER_CTRL_IE | TIMER_CTRL_NH, MAX_COUNT/100);
        }
}
return E_SYS;
}
```

The main function looks very long, but in fact there is a considerable part of it that is repetitive (we can also build a small function to make the code look more concise).

In the first lab, we have already discussed the configuration of the timer and the creation of the interrupt, we will not repeat them here.

The main function is simple: when the interrupt of timer0 occurs 5 times, change the priority relationship of the two interrupts. The hits mentioned earlier are count variables to assist in the above functions.

Lab two Labal phenomenon

The labal phenomenon of Lab two is shown in the figure.

“Interrupt nesting!” indicates that interrupt nesting has occurred, and “Interrupt” indicates that it has not occurred.

For a better understanding, let’s go back and look at the priority settings in the main function.

It is easy to see that when the timer0 interrupt priority is low (INT_PRI_MAX is low priority, this setting is contrary to most people’s intuition), the timer1 interrupt can be embedded therein; when the timer0 interrupt priority is high, the timer1 interrupt cannot be embedded.

To summarize, high-priority interrupts can interrupt low-priority interrupts, and low-priority interrupts can be embedded by high-priority interrupts. The Main function can be understood as the lowest priority interrupt.

```
embARC Build Time: Mar 16 2018, 09:58:46
Compiler Version: Metaware, 4.2.1 Compatible Clang 4.0.1

This test will start in 1s.

/********TEST MODE START********/

Interrupt  nesting!
Interrupt  nesting!
Interrupt  nesting!
Interrupt  nesting!
Interrupt  nesting!
Interrupt
Interrupt
Interrupt
Interrupt
Interrupt
Interrupt  nesting!
Interrupt  nesting!
```

(continues on next page)

(continued from previous page)

```
Interrupt  nesting!
Interrupt  nesting!
Interrupt  nesting!
Interrupt
Interrupt
Interrupt
```

Exercises

Try using an interrupt other than a timer to write a small program. (For example, try to implement a button controled LED using GPIO interrupt)

How to use ARC board

Purpose

Equipment

Content

Principles

Steps

Exercises

A simple bootloader

Purpose

- Understand the memory map of ARC boards
- Understand the principles of bootloader and self-booting
- Understand the usage of shell commands in cmd
- Create a self-booting application

Equipment

The following hardware and software tools are required:

- PC host
- ARC GNU toolchain/MetaWare Development Toolkit
- ARC board (EM Starter kit/IoT Development Kit)
- SD card
- embARC OSP package
- `example/baremetal/bootloader`

Content

1. Build and run the `example/baremetal/bootloader`
2. Download the generated `bootloader.bin` into flash
3. Build a self-boot applicaiton and boot it from SD card
4. Use the ntshell commands

Principles

Memory Map of ARC board

EM Starter Kit

The available memory regions of EM Starter Kit are shown below.

Table 1: Memory Map of EM Starter Kit

Name	Start address	Size
on-chip ICCM	0x00000000	256/128 KB
on-chip DCCM	0x80000000	128 KB
on-board DDR RAM	0x10000000	128 MB

In this lab, the last 1 MB of DDR is reserved for the simple bootloader, other memory regions are available for application.

IoT Development Kit

The available memory regions of IoT Development Kit are shown below.

Table 2: Memory Map of IoT Development Kit

Name	Start address	Size
on-chip eflash	0x00000000	256 KB
external boot SPI flash	0x10000000	2 MB
on-chip ICCM	0x20000000	256 KB
on-chip SRAM	0x30000000	128 KB
on-chip DCCM	0x80000000	128 KB
on-chip XCCM	0xC0000000	32 KB
on-chip YCCM	0xE0000000	32 KB

In this lab, on-chip eflash and on-chip SRAM are reserved for the simple bootloader, CCMs are reserved for application.

Boot of ARC board

EM Starter Kit

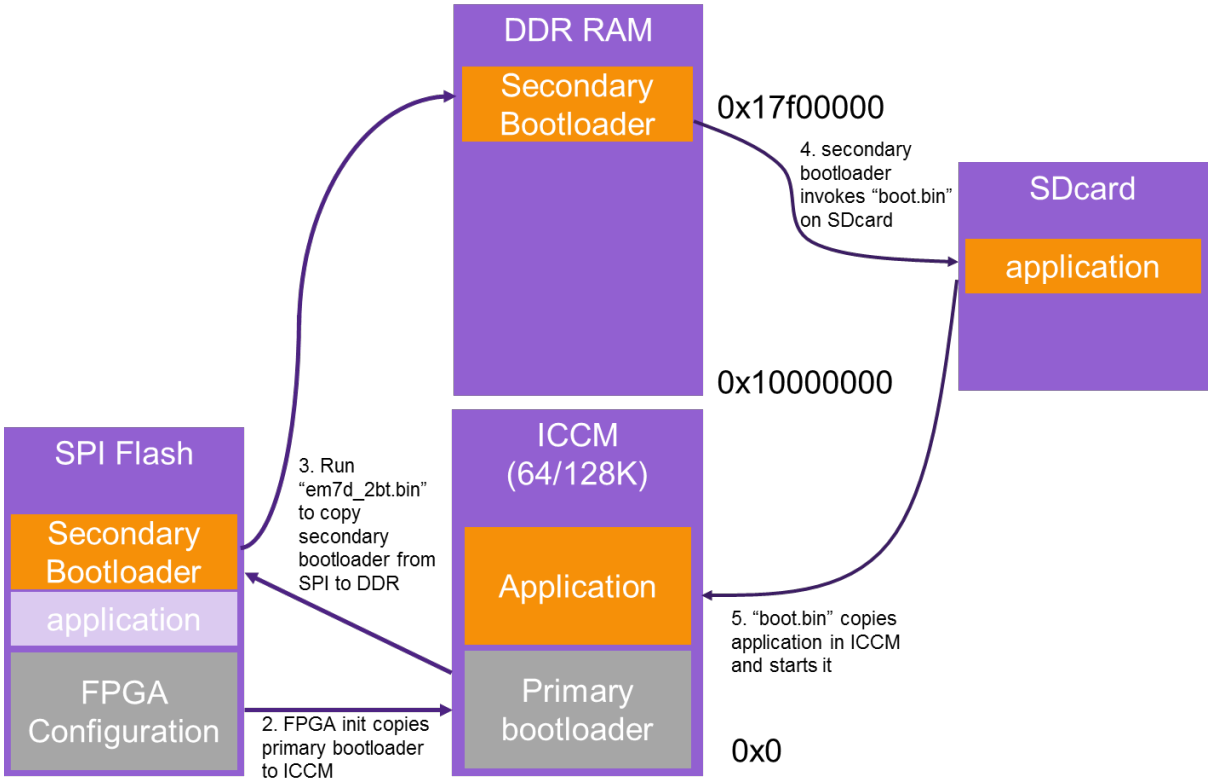
The EMSK uses a Xilinx SPARTAN-6 FPGA part which can be configured to run different members of the ARCv2 EM Processor family. The EMSK includes a SPI flash pre-programmed with four FPGA configurations of ARC EM cores.

When a "power on" or reset/configure is issued, the FPGA will auto-load one of the pre-installed FPGA configurations from SPI flash. After the FPGA configuration is loaded from the SPI flash, a simple primary bootloader

is loaded in ICCM. Through the primary bootloader, an application can be loaded from SPI Flash into ICCM or external DDR memory.

Considering that the SPI Flash is used to store FPGA images, the secondary bootloader is designed based on the primary bootloader to load an application from an SD card since it can be read and written easily. The startup sequence is listed below:

1. Power on or reset event.
2. Load FPGA configuration from the SPI flash.
3. Run primary bootloader, which loads the secondary bootloader from the SPI Flash into main memory (DRAM).
4. Run secondary bootloader from main memory to load application from the SD card into ICCM/DRAM memory.
5. Run the application from ICCM/DRAM memory.



IoT Development Kit

IoT Development Kit can boot from on-chip eflash and extern boot SPI eflash, which is decided by the FWU switch of IOTDK. When this switch is set of "off", the processor starts executing the program stored in on-chip eflash; When this switch is set of "on", the processor starts executing the program stored in external boot SPI eflash;

How to flash the ARC board

EM Starter Kit

IoT Development Kit

Simple bootloader

This simple bootloader is designed to work as a secondary/simple bootloader for embARC, it will load boot.hex or boot.bin on SDCard and run that program. And this example itself can be used as ntshell application.

The following features are provided in this simple bootloader:

- Boot application from SD card
- File operations on SD card
- UART Y-modem protocol to update application
- Operations on ARC processors

Steps

1. Build and run the `example/baremetal/bootloader`
2. Download the generated `bootloader.bin` into flash
3. Build a self-boot applicaiton and boot it from SD card
4. Use the ntshell commands

Exercises

1. Create and build a different self-boot embARC application
2. Use the ntshell commands
3. Use the UART-ymodem to load your application

3.2.2 Level 2 Labs

Compiler Optimizations

Part 1. Prerequisites

Before starting using ARC DSP the following prerequisites are required:

- Have MetaWare tools for Windows installed
  https://www.synopsys.com/dw/ipdir.php?ds=sw_metaware
- Known how to create, edit, build and debug projects in MetaWare IDE
- Have IOT Design Kit (IOTDK) board and Digilent USB drivers (Digilent Adept 2) installed and tested
  http://store.digilentinc.com/digilent-adept-2-download-only
- IOTDK board is EM9D based

The following needs to be tested before starting this lab:

- Connecting IOTDK board to computer

- Connecting serial console (PuTTY) to IOTDK COM port (For information on how to do initial board setup and configuration please refer to *Getting Started* chapter of *ARC IOT Design Kit User Guide* that came along with IOTDK board).

Part 2. Lab Objectives

Use MetaWare compiler options to optimize regular C code to use DSP extensions as well as try direct usage of DSP extensions through intrinsic etc.

Part 3. Lab principle and method

This section describes compiler options in MetaWare used in this lab.

To optimize code to use DSP extensions two sets of compiler options will be used throughout the lab, DSP Extensions options and optimization level.

DSP Extensions Options

We will use EMBAC OSP build system to build software. The details can be found in EMBARC OSP document page. Here is the example command. You can pass extra compiler/liner options by ADT_COPT/ADT_LOPT.

```
gmake BOARD=emsk BD_VER=23 CUR_CORE=arcem9d TOOLCHAIN=mw gui
ADT_COPT="-Hfxapi -Xdsp2" OLEVEL=O2
```

Options that are used in the lab are:

- `-Xdsp[1/2]`:

  Enable DSP instructions

- `-Xdsp_complex, -Xdsp_divsqrt`:

  Enable complex arithmetic DSP, divide & sqrt instructions

- `-Xdsp_ctrl[=up|convergent,noguard|guard, preshift|postshift]`:

  Fine-tune the compiler's assumptions about the rounding, guard bit and fractional product shift behavior

- `-Hdsplib`: Link in the DSP library

  For programming ARC fixed-point DSP in C and C++

  Contains functions to carry out DSP algorithms such as filtering & transforms

- `-Hfxapi`: Use the Fixed Point API support library

  Used with `-Xdsp`. Provides low level intrinsic support for EM DSP instructions.

  Programs written using this API execute natively on an ARC EM processor with DSP extensions and can also be emulated on x86 Windows hosts.

- `-Xxy`: Specifies that XY memory is available

  Used with `-Xdsp2`. Enables optimization for XY memory

- `-Xagu_small, -Xagu_medium, -Xagu_large`:

  Enables AGU, and specifies its size. Note IOTDK has small AGU

Note: Because ARC is configurable processor, different cores can contain different extensions on hardware level. Thus options set for compiler should match underlying hardware. On the other hand if specific hardware future is present in the core but compiler option is not set, it won't be used effectively, if used at all. IOTDK Core default options are presented in Appendix A.

Optimization level

MetaWare compiler enables to set optimization level, which enables or disables different optimization techniques include in the compiler. You pass the optimization option to gmake by "OLEVEL=O2".

The lowest level is the default -O0, which does little optimization to the compiled assembly code, which can be used for debugging, because in un-optimized assembly code all source code commands will have 1:1 representation. On the other hand -O3 highest level optimization highly modifies output assembly code to make it smaller and fast, but debugging such a code is harder, because it is not close match with source code. Also, high level of optimization requires longer compilation time, which for large project can be significant, if many compilation iterations are to be made.

Optimization for DSP extensions

A regular code without direct usage of DSP extensions can be optimized to use DSP extensions wherever applicable, which compiler can do automatically with DSP extension options corresponding to hardware are set and high level of optimization is selected.

Checking options

Options are specified in the makefile or command line, as shown in the previous section.

Part 4. Optimizing code

An example code below contains a function called "test" which contains a 20 step for loop and a multiply accumulate operation done manually.

```
#include <stdio.h>

short test(short *a, short *b) {
    int i;

    long acc = 0;
    for(i = 0; i < 10; i++)
            acc += ( ((long)(*a++)) * *b++) <<1 ;

    return (short) (acc);
}

short a[] = {1,2,3,4,5, 6,7,8,9,10};
short b[] = {11,12,13,14,15, 16,17,18,19,20};

int main(int argc, char *argv[]) {

    short c = test(a,b);

    printf("result=%d",c);

    return 0;
}
```

Step 1. Compiling without DSP extensions

Set optimization level "-O0", and no DSP extensions (unchecking -Xdsp1, -Xdsp2).

After compilation open disassembly window and check assembly code for function "test".

Below is the list options used when launching gmake:

```
OLEVE=O0 ADT_COPT="-arcv2em -core1 -Xlib -Xtimer0 -Xtimer1"
```

You can use the following command to generate disassembly code:

```
elfdump -T -S
<your_working_director>/obj_iotdk_10/mw_arcem9d/lab1_mw_arcem9d.elf
```

Notice assembly code in the disassembled output. See how many assembly instruction are used for each line, for example for loop spends several instruction to calculate loop variable value and check whether to stop.

```
   29 short test(short *a, short *b) {
test          sub_s         %sp,%sp,16
test+0x02     st_s          %r0,[%sp,12]
test+0x04     st_s          %r1,[%sp,8]
   32          long acc = 0;
test+0x06     mov_s         %r0,0
test+0x08     st_s          %r0,[%sp]
test+0x0a     st_s          %r0,[%sp,4]
   33          for(i = 0; i < 10; i++)
test+0x0c     ld_s          %r0,[%sp,4]
   33          for(i = 0; i < 10; i++)
test+0x0e     cmp_s         %r0,9
test+0x10     bgt_s         0xdc = test+0x34 = basic.c!36
   34                  acc += ( ((long)(*a++)) * *b++) << 1;
test+0x12     ld_s          %r0,[%sp,12]
test+0x14     add_s         %r1,%r0,2
test+0x16     st_s          %r1,[%sp,12]
test+0x18     ldh_s.x       %r0,[%r0]
test+0x1a     ld_s          %r1,[%sp,8]
test+0x1c     add_s         %r2,%r1,2
test+0x1e     st_s          %r2,[%sp,8]
test+0x20     ldh_s.x       %r1,[%r1]
test+0x22     mpy_s         %r0,%r0,%r1
test+0x24     asl_s         %r0,%r0
test+0x26     ld_s          %r1,[%sp]
test+0x28     add_s         %r0,%r0,%r1
test+0x2a     st_s          %r0,[%sp]
test+0x2c     ld_s          %r0,[%sp,4]
test+0x2e     add_s         %r0,%r0,1
test+0x30     st_s          %r0,[%sp,4]
test+0x32     b_s           0xb4 = test+0x0c = basic.c!33
```

Step 2. Compiling without DSP extensions, with -O2

Compile with:

```
OLEVE=O2 ADT_COPT="-arcv2em -core1 -Xlib -Xtimer0 -Xtimer1"
```

Adding optimization level -O2, optimizes out many of the instructions:

```
   29 short test(short *a, short *b) {
test          mov_s         %r2,0
test+0x02     mov           %lp_count,10
test+0x06     lp            0xbe = test+0x16 = basic.c!36
   34                  acc += ( ((long)(*a++)) * *b++) << 1;
test+0x0a     ldh.x.ab      %r3,[%r1,2]
test+0x0e     ldh.x.ab      %r12,[%r0,2]
test+0x12     mpyw_s        %r3,%r3,%r12
test+0x14     add1_s        %r2,%r2,%r3
   36          return (short) ((acc+0x8000)>>16);
test+0x16     add           %r0,%r2,0x8000
test+0x1e     j_s.d         [%blink]
test+0x20     asr_s         %r0,%r0,16
.0+0x2c       nop_s
   47          return 0;
➪main         j_s.d         [%blink]        ; _mwcall_main+0x6e
```

In this code it is easy to find zero-delay loop ("lp" command) which acts as for loop. Note that multiply-accumulate is done with separate "mpyw_s" and "add1_s" instructions.

Step 3. Compiling with DSP extensions

Compile with:

```
OLEVE=O3 ADT_COPT="-arcv2em -core1 -Xlib -Xtimer0 -Xtimer1 -Xdsp1"
```

Adding -Xdsp1 (optimization level changed to -O3) helps compiler to optimize away "mpyw_s" and "add1_s" instructions and replace them with hardware dual-16bit SIMD multilication "vmpy2h". Notice the loop count is now 5.

```
3: short test(short *a, short *b) {
 13c: 244a7140                mov     %lp_count,5
 140: 244a1000                mov     %r12,0
4:      int i;
5:
6:      long acc = 0;
7:      for(i = 0; i < 10; i++)
 144: 20a80300                lp      0x15c = test+0x20
8:              acc += ( ((long)(*a++)) * *b++) << 1  ;
 148: 11040402                ld.ab   %r2,[%r1,4]
 14c: 10040403                ld.ab   %r3,[%r0,4]
 150: 2b1c0084                vmpy2h  %r4,%r3,%r2
 154: 24141102                add1    %r2,%r12,%r4
 158: 2214014c                add1    %r12,%r2,%r5
9:
10:     return (short) (acc);
 15c: 7fe0                    j_s.d   [%blink]
 15e: 788e                    sexh_s  %r0,%r12
main:
13: short a[] = {1,2,3,4,5, 6,7,8,9,10};
14: short b[] = {11,12,13,14,15, 16,17,18,19,20};
15:
16: int main(int argc, char *argv[]) {
```

Note: **Assignment:** Remove "<<1" from test function, see what changes in output instructions.

Appendix A.IOTDK Default Core Configurations

ARC_EM5D

This is an ARC EM core with 32 bits of address space, 128 KB of code memory (ICCM) and 256 KB of data memory (DCCM).

```
-arcv2em -core1 -HL -Xcode_density -Xswap -Xnorm -Xmpy16 -Xmpy -Xmpyd
-Xshift_assist -Xbarrel_shifter -Xdsp2 -Xdsp_complex -Xtimer0 -Xtimer1
```

ARC_EM7D

This is an ARC EM core with 32 bits of address space, 256 KB of code memory (ICCM) and 128 KB of data memory (DCCM). Corresponding MetaWare compiler options for this configuration are:

```
-arcv2em -core2 -HL -Xcode_density -Xdiv_rem=radix2 -Xswap
-Xbitscan -Xmpy_option=mpyd -Xshift_assist -Xbarrel_shifter
-Xdsp2 -Xdsp_complex -Xdsp_divsqrt=radix2 -Xdsp_accshift=limited -Xtimer0
-Xtimer1 -Xstack_check -Hccm -Xdmac
```

ARC_EM9D

This is an ARC EM core with 32 bits of address space, 256 KB of code memory (ICCM) and 128 KB of data memory (DCCM). The corresponding MetaWare compiler options for this configuration are:

```
-arcv2em -core2 -Hrgf_banked_regs=32 -HL -Xcode_density
-Xdiv_rem=radix2 -Xswap -Xbitscan -Xmpy_option=mpyd
-Xshift_assist -Xbarrel_shifter -Xdsp2 -Xdsp_complex
-Xdsp_divsqrt=radix2 -Xdsp_itu-Xdsp_accshift=full -Xagu_large
-Xxy -Xbitstream -Xfpus_div -Xfpu_mac -Xfpus_mpy_slow
-Xfpus_div_slow -Xtimer0 -Xtimer1 -Xstack_check -Hccm -Xdmac
```

ARC_EM11D Configuration

This is an ARC EM core with 32 bits of address space, 64 KB of code memory (ICCM) and 64 KB of data memory (DCCM). Corresponding MetaWare compiler options for this configuration are:

```
-arcv2em -core2 -Hrgf_banked_regs=32 -HL -Xcode_density
-Xdiv_rem=radix2 -Xswap -Xbitscan -Xmpy_option=mpyd
-Xshift_assist -Xbarrel_shifter -Xdsp2 -Xdsp_complex -Xdsp_divsqrt=radix2
-Xdsp_itu -Xdsp_accshift=full -Xagu_large -Xxy -Xbitstream -Xfpus_div
-Xfpu_mac -Xfpuda -Xfpus_mpy_slow -Xfpus_div_slow -Xtimer0 -Xtimer1
-Xstack_check -Hccm -Xdmac
```

Programming ARC DSP Using FXAPI

Part 1. Prerequisites

Before starting using ARC DSP the following prerequisites are required:

- Have MetaWare tools for Windows installed

  https://www.synopsys.com/dw/ipdir.php?ds=sw_metaware
- Known how to create, edit, build and debug projects in MetaWare IDE
- Have ARC IOT Design Kit (IOTDK) board and Digilent USB drivers (Digilent Adept 2) installed and tested

  http://store.digilentinc.com/digilent-adept-2-download-only
- IOTDK board is based on DSP-enabled core configurationEM9D

The following needs to be tested before starting this lab:

- Connecting IOTDK board to computer

- Connecting serial console (PuTTY) to IOTDK COM port (For information on how to do initial board setup and configuration please refer to *Getting Started* chapter of *ARC IOT Design Kit User Guide* that came along with IOTDK board).

Part 2. Lab Objectives

Use FXAPI and compare program run speed with and without FXAPI, i.e. DSP extension usage.

Part 3. Lab principle and method

This lab uses complex number multiplication as an example where using just compiler optimization options cannot gain the same effect as calling DSP instructions manually through FXAPI.

In this lab two implementations of complex multiplication are shown with and without FXAPI.

Complex number multiplication

Multiplication of two complex numbers $\mathrm{a}\ (R_a + I_a i)$ and $\mathrm{b}\ (R_b + I_b i)$

Is done using formula:

$ab = (R_a + I_a i)(Rb + Ibi) = (R_a R_b - I_a I_b) + (R_a I_b + R_b I_a)i$

In this lab example multiplication and accumulation of two arrays of complex numbers will be used as a way to compare performance of ARC DSP extensions when used effectively.

The sum of element wise products of two arrays of complex numbers is calculated according to the following formula:

$$result = \sum_{i=0}^{N} a_i + b_i$$

where a and b are arrays of N complex numbers.

Implementation without DSP

In order to calculate element wise products of two arrays of complex numbers a struct cat be defined that stores real and imaginary parts of the complex number, thus the calculation process receives an array of structures and works on it. The code is shown below:

```
typedef struct { short real; short imag; } complex_short;

complex_short short_complex_array_mult (complex_short *a, complex_short *b, int␣
↪size) {
    complex_short result = {0,0};
    int acci=0;
    int accr=0;

    for (int i=0; i < size; i++) {
            accr  += (int) ( a[i].real * b[i].real );
            accr  -= (int) ( a[i].imag * b[i].imag );

            acci  += (int) ( a[i].real * b[i].imag );
            acci  += (int) ( a[i].imag * b[i].real );
    }

    result.real = (short) accr;
```

(continues on next page)

(continued from previous page)

```
    result.imag = (short) acci;

    return result;
}
```

The example keep real and imaginary values in variables of type "short", while multiplication results are kept in "int" integer to avoid truncation. Final result is casted to short to return complex number as a result.

Implementation with FXAPI

FXAPI makes it possible to directly access complex number instructions (like MAC) available in ARC DSP Extensions. This is done through complex number type cq15_t, and various fx_* functions. Here `fx_v2a40_cmac_cq15` FXAPI function is called which performs MAC of two cq15_t complex numbers.

```
cq15_t fx_complex_array_mult(cq15_t *a, cq15_t *b, int size) {
    v2accum40_t acc = { 0, 0 };

    for (int i=0; i < size; i++) {
       acc = fx_v2a40_cmac_cq15(acc, *a++, *b++);
    }

    return fx_cq15_cast_v2a40( acc );
}
```

As with previous implementation `q15_t` is of similar size as `short` type thus multiplication result needs larger storage. Here 40b vector accumulator is used directly to store intermediate results of MAC, and is casted to `cq15_t` on return.

Using IOTDK board for performance comparison

To compare performance of these two functions a simple ESMK application is created that performance complex array multiplication using either of the implementations above. The program initializes two arrays of complex numbers with random values and calls functions above in a loop (1 000 000-10 000 000 times) to make calculation delay measurable in seconds, this is done 8 times, and after each loop a LED on board is turn on. In the result LED strip on board works as a "progress bar" showing the process of looped multiplicaitons.

The main performance check loop is shown below, the outer loop runs 8 times (number of LEDs on LED strip) the inner loop makes "LOOPS/8" calls to complex multiplication function, LOOPS variable is configurable to change the total delay. The example below uses DSP types, and can be changed to use short-based struct type.

```
#include "fxarc.h"
#define LOOPS 10000000
int main(int argc, char *argv[]) {
    unsigned int led_status = 0x40 ;
    DWCREG_PTR pctr =
         (DWCREG_PTR) (DWC_GPIO_0 | PERIPHERAL_BASE);
    DWCREG_PTR uart =
         (DWCREG_PTR) (DWC_UART_CONSOLE | PERIPHERAL_BASE);

    gpio_init(pctr);
    uart_initDevice(uart, UART_CFG_BAUDRATE_115200,
           UART_CFG_DATA_8BITS,
           UART_CFG_1STOP, UART_CFG_PARITY_NONE);

    cq15_t  cq15_a[20] = {{0x2000,10},{0x100,20},{4,30}};
    cq15_t  cq15_b[20] = {{0x2000,11},{0x100,21},{5,31}};
    cq15_t res;
```

(continues on next page)

(continued from previous page)

```
    uart_print(uart, "*** Start ***\n\r");

    led_status = 0x7F;

    for (int i =0; i< 8; i++) {
            gpio_set_leds(pctr, led_status);
            for (int j = 1; j < LOOPS/8; j++ ) {
                    res = fx_complex_array_mult(cq15_a, cq15_b, 2);
            };

            led_status = led_status >> 1;
    }

    gpio_set_leds(pctr, 0x01ff);
    uart_print(uart, "*** End ***\n\r");
    return 0;
}
```

Part 4. Test

To test the example below some modification of the code will be required to make have two loops with and without DSP. First you must re-build libraries for this particular configuration of IOTDK:

```
buildlib my_dsp -tcf=<IOTDK tcf file> -bd . -f
```

IOTDK tcf file can be found in https://github.com/foss-for-synopsys-dwc-arc-processors/embarc_osp/tree/feature/iotdk/board/iotdk/configs/10/tcf/arcem9d.tcf

Both examples are to be compiled with DSP extensions, with the following options set:

```
gmake BOARD=iotdk BD_VER=10 CUR_CORE=arcem9d TOOLCHAIN=mw gui
ADT_COPT="-Hdsplib -Xdsp2 -tcf=./arcem9d.tcf -Xdsp_complex"
ADT_LOPT="-Hdsplib -Xdsp2 -tcf=./arcem9d.tcf -Hlib=./my_dsp"
```

With high optimization level set high function using “short” type is compiled to us DSP MAC operation, enabling significant speedup.

```
  158 complex_short short_complex_array_mult(complex_short *a, complex_short *b, int size) {
short_complex_array_mult  mov_s       %r11,%r0
  163         for (int i=0; i < size; i++) {
.1+0x02     brlt.d      %r3,1,0xe4 = .1+0x3c = cmplx_mul.c!164+0xe
.1+0x06     mov_s       %r8,0
.1+0x08     mov         %lp_count,%r3
.1+0x0c     mov_s       %r9,0
.1+0x0e     lp          0xe0 = .1+0x38 = cmplx_mul.c!164+0xa
.1+0x12     ldh.x.ab    %r6,[%r2,4]
.1+0x16     ldh_s.x     %r12,[%r1,2]
  168               acci  += (int) ( a[i].imag * b[i].real );
.1+0x18     mov         %accl,%r9
.1+0x1c     ldh.x.ab    %r0,[%r1,4]
.1+0x20     mac         0,%r12,%r6
.1+0x24     ldh.x       %r3,[%r2,-2]
  167               acci  += (int) ( a[i].real * b[i].imag );
.1+0x28     mac         %r9,%r3,%r0
  165               accr  -= (int) ( a[i].imag * b[i].imag );
.1+0x2c     mpyw_s      %r12,%r12,%r3
  164               accr  += (int) ( a[i].real * b[i].real );
.1+0x2e     mpyw        %r0,%r6,%r0
.1+0x32     add_s       %r0,%r0,%r8
.1+0x34     sub         %r8,%r0,%r12
.1+0x38     nop_s
.1+0x3a     b_s         0xe6 = .1+0x3e = cmplx_mul.c!174
.1+0x3c     mov_s       %r9,0
  174       return result;
.1+0x3e     sth         %r8,[%r11]
.1+0x42     j_s.d       [%blink]
.1+0x44     sth         %r9,[%r11,2]
```

However, using FXAPI enables compiler to directly use complex MAC instruction “cmachfr”.

```
  186 cq15_t fx_complex_aray_mult(cq15_t *a, cq15_t *b, int size) {
fx_complex_aray_mult  mov_s       %r10,0
  189         for (int i=0; i < size; i++) {
.0+0x02       setacc      0,%r10,0x201    ; 0x201 = uart_initDevice+0x01 = uart.c!32+0x1
.0+0x0a       setacc      0,%r10,0x101    ; 0x101 = main+0x11 = cmplx_mul.c!59+0x11
.0+0x12       brlt.d      %r3,1,0x1be = .0+0x2a = cmplx_mul.c!193
.0+0x16       mov         %lp_count,%r3
.0+0x1a       lp          0x1be = .0+0x2a = cmplx_mul.c!193
  190             acc = fx_v2a40_cmac_cq15(acc, *a++, *b++);
.0+0x1e       ld.ab       %r3,[%r2,4]
.0+0x22       ld.ab       %r12,[%r1,4]
.0+0x26       cmachfr     0,%r12,%r3
  193         return fx_cq15_cast_v2a40( acc );
.0+0x2a       getacc      %r1,0xf00       ; 0xf00 = __EH_FRAME_END+0x70
.0+0x32       j_s.d       [%blink]
.0+0x34       st_s        %r1,[%r0]
```

ARC DSP Using DSP Library

Part 1. Prerequisites

Before starting using ARC DSP the following prerequisites are required:

- Have MetaWare tools for Windows installed

  https://www.synopsys.com/dw/ipdir.php?ds=sw_metaware
- Known how to create, edit, build and debug projects in MetaWare IDE
- Have ARC EM Starter Kit (IOTDK) board and Digilent USB drivers (Digilent Adept 2) installed and tested

  http://store.digilentinc.com/digilent-adept-2-download-only
- IOTDK board configured with DSP-enabled core configuration EM9D

The following needs to be tested before starting this lab:

- Connecting IOTDK board to computer
- Configuring IOTDK board to use ARC core with DSP extensions (these labs use EM5D core configuration)
- Connecting serial console (PuTTY) to IOTDK COM port (For information on how to do initial board setup and configuration please refer to *Getting Started* chapter of *ARC IOT Design Kit User Guide* that came along with IOTDK board).

Part 2. Lab Objectives

Use DSP Library and compare program run speed with and without DSP library, i.e. DSP extension usage.

Part 3. Lab principle and method

This lab uses matrix multiplication as an example where DSP library helps to efficiently use DSP extensions as well as write shorter code.

In this lab two implementations of matrix multiplication are shown: done manually and with the use of DSP library.

Matrix multiplication

Multiplication of two matrices A and B of sizes [M*N] and [N*K] respectively is done using the following formula:

$$c_{ij} = \sum_{k=0}^{N-1} a_{ik}\, b_{kj}$$

Where i= 0. . . (M-1) and j = 0..(K-1) are row and column indexes of output matrix, with size [M*N].

Implementation without DSP

An implementation of matrix multiplication of two matrices containing "short" values is shown below. By convention matrices here are implemented as 1D arrays with row-first indexing, where element a_ik is indexed as a_{ik} . Build with the command:

```
gmake BOARD=iotdk BD_VER=10 CUR_CORE=arcem9d TOOLCHAIN=mw gui
ADT_COPT="-Hdsplib -Xdsp2 -tcf=./arcem9d.tcf -Xdsp_complex"
ADT_LOPT="-Hdsplib -Xdsp2 -tcf=./arcem9d.tcf -Hlib=./my_dsp"
```

```
#include "embARC.h"
#include "embARC_debug.h"
#include <stdlib.h>

#define MATRIX_SIZE 20
#define MAX_NUM 1000
#define LOOPS 100000

/* ****************************************** */

/* Matrix manipulation functions */

/* randomize matrix with values up to 'max_value */
void rand_sq_mat(short x[][MATRIX_SIZE], int SIZE, int max_value) ;

/* multiply two square matrices of same size*/
void mul_sq_mat(short x[][MATRIX_SIZE], short y[][MATRIX_SIZE], short z[][MATRIX_
↪SIZE], int size) ;

/* print square matrix through UART*/
void print_sq_mat(short x[][MATRIX_SIZE], int SIZE);

/* ****************************************** */

int main(int argc, char *argv[]) {

    short a[MATRIX_SIZE][MATRIX_SIZE];
    short b[MATRIX_SIZE][MATRIX_SIZE];
    short c[MATRIX_SIZE][MATRIX_SIZE];
    int n =MATRIX_SIZE;

    rand_sq_mat(a,n, MAX_NUM);
    rand_sq_mat(b,n, MAX_NUM);

    print_sq_mat(a,n);
    print_sq_mat(b,n);

    unsigned int led_status = 0x40 ;
    led_status = 0x7F;

    EMBARC_PRINTF("*** Start ***\n\r");

    for (int i =0; i< 8; i++) {
            for (int j = 1; j < LOOPS/8; j++ ) {
                    mul_sq_mat(a,b,c,n);
            };
            led_write(led_status, BOARD_LED_MASK);
            led_status = led_status >> 1;
    }
```

(continues on next page)

(continued from previous page)

```
    print_sq_mat(c,n);

    EMBARC_PRINTF("*** Exit ***\n\r");

    return 0;
}



void rand_sq_mat(short x[][MATRIX_SIZE], int SIZE, int max_value) {
    for (int i=0;i<SIZE;i++) {
            for(int j=0;j<SIZE;j++) {
                    x[i][j] = 1 + (rand() % max_value); //plus 1 to avoid zeros
            }
    }
}

void mul_sq_mat(short x[][MATRIX_SIZE],short y[][MATRIX_SIZE], short z[][MATRIX_
↪SIZE], int size) {
    for (int i=0; i<size; i++) {
            for(int j=0;j<size;j++) {
                    z[i][j]=0;
                    for(int k=0;k<size;k++) {
                            z[i][j] += x[i][k]*y[k][j];
                    }
            }
    }
}

void print_sq_mat(short x[MATRIX_SIZE][MATRIX_SIZE], int SIZE){

    EMBARC_PRINTF("------\n\r");

    for(int j = 0; j < SIZE; j++ ){
        for(int i = 0; i < SIZE; i ++){
            EMBARC_PRINTF("%d\t", x[j][i]);
        }
        EMBARC_PRINTF("\n\r" );
    }

    EMBARC_PRINTF("------\n\r");
}
```

Implementation with DSPLIB

DSP library contains matrix multiplication function so doing matrix multiplication using DSP library requires just initialization of matrix arrays (1D) and call to `dsp_mat_mult_q15`. The overall code is just 4 lines, as highlighted in the following code. Note that dsplib.h must be included, and matrix a, b and c must be declared as global variable. As the numbers are in q15 type, it is better to make elements of a and b between 32767 (~0.99) and 16384 (0.5), or 32768(-1) and 49152 (-0.5) so that the result in c is not round to zero. Note as IOTDK is configured to have small AGU, the DSP library routine is not significantly faster.

```
#include "embARC.h"
#include "embARC_debug.h"
#include <stdlib.h>
#include "dsplib.h"

#define MATRIX_SIZE 20
```

(continues on next page)

(continued from previous page)

```
#define MAX_NUM 1000
#define LOOPS 100000

/* ****************************************** */

/* Matrix manipulation functions */

/* randomize matrix with values up to 'max_value */
//void rand_sq_mat(short x[][MATRIX_SIZE], int SIZE, int max_value) ;

/* multiply two square matrices of same size*/
void mul_sq_mat(short x[][MATRIX_SIZE], short y[][MATRIX_SIZE], short z[][MATRIX_
↪SIZE], int size) ;

/* print square matrix through UART*/
void print_sq_mat(short* x, int SIZE);

/* ****************************************** */
    __xy q15_t a[MATRIX_SIZE*MATRIX_SIZE];
    __xy q15_t b[MATRIX_SIZE*MATRIX_SIZE];
    __xy q15_t c[MATRIX_SIZE*MATRIX_SIZE];

int main(int argc, char *argv[]) {

    int n =MATRIX_SIZE;
matrix_q15_t matA, matB, matC;

    //rand_sq_mat(a,n, MAX_NUM);
    //rand_sq_mat(b,n, MAX_NUM);
    for (int i =0; i< MATRIX_SIZE*MATRIX_SIZE; i++) { a[i]=16384; }
    for (int i =0; i< MATRIX_SIZE*MATRIX_SIZE; i++) { b[i]=16383; }


    print_sq_mat(a,n);
    print_sq_mat(b,n);

dsp_mat_init_q15(&matA,  MATRIX_SIZE, MATRIX_SIZE, a);
dsp_mat_init_q15(&matB,  MATRIX_SIZE, MATRIX_SIZE, b);
dsp_mat_init_q15(&matC,  MATRIX_SIZE, MATRIX_SIZE, c);
dsp_status status;

    unsigned int led_status = 0x40 ;
    led_status = 0x7F;

    EMBARC_PRINTF("*** Start ***\n\r");

    for (int i =0; i< 8; i++) {
            for (int j = 1; j < LOOPS/8; j++ ) {
                    status = dsp_mat_mult_q15(&matA, &matB, &matC);
            };
            led_write(led_status, BOARD_LED_MASK);
            led_status = led_status >> 1;
    }

    if ( status == DSP_ERR_OK ) EMBARC_PRINTF("done\n");
    else EMBARC_PRINTF("something wrong");
    print_sq_mat(c,n);

    EMBARC_PRINTF("*** Exit ***\n\r");

    return 0;
```

(continues on next page)

(continued from previous page)

```
}



//void rand_sq_mat(short x[][MATRIX_SIZE], int SIZE, int max_value) {
//  for (int i=0;i<SIZE;i++) {
//          for(int j=0;j<SIZE;j++) {
//                  x[i][j] = 1 + (rand() % max_value); //plus 1 to avoid zeros
//          }
//  }
//}
//
//void mul_sq_mat(short x[][MATRIX_SIZE],short y[][MATRIX_SIZE], short z[][MATRIX_
↪SIZE], int size) {
//  for (int i=0; i<size; i++) {
//          for(int j=0;j<size;j++) {
//                  z[i][j]=0;
//                  for(int k=0;k<size;k++) {
//                          z[i][j] += x[i][k]*y[k][j];
//                  }
//          }
//  }
//}

void print_sq_mat(short* x, int SIZE){

    EMBARC_PRINTF("------\n\r");

    for(int j = 0; j < SIZE; j++ ){
        for(int i = 0; i < SIZE; i ++){
            EMBARC_PRINTF("%d\t", x[i+j*SIZE]);
        }
        EMBARC_PRINTF("\n\r" );
    }

    EMBARC_PRINTF("------\n\r");
}
```

Note: **Assignment 1:** Use example in previous lab to create an IOTDK board application that uses LED strip as progress bar for large number of matrix multiplications with and without DSP library. Adjust number of loops made to achieve measurable delay.

Part 4. Test

To test the example below an example program needs to be created that has two loops of matrix multiplications with and without DSP library.

Both examples are to be compiled with DSP extensions, with the following options set:

```
-O2 -arcv2em -core1 -Xlib -Xtimer0 -Xtimer1 -Xdsp1 -Hdsplib
```

Note: **Assignment 2:** Run the example and compare computational delay with and without DSPLIB

Note: Note that DSPLIB is statically linked with the project when -Hdsplib is set, and as the DSPLIB itself is pre-compiled with high level of optimization, changing optimization option for example program won't affect DSPLIB

performance. On the other hand even with highest optimization level a function utilizing simple instructions on "short" type (even converted to MACs if possible) will be less efficient that direct use of DSPLIB.

A WiFi temperature monitor

ESPB266 WIFI module

Purpose

- Learn how to build a wireless sensor terminal based on the embARC_osp package
- Further familiarity with the use of the FreeRTOS operating system
- Learn about the use of the esp8266 module and AT command

Equipment

The following hardware and software tools are required:

- PC host
- ARC GNU toolchain/MetaWare Development Toolkit
- ARC board (EM Starter kit/IoT Development Kit)
- embARC OSP package
- `embarc_osp/arc_labs/labs/lab10_esp8266_wifi`

Content

This experiment is based on the embARC_osp package extension to support the popular esp8266 WIFI module, use the AT command to set the esp8266 to the server mode, and then use the PC or mobile phone under the same LAN to access its IP address, based on the TCP protocol to achieve static page display of the client.

Through this experiment, we will have a preliminary understanding of the use of the esp8266 WIFI module and the AT command.

Principles

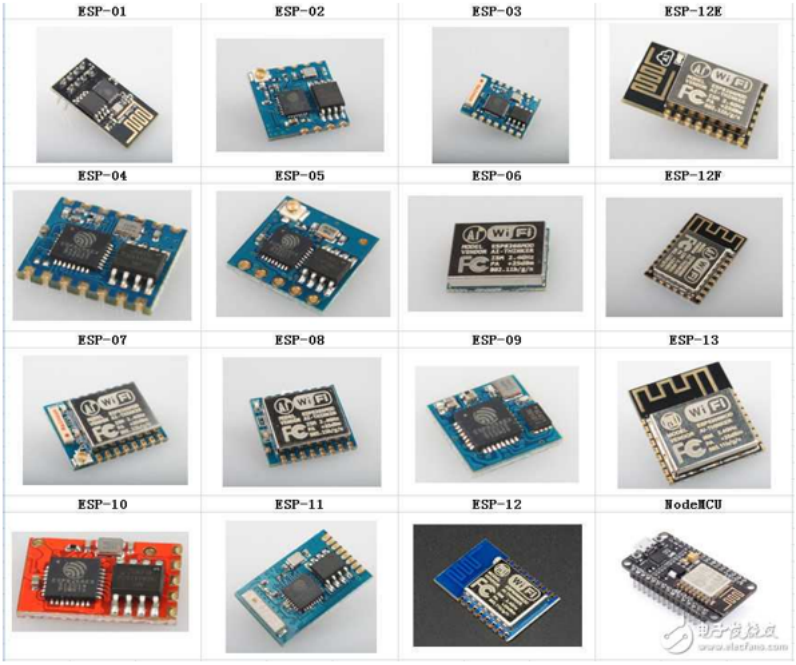
Esp8266

The ESP8266 is an ultra-low-power Wi-Fi chip with industry-leading package size and ultra-low power technology designed for mobile devices and IoT applications, connecting users' physical devices to Wi-Fi wireless On the network, Internet or LAN communication is implemented to implement networking functions.

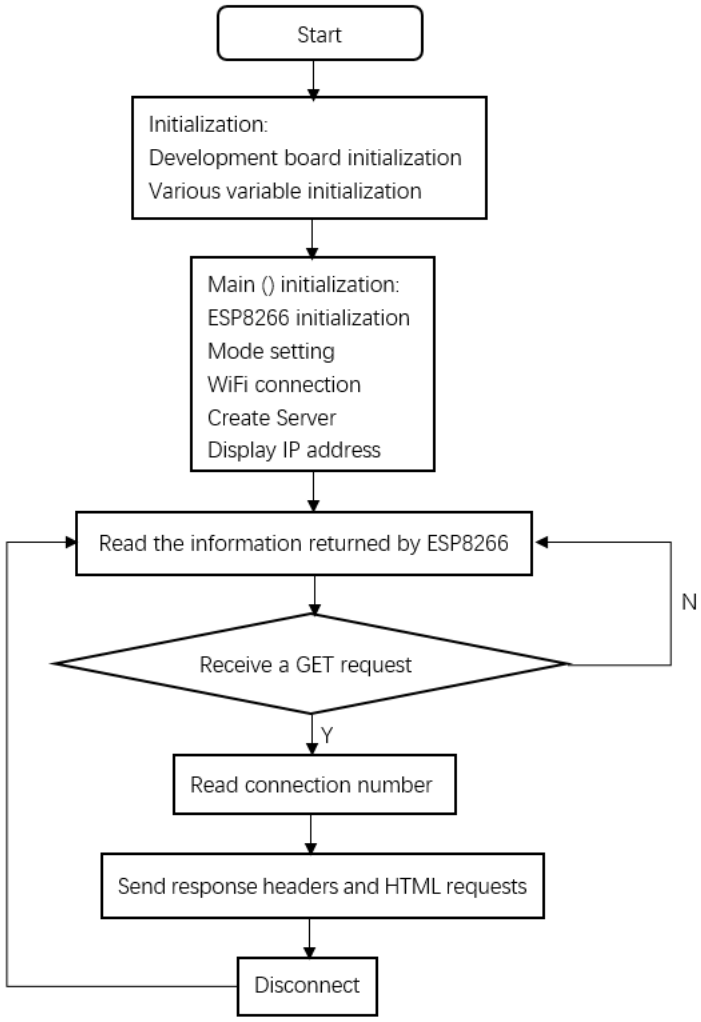
The ESP8266 is available in a variety of encapsulation. The antenna supports the onboard PCB antenna, IPEX interface and stamp hole interface.

ESP8266 can be widely used in smart grid, intelligent transportation, smart furniture, handheld devices, industrial control and other fields.

Ai-Thinker company has developed several Wi-Fi modules with ESP8266 as its core(The Wi-Fi modules are shown below).



Program structure (as shown bellow)



Code (as shown bellow)

```
#include "embARC.h"
#include "embARC_debug.h"

#include "board.h"
//#include "dev_uart.h"
#include "esp8266.h"

#include <stdio.h>
#include <string.h>

#define WIFI_SSID    "\"liu\""
#define WIFI_PWD     "\"12345678\""

static char http_get[] = "GET /";
static char http_IDP[] = "+IPD,";
static char http_html_header[] = "HTTP/1.x 200 OK\r\nContent-type: text/
↪html\r\n\r\n";
static char http_html_body_1[] = "<html><head><title>ESP8266_AT_HttpServer</title>
↪</head><body><h1>Welcome to this Website</h1>";
static char http_html_body_2[] = "<p>This Website is used to test the AT command␣
↪about HttpServer of ESP8266.</p></body></html>";

int main(void)
{
        char *conn_buf;
        char scan_result[1024];

        //ESP8266 Init
        EMBARC_PRINTF("============================ Init␣
↪============================\n");

        ESP8266_DEFINE(esp8266);
        esp8266_init(esp8266, UART_BAUDRATE_115200);
        at_test(esp8266->p_at);
        board_delay_ms(100, 1);

        //Set Mode
        EMBARC_PRINTF("============================ Set Mode␣
↪============================\n");

        esp8266_wifi_mode_get(esp8266, false);
        board_delay_ms(100, 1);
        esp8266_wifi_mode_set(esp8266, 3, false);
        board_delay_ms(100, 1);

        //Connect WiFi
        EMBARC_PRINTF("============================ Connect WiFi␣
↪============================\n");

        do
        {
                esp8266_wifi_scan(esp8266, scan_result);
                EMBARC_PRINTF("Searching for WIFI %s ......\n", WIFI_SSID);
                board_delay_ms(100, 1);
        }
        while (strstr(scan_result, WIFI_SSID) == NULL);

        EMBARC_PRINTF("WIFI %s found! Try to connect\n", WIFI_SSID);

        while(esp8266_wifi_connect(esp8266, WIFI_SSID, WIFI_PWD, false)!=AT_OK)
        {
                EMBARC_PRINTF("WIFI %s connect failed\n", WIFI_SSID);
```

(continues on next page)

(continued from previous page)

```
                board_delay_ms(100, 1);
        }

        EMBARC_PRINTF("WIFI %s connect succeed\n", WIFI_SSID);

        //Creat Server
        EMBARC_PRINTF("============================ Connect Server␣
↪============================\n");

        esp8266_tcp_server_open(esp8266, 80);

        //Show IP
        EMBARC_PRINTF("============================ Show IP␣
↪============================\n");

        esp8266_address_get(esp8266);
        board_delay_ms(1000, 1);

        EMBARC_PRINTF("============================ while␣
↪============================\n");

        while (1)
        {
                memset(scan_result, 0, sizeof(scan_result));
                at_read(esp8266->p_at ,scan_result ,1000);
                board_delay_ms(200, 1);
                //EMBARC_PRINTF("Alive\n");

                if(strstr(scan_result, http_get) != NULL)
                {

                        EMBARC_PRINTF("============================ send␣
↪============================\n");

                        EMBARC_PRINTF("\nThe message is:\n%s\n", scan_result);

                        conn_buf = strstr(scan_result, http_IDP) + 5;
                        *(conn_buf+1) = 0;

                        EMBARC_PRINTF("Send Start\n");
                        board_delay_ms(10, 1);

                        esp8266_connect_write(esp8266, http_html_header, conn_buf,␣
↪(sizeof(http_html_header)-1));
                        board_delay_ms(100, 1);

                        esp8266_connect_write(esp8266, http_html_body_1, conn_buf,␣
↪(sizeof(http_html_body_1)-1));
                        board_delay_ms(300, 1);

                        esp8266_connect_write(esp8266, http_html_body_2, conn_buf,␣
↪(sizeof(http_html_body_2)-1));
                        board_delay_ms(300, 1);

                        esp8266_CIPCLOSE(esp8266, conn_buf);

                        EMBARC_PRINTF("Send Finish\n");
                }
        }

        return E_OK;
```

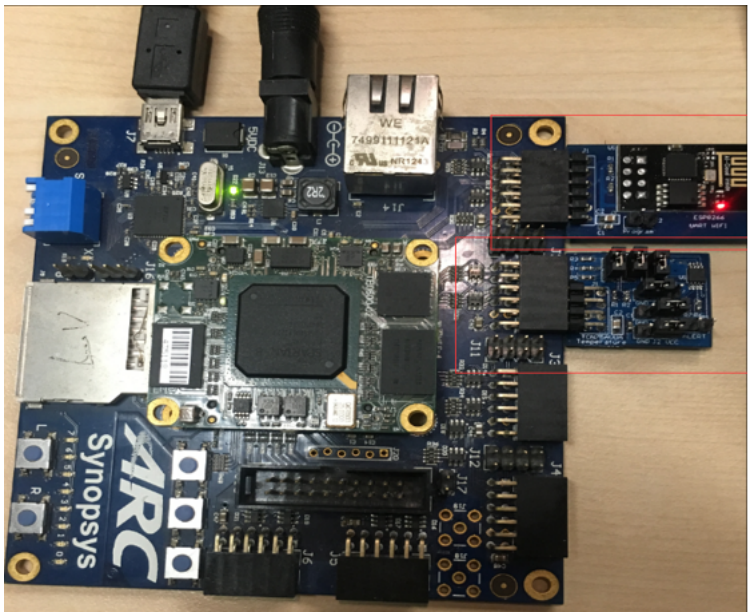
(continues on next page)

(continued from previous page)

```
}
```

Steps

Hardware connection (as shown bellow)



Compile and download

Compile and download the program, after downloading successfully, you will see the relevant download information in the cmd window(as shown bellow).

```
0x00000004 in ?? ()
Loading section .init, size 0x1b0 lma 0x10000000
Loading section .vector, size 0x400 lma 0x10000400
Loading section .text, size 0x1446c lma 0x10000800
Loading section .rodata, size 0x1cb4 lma 0x10014c6c
Loading section .data, size 0xc2c lma 0x10016920
Start address 0x10000004, load size 94972
Transfer rate: 602 KB/sec, 9497 bytes/write.
Continuing.
```

At this point, the serial port debugging tool will see the serial port feedback information, reflecting the process of the EMSK development board using the AT command to establish the http server(as shown bellow).

```
embARC Build Time: Mar 21 2018, 17:53:27
Compiler Version: ARC GNU, 7.1.1 20170710
============================ Init ============================
[at_parser_init]56: obj->psio 0x1006ba30 -> 0x10057948

.............

OK" (9)
============================ Set Mode ============================
[at_send_cmd]117: at_out: "AT+CWMODE_CUR?
" (16)

.................
```

(continues on next page)

(continued from previous page)

```
OK" (24)
============================ Connect WiFi ============================
[at_send_cmd]117: at_out: "AT+CWLAP
" (10)

...................

OK" (24)
[at_send_cmd]117: at_out: "AT+CWJAP_CUR="liusongwei","632139751"

..........

WIFI "liusongwei" connect succeed
============================ Connect Server ============================
[at_send_cmd]117: at_out: "AT+CIPMUX=1
" (13)

........

OK" (26)
============================ Show IP ============================
[at_send_cmd]117: at_out: "AT+CIFSR
" (10)
[at_get_reply]137: "
AT+CIFSR
+CIFSR:STAIP,"192.168.137.81"
+CIFSR:STAMAC,"5c:cf:7f:0b:5c:d1"

OK" (83)
============================ while ============================
.............
============================ send  ============================
..............

Send Start
Send Finish
```

Access server

It can be seen from the serial port feedback information in above *Compile and download* that the EMSK development board has successfully connected to the target WIFI through esp8266, and is set to the server mode by using the AT command, and the IP address of the server is given.

At this point, use a PC or mobile phone to connect to the same WIFI, open a browser, and enter the IP address 192.168.137.81 to see the static HTTP page.

Exercises

Referring to the experiment of MRF24G WIFI module, using esp8266 and TCN75 temperature sensor to build http server to make the page display the sensor temperature in real time.

BLE Communication

Purpose

- Familiar with the wireless communication in IoT
- Familiar with the usage of RN4020 BLE module on IoT Development Kit
- Learn to use the API of RN4020 driver in embARC OSP

Equipment

The following hardware and software tools are required:

- PC host
- ARC GNU toolchain/MetaWare Development Toolkit
- ARC board (IoT Development Kit)
- embARC OSP package
- `embarc_osp/arc_labs/labs/lab6_ble_rn4020`

Content

Complete the communication between smartphone and IoTDK board through RN4020 BLE module. At first, setup RN4020 BLE module by using API of RN4020 driver. Then connect mobile phone and RN4020 by BLE, and check the data send by IoTDK in smartphone. Finally, send data from smartphone to IoTDK board, and print this data value in terminal.

Principles

RN4020 BLE module is controlled by the user through input/output lines (i.e., physical device pins) and an UART interface. The UART Interface supports ASCII commands to control/configure the RN4020 modules for any specific requirement based on the application.

Setup

Before connecting an RN4020 module to a smartphone device, user may need to set up the RN4020 module as follows.

1. Configure UART which connected to RN4020 with these parameters: **Baud rate - 115200, Data bits - 8, Parity - None, Stop bits - 1**
2. Set the **WAKE_SW** pin high to enter Command mode
3. Issue the command **SF, 1** to reset to the factory default configuration
4. Issue the command **SN, IoT DK** to set the device name to be "IoT DK"
5. Issue the command **SS, C0000001** to enable support of the Device Information, Battery Service and User Defined Private Service
6. Issue the command **SR, 00002000** to set the RN4020 module as a server
7. Issue the command **PZ** to clear all settings of the private service and the private characteristics
8. Issue the command **PS, 11223344556677889900AABBCCDDEEFF** to set the UUID of user defined private service to be 0x11223344556677889900AABBCCDDEEFF
9. Issue the command **PC, 0102030405060708090 00A0B0C0D0E0F, 18, 06** to add private characteristic 0x0102030405060708090 00A0B0C0D0E0F to current private service. The property of this characteristic is 0x18 (writable and could notify) and has a maximum data size of 6bytes.
10. Issue the command **R, 1** to reboot the RN4020 module and to make the new settings effective
11. Issue the command **LS** to display the services

The source code using the API of RN4020 driver in embARC OSP as follows.

```
rn4020_setup(rn4020_ble);
rn4020_reset_to_factory(rn4020_ble);

/* Set device Name */
rn4020_set_dev_name(rn4020_ble, "IoT DK");

/* Set device services */
rn4020_set_services(rn4020_ble, RN4020_SERVICE_DEVICE_INFORMATION |
                                RN4020_SERVICE_BATTERY |
                                RN4020_SERVICE_USER_DEFINED);

rn4020_set_features(rn4020_ble, RN4020_FEATURE_SERVER_ONLY);
rn4020_clear_private(rn4020_ble);

/* Set private service UUID and private characteristic */
rn4020_set_prv_uuid(rn4020_ble, RN4020_PRV_SERV_HIGH_UUID, RN4020_PRV_SERV_LOW_
↪UUID);
rn4020_set_prv_char(rn4020_ble, RN4020_PRV_CHAR_HIGH_UUID, RN4020_PRV_CHAR_LOW_
↪UUID, 0x18, 0x06, RN4020_PRIVATE_CHAR_SEC_NONE);

/* Reboot RN4020 to make changes effective */
rn4020_reset(rn4020_ble);

rn4020_refresh_handle_uuid_table(rn4020_ble);
```

Advertise

Issue the command **A** to start advertisement. The source code using the API of RN4020 driver in embARC OSP as follows.

```
rn4020_advertise(rn4020_ble);
```

Send data

Issue the command **SUW, 2A19, value** to set the level of Battery. The source code using the API of RN4020 driver in embARC OSP as follows.

```
while (1) {

        rn4020_battery_set_level(rn4020_ble, battery--);

        board_delay_ms(1000, 0);
        if (battery < 30) {
                battery = 100;
        }
}
```

Note: About detailed usage of RN4020 BLE module, please refer to the document "RN4020 Bluetooth Low Energy Module User's Guide"

Steps

Run project

Open the serial terminal emulator in PC (e.g. Tera Term), set as **115200 baud, 8 bits data, 1 stop bit and no parity**, connect to the IoTDK board.

Open `cmd` under the folder *embarc_osp/arc_labs/labs/lab6_ble_rn4020*, input the command as follow

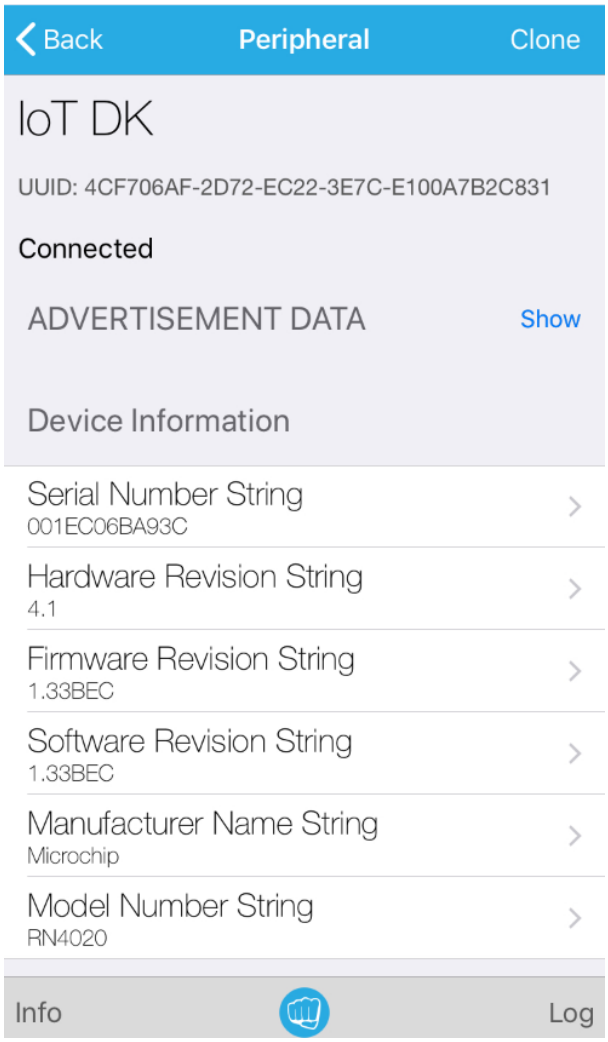
```
make run
```

Then you will see the output in the serial terminal

Connection

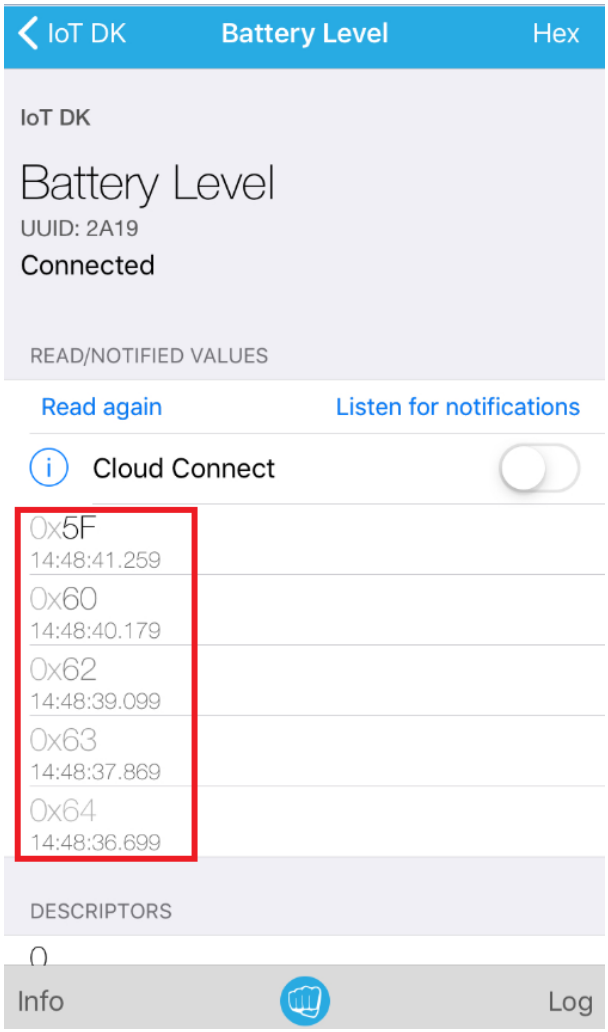
Open the BLE browser APP in smartphone (e.g. LightBlue in IOS), and scan for BLE peripherals, connect the "IoT DK" device. Then you will see the output in the serial terminal.

And the device information in BLE browser APP.

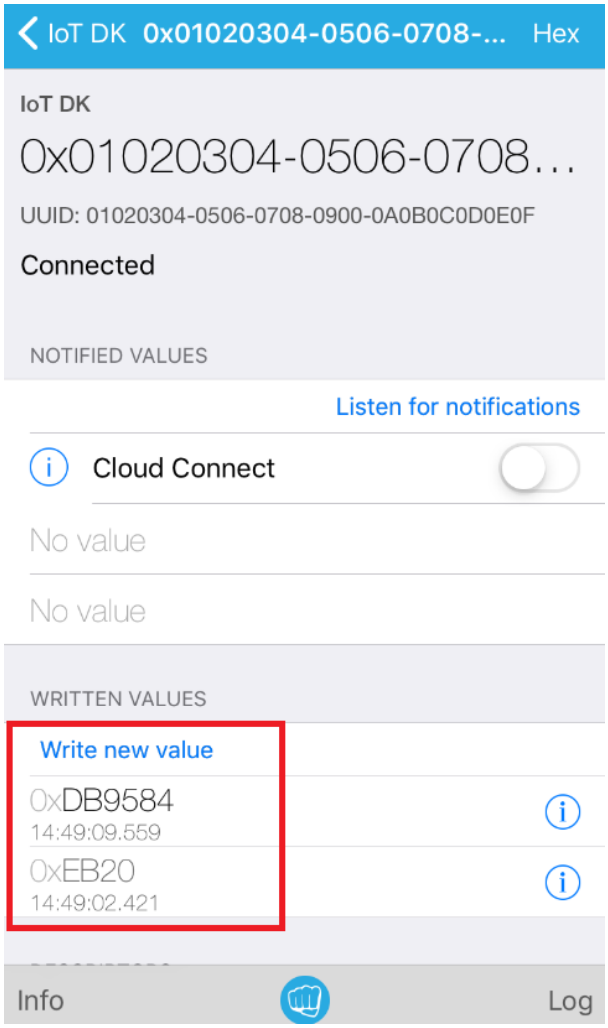


Communication

Read the data of Battery services in BLE browser APP. Check whether the data decrease gradually.



Write data to in BLE browser APP. Check the received data in BLE browser APP.



Exercises

Try to use the received data in IoTDK board, and do some control by using GPIO. (e.g. LED on/off)

Memory map and linker

Purpose

- To get familiar with the memory mapping in the compile process
- To learn how to use the linker

Equipment

The following hardware and software tools are required:

- PC host
- ARC GNU toolchain/MetaWare Development Toolkit
- nSIM simulator
- `embarc_osp/arc_labs/labs/lab8_linker`

Content

Customizing your program with **compiler pragmas**. At first, using "pragma code" to specify a new name of section in which the code of function reside. Then mapping this code section into specified memory location with linker. Finally, checking the location of this code section after build process.

Principles

By default, compiler-generated code is placed in the *.text* section. The default code section name can be overridden by using the *code pragma*. After compile process, the linker will automatically map all input sections from object files to output sections in executable files. If you want to customize the mapping, you can change the default linker mapping by invoking a linker command file.

Steps

Create a project and overriding code section name

Open the MetaWare IDE, create an empty C project called lab_linker and select ARC EM series processor. Then import the main.c and link.cmd files under *embarc_osp/arc_labs/labs/lab8_linker* directory into the project.

Open main.c file in MetaWare IDE, using "pragma code" to change the section in which function `modify` reside from *.text* to a new name "*modify_seg*".

```
#pragma Code ("modify_seg")
void modify(int list[], int size) {
    int out, in, temp;

    for(out=0; out<size; out++)
        for(in=out+1; in<size; in++)
            if(list[out] > list[in]) {
                temp = list[in];
                list[in] = list[out];
                list[out] = temp;
            }
}
#pragma Code ()
```

Pragma code has two forms that must be used in pairs to bracket the affected function definitions:

```
#pragma code(Section_name)
/* ----- Affected function definitions go here ---- */
#pragma code() /* No parameters here */
```

Section_name is a constant string expression that denotes the name of the section

Note: About detailed usage of the compiler pragmas, please refer to the document "MetaWare C/C++ Programmer's Guide for the ccac Compiler"

Edit the linker command file

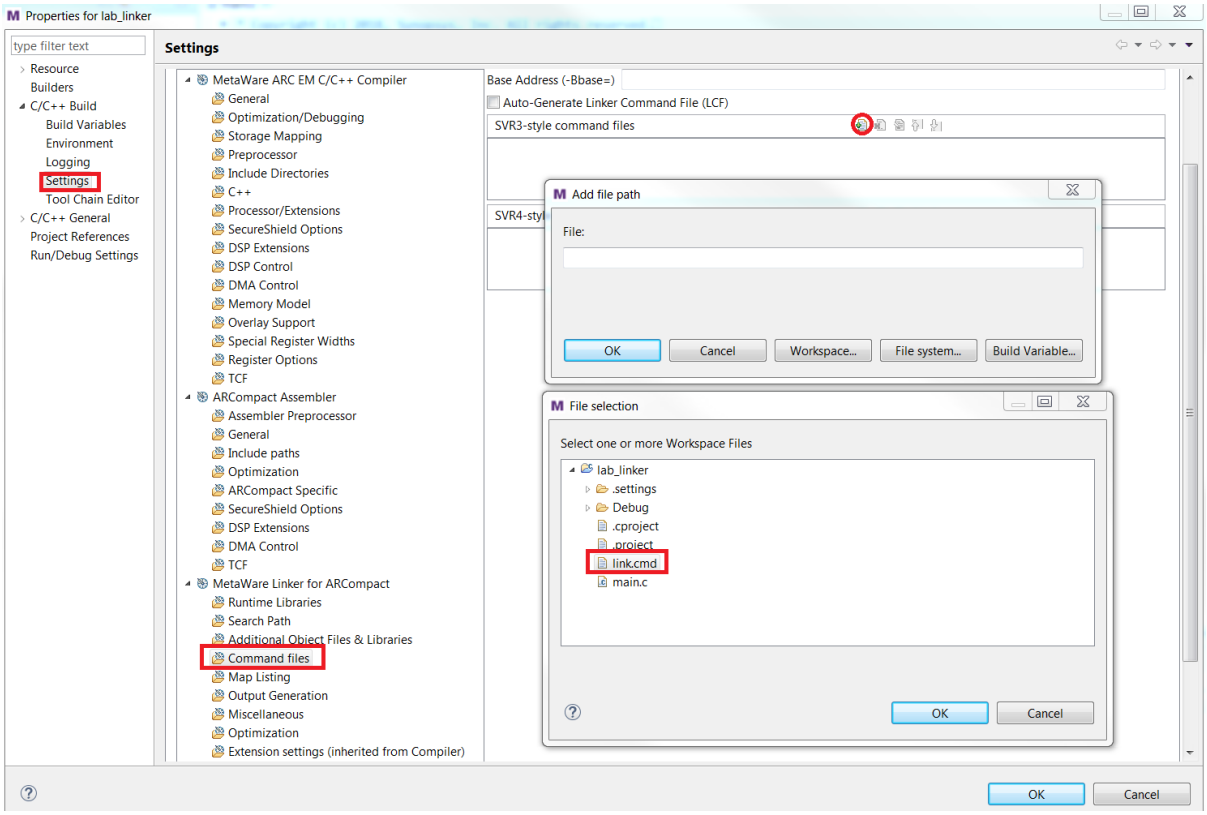
Open the link.cmd file, there are two parts, one is for memory blocks location, the other is for sections mapping. Add one new block named "*MyBlock*" in MEMORY, the start address is 0x00002000, and the size is 32KB. Then add one new GROUP in SECTIONS, and mapping section "*modify_seg*" into *MyBlock*.

```
MEMORY {
    // Note: overlap of code and data spaces is not recommended since it makes
    //       Address validity checking impossible with the debugger and simulator
    MyBlock: ORIGIN = 0x00002000, LENGTH = 32K
    MEMORY_BLOCK1:  ORIGIN = 0x0010000, LENGTH = 64K
    MEMORY_BLOCK2:  ORIGIN = 0x0020000, LENGTH = 128K
}

SECTIONS {
    GROUP: {
                modify_seg: {}
            }>MyBlock
......
```

Note: About format and syntax of the linker command file, please refer to the document "MetaWare ELF Linker and Utilities User's Guide"

Add the linker command file into the project

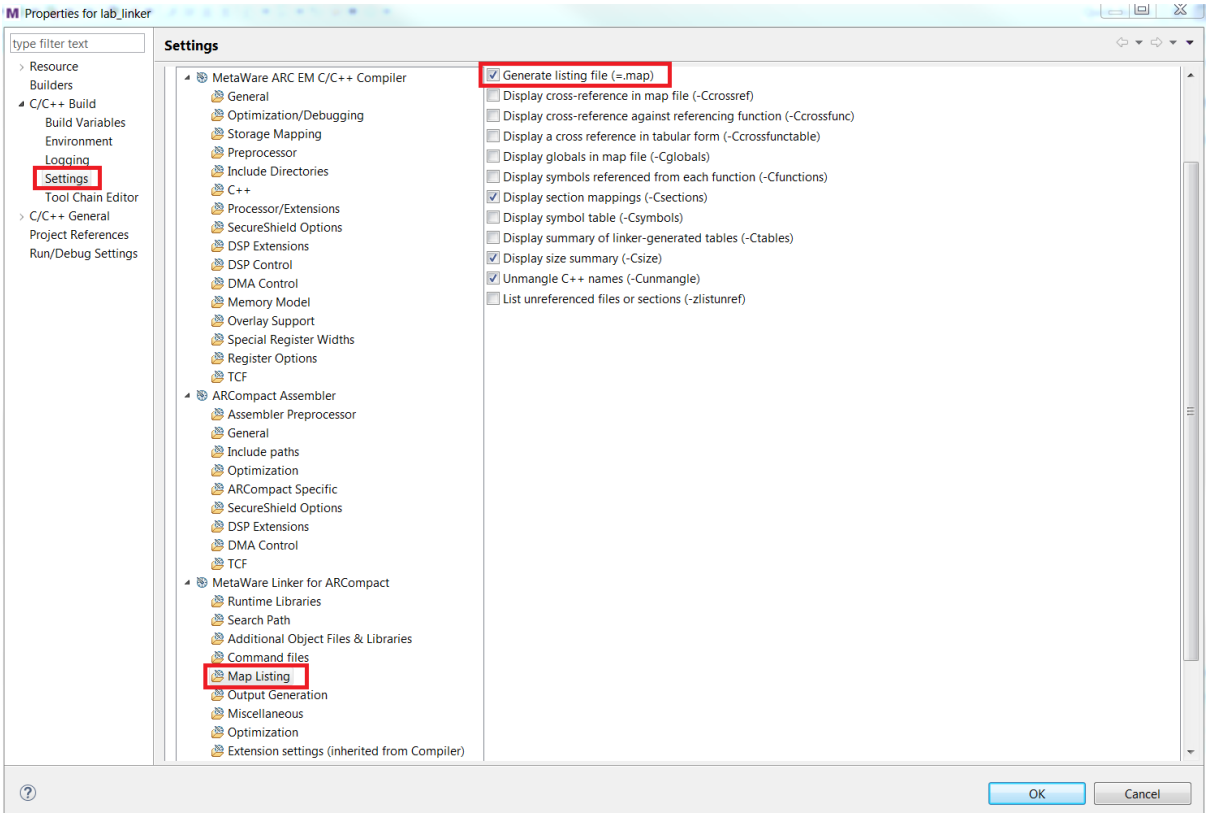
Right click the current project lab_linker and select Properties in the popup tab. Click C/C++ build **>>** settings **>>** Tool Settings to open the linker option settings page.



In the current page, select Command files to add linker.cmd file into this project.

Check the result

In linker option settings page, select Map listing to check Generate listing file(=.map)



Build the lab_linker project, then open the lab_linker.map file under Debug directory.

```
SECTION SUMMARY
_______________

OUTPUT/  TYPE   START     END
 INPUT SECTION  ADDRESS   ADDRESS   LENGTH

modify_seg
         text   00002000  00002039  0000003a
.fini    text   00010000  00010005  00000006
.init    text   00010008  0001000d  00000006
.text    text   00010010  0001013d  0000012e
.vectors text   00010140  0001017f  00000040
.sdata   bss    00020000  0001ffff  00000000
.data    data   00020000  0002001f  00000020
.stack   bss    00020020  0003001f  00010000
```

In this file, search SECTIONS SUMMARY, then you can check the size and location of *modify_seg* section, it resides in *MyBlock*, same as you set in the linker cmd file.

Exercises

Try to check the memory mapping info of *modify_seg* section by using elfdump tool.

How to use FreeRTOS

Purpose

- To learn how to implement tasks in FreeRTOS operating system
- To learn how to register tasks in FreeRTOS
- To get familiar with inter-task communication of FreeRTOS

Equipment

The following hardware and software tools are required:

- PC host
- ARC GNU toolchain/MetaWare Development Toolkit
- ARC board (EM Starter kit/IoT Development Kit)
- embARC OSP package
- `embarc_osp/arc_labs/labs/lab9_freertos`

Content

This lab utilizes FreeRTOS v9.0.0, and will create 3 tasks based on embARC_osp. You should apply inter-task communicating methods such as semaphore and message queue in order to get running LEDs result. We should go though basic functions of FreeRTOS first.

Principles

Background

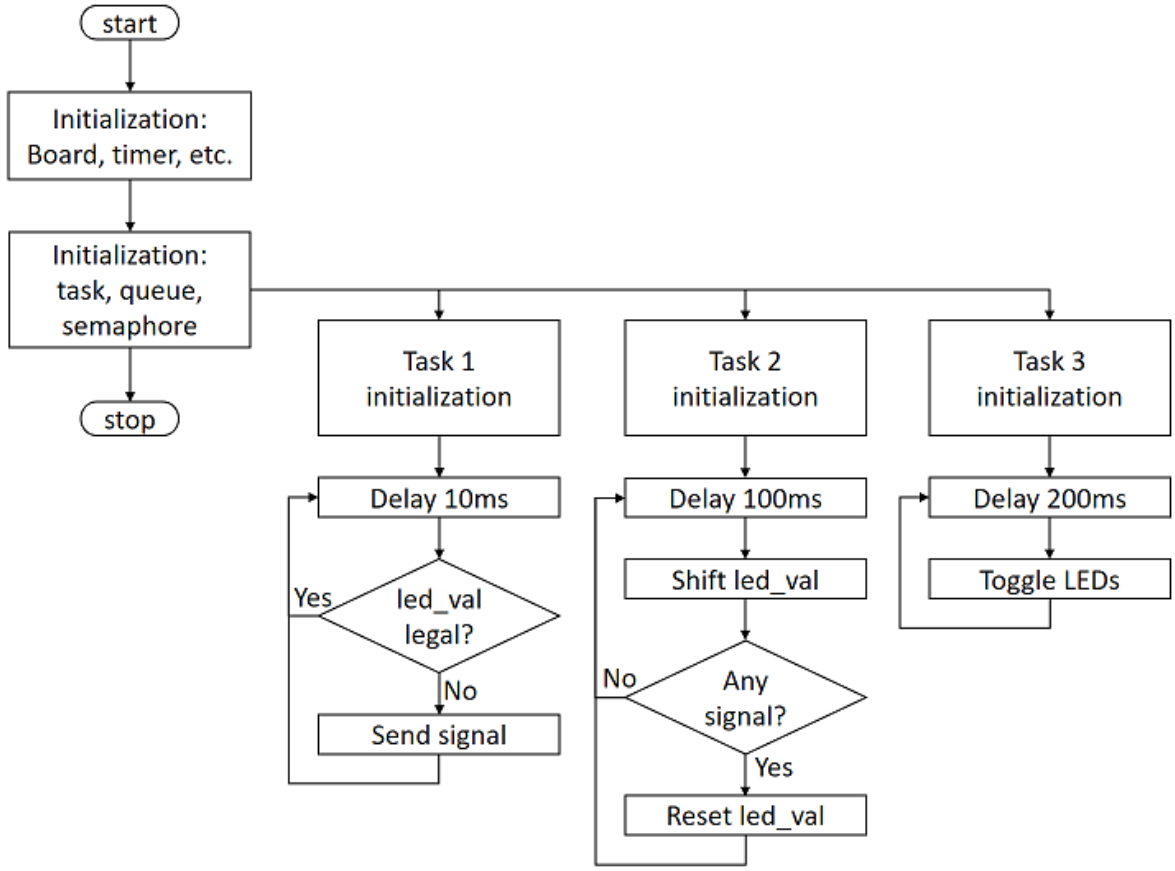
A Real Time Operating System (RTOS) is an operating system intended to serve real-time applications that process data in limited time as it comes in. Being within time bound and highly reliable are two important characters of RTOS.

As resources becoming abundant for modern micro processors, the cost to run RTOS is become increasingly neglectable. RTOS also provides event-driven mode for better utilization of CPU with efficiency. Among RTOSs for micro processors, FreeRTOS stands out as a free for use, opensourced RTOS with complete documents. These are the reasons of why we choose to learn FreeRTOS in this lab.

Design

This lab implements a running LED light with 3 tasks on FreeRTOS. Despite using 3 tasks is an overkill for a running LED, but it's beneficial for the understanding of FreeRTOS itself and inter-task communication as well.

The flow chat of the program is shown below:



Realization

The code of system is shown below, including various Initialization and task time delay.

```
#include "embARC.h"
#include "embARC_debug.h"
#include <stdlib.h>
```

(continues on next page)

(continued from previous page)

```
static void task1(void *par);
static void task2(void *par);
static void task3(void *par);

#define TSK_PRIOR_1             (configMAX_PRIORITIES-1)
#define TSK_PRIOR_2             (configMAX_PRIORITIES-2)
#define TSK_PRIOR_3             (configMAX_PRIORITIES-3)

// Semaphores
static SemaphoreHandle_t sem1_id;

// Queues
static QueueHandle_t dtq1_id;

// Task IDs
static TaskHandle_t task1_handle = NULL;
static TaskHandle_t task2_handle = NULL;
static TaskHandle_t task3_handle = NULL;

int main(void)
{
        vTaskSuspendAll();

        // Create Tasks
        if (xTaskCreate(task1, "task1", 128, (void *)1, TSK_PRIOR_1, &task1_
↪handle)      != pdPASS){
                /*!< FreeRTOS xTaskCreate() API function */
                EMBARC_PRINTF("Create task1 Failed\r\n");
                return -1;
        } else {
                EMBARC_PRINTF("Create task1 Successfully\r\n");
        }

        if (xTaskCreate(task2, "task2", 128, (void *)2, TSK_PRIOR_2, &task2_
↪handle)      != pdPASS){
                /*!< FreeRTOS xTaskCreate() API function */
                EMBARC_PRINTF("Create task2 Failed\r\n");
                return -1;
        } else {
                EMBARC_PRINTF("Create task2 Successfully\r\n");
        }

        if (xTaskCreate(task3, "task3", 128, (void *)3, TSK_PRIOR_3, &task3_
↪handle)      != pdPASS){
                /*!< FreeRTOS xTaskCreate() API function */
                EMBARC_PRINTF("Create task3 Failed\r\n");
                return -1;
        } else {
                EMBARC_PRINTF("Create task3 Successfully\r\n");
        }

        // Create Semaphores
        sem1_id = xSemaphoreCreateBinary();
        xSemaphoreGive(sem1_id);

        // Create Queues
        dtq1_id = xQueueCreate(8, sizeof(uint32_t));

        xTaskResumeAll();
        vTaskSuspend(NULL);
```

(continues on next page)

(continued from previous page)

```
        return 0;
}

static void task1(void *par)
{
        uint32_t led_val = 0;

        static portTickType xLastWakeTime;
        const portTickType xFrequency = pdMS_TO_TICKS(10);

        // Use current time to init xLastWakeTime, mind the difference with␣
↪vTaskDelay()
        xLastWakeTime = xTaskGetTickCount();

        while (1) {
                /* call Freertos system function for 10ms delay */
                vTaskDelayUntil( &xLastWakeTime,xFrequency );

                //####Insert code here###
        }
}

static void task2(void *par)
{
        uint32_t led_val = 0x0001;

        static portTickType xLastWakeTime;
        const portTickType xFrequency = pdMS_TO_TICKS(100);

        // Use current time to init xLastWakeTime, mind the difference with␣
↪vTaskDelay()
        xLastWakeTime = xTaskGetTickCount();

        while (1) {
                /* call Freertos system function for 100ms delay */
                vTaskDelayUntil( &xLastWakeTime,xFrequency );

                //####Insert code here###
        }
}

static void task3(void *par)
{
        uint32_t led_val = 0;

        static portTickType xLastWakeTime;
        const portTickType xFrequency = pdMS_TO_TICKS(200);

        // Use current time to init xLastWakeTime, mind the difference with␣
↪vTaskDelay()
        xLastWakeTime = xTaskGetTickCount();

        while (1) {
                /* call Freertos system function for 100ms delay */
                vTaskDelayUntil( &xLastWakeTime,xFrequency );

                //####Insert code here###
        }
}
```

Steps

Build and run the incompleted code

the code is at `embarc_osp/arc_labs/labs/lab9_freertos`, use an uart terminal console and run the code, you will see a message from program like the one shown below:

```
embARC Build Time: Mar  9 2018, 17:57:50
Compiler Version: Metaware, 4.2.1 Compatible Clang 4.0.1 (branches/release_40)
Create task1 Successfully
Create task2 Successfully
Create task3 Successfully
```

This message implys that three tasks are working correctly.

Implement task 3

It is required for task 3 to retrieve new value from the queue and assign the value to led_val. The LED controls are already implemented in previous labs, so the only new function to learn is `xQueueReceive()`. This is a FreeRTOS API to pop and read an item from queue. Please take reference from FreeRTOS documents and complete the code for this task. (An example is in 'complete' folder)

Implement task 1

It is required for task 1 to check if value from queue is legal. If not, a reset signal is needed to be sent.

Two new functions might be helpful for this task: `xSemaphoreGive()` for release a signal and `xQueuePeek()` for read item but not pop from a queue. Please take reference from FreeRTOS documents and complete the code for this task. (An example is in 'complete' folder)

Do notice the difference between `xQueueReceive()` and `xQueuePeek()`.

Implement task 2

There are two different works for task 2 to complete: to shift led_val and queue it, and to reset both led_val and queue when illegal led_val is detected.

Three functions can be helpful: `xQueueSend()`, `xSemaphoreTake()`, `xQueueReset()`. Please take reference from FreeRTOS documents and complete the code for this task. (An example is in 'complete' folder)

Build and run the completed code

BUild the completed program and debug it to fulfill all requirements. (8-digit running LEDs are used in example code)

Exercises

The problem of philosophers having meal:

Five philosophers sitting at a round dining table. Suppose they are either thinking or eating, but they can't do these two things at same time. So each time when they are having food, they stop thinking and vice versa. There are five forks on the table for eating noddle, each fork is placed between two adjacent philosophers It's hard to eat noddle with one fork, so all philosophers need two forks in order to eat.

Please write a program with proper console output to simulate this process.

3.2.3 Level 3 Labs

AWS IoT Smarthome

Purpose

- Show the smart home solution based on ARC and AWS IoT Cloud
- Learn how to use the AWS IoT Cloud
- Learn how to use the EMSK Board peripheral modules and on-board resources

Equipment

Reguired Hardware

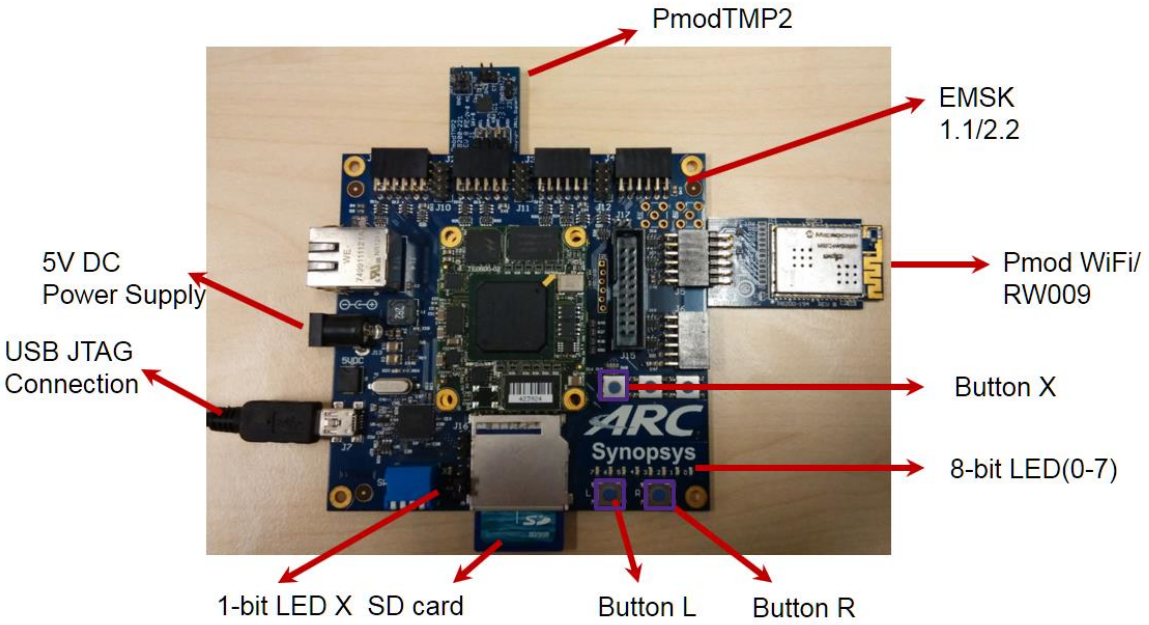
- [DesignWare ARC EM Starter Kit (EMSK)]
- [Digilent PMOD WiFi (MRF24WG0MA)]
- [Digilent PMOD TMP2]
- SD Card
- WiFi Hotspot (default SSID:**embARC**, Password:**qazwsxedc**, WPA/WPA2 encypted)

Required Software

- Metaware or ARC GNU Toolset
- Serial port terminal (e.g. putty, tera-term or minicom)

Hardware Connection (EMSK Board)

- Connect PMOD WiFi to J5, connect PMOD TMP2 to J2.



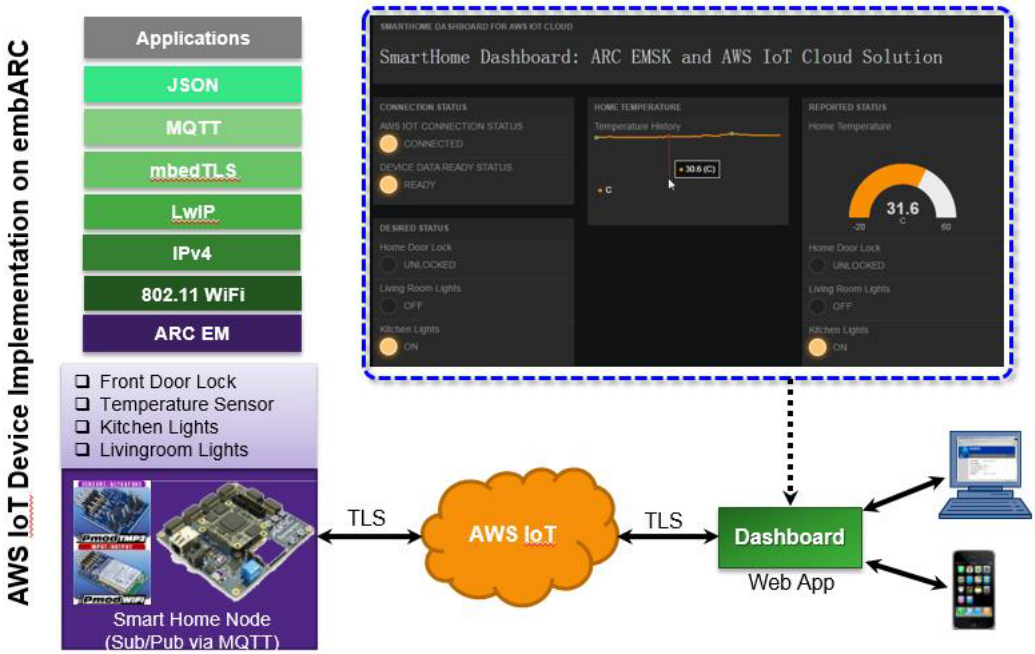
- Required hardware: EMSK 1.1/2.2, PmodTMP2, PmodWiFi, SDCard
- USB UART can be connected to PC for interoperation using NT-Shell.

- Configure your hardware with proper core configuration.
- The hardware resources are described as per the table below.

Hardware Resources	Represent
BUTTON R	Livingroom Lights Control
LED 0-1	Livingroom Lights Status (On or Off)
BUTTON L	Kitchen Lights Control
LED 2-3	Kitchen Lights Status (On or Off)
BUTTON X	Front Door Lock Control
LED 4-5	Front Door Lock Status (On or Off)
LED 7	WiFi connection status (On for connected, Off for not)
LED X	Node working status (toggling in 2s period if working well)
PMOD TMP2	Temperature Sensor
PMOD WiFi	Provide WiFi Connection

Content

This lab provides instructions on how to establish connection between the EMSK and Amazon Web Services Internet of Things (AWS IoT) cloud with a simulated smart home application. With the help of AWS IoT as a intermediate cloud platform, devices can securely interact with cloud applications and other devices. AWS IoT also supports Message Queue Telemetry Transport (MQTT) and provides authentication and end-to-end encryption.



This application is designed to show how to connect only 1 EMSK and AWS IoT Cloud using embARC. The connection between EMSK and AWS IoT Cloud is secured by TLS.

Principles

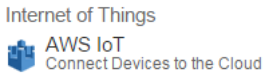
This lab demonstrates the smart home solution based on EMSK by establishing the connection between EMSK Board and AWS IoT Cloud. The AWS IoT Device C SDK for the embedded platform has been optimized and ported for embARC.

In this lab application, the peripheral modules and onboard resources of EMSK board are used to simulate the objects which are controlled and monitored in smart home scenario. The AWS IoT Cloud is used as the Cloud host and control platform that communicate with the EMSK Board using MQTT protocol. A HTML5 Web APP is designed to provide a dash board in order to monitor and control smart home nodes.

Steps

Creating and setting smart home node

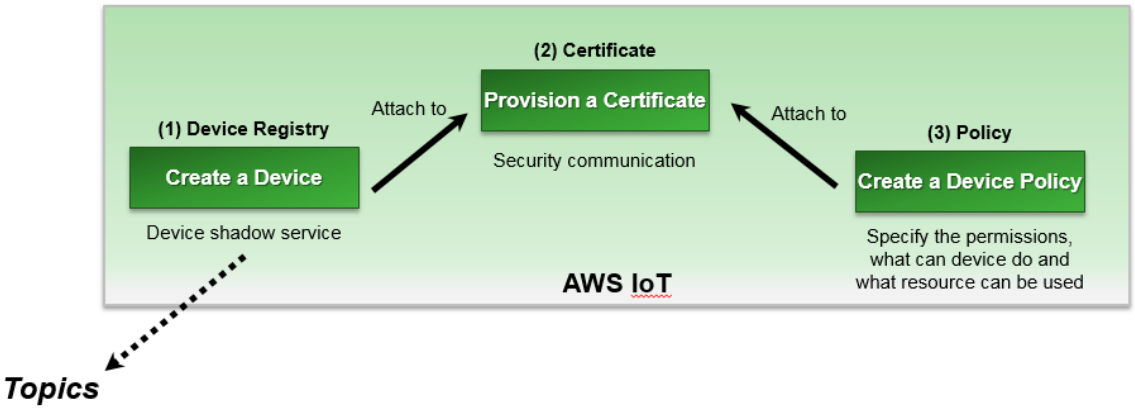
1. Create an AWS account in [here]. Amazon offers various account levels, including a free tier for AWS IoT.
2. Login AWS console and choose AWS IoT.



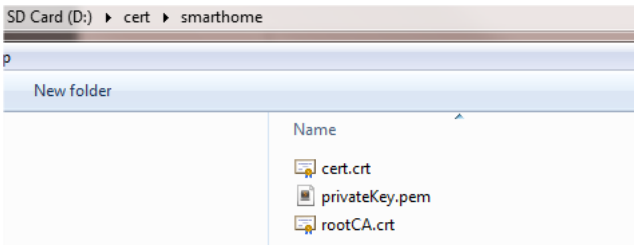
3. Choose an appropriate IoT server in the top right corner of the AWS IoT console page.

4. Create your smart home node in the thing registry and generate X.509 certificate for the node. Create an AWS IoT policy. Then attach your smart home node and policy to the X.509 certificate.

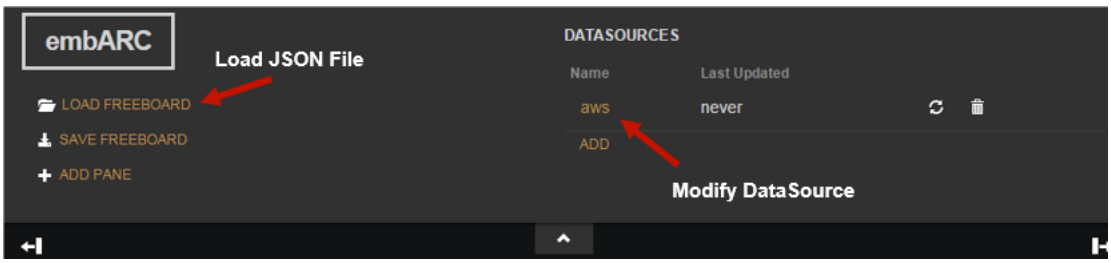
Note: for more details [Using a Smart Home Iot Application with EMSK]



5. Download the root CA certificate from [here]. Rename it *rootCA.crt*. Copy the certificate files *cert.crt, privateKey.pem and rootCA.crt* to folder *cert/smarthome*. Insert the SD card to your PC, and copy the certificate folder cert to the SD Card root.



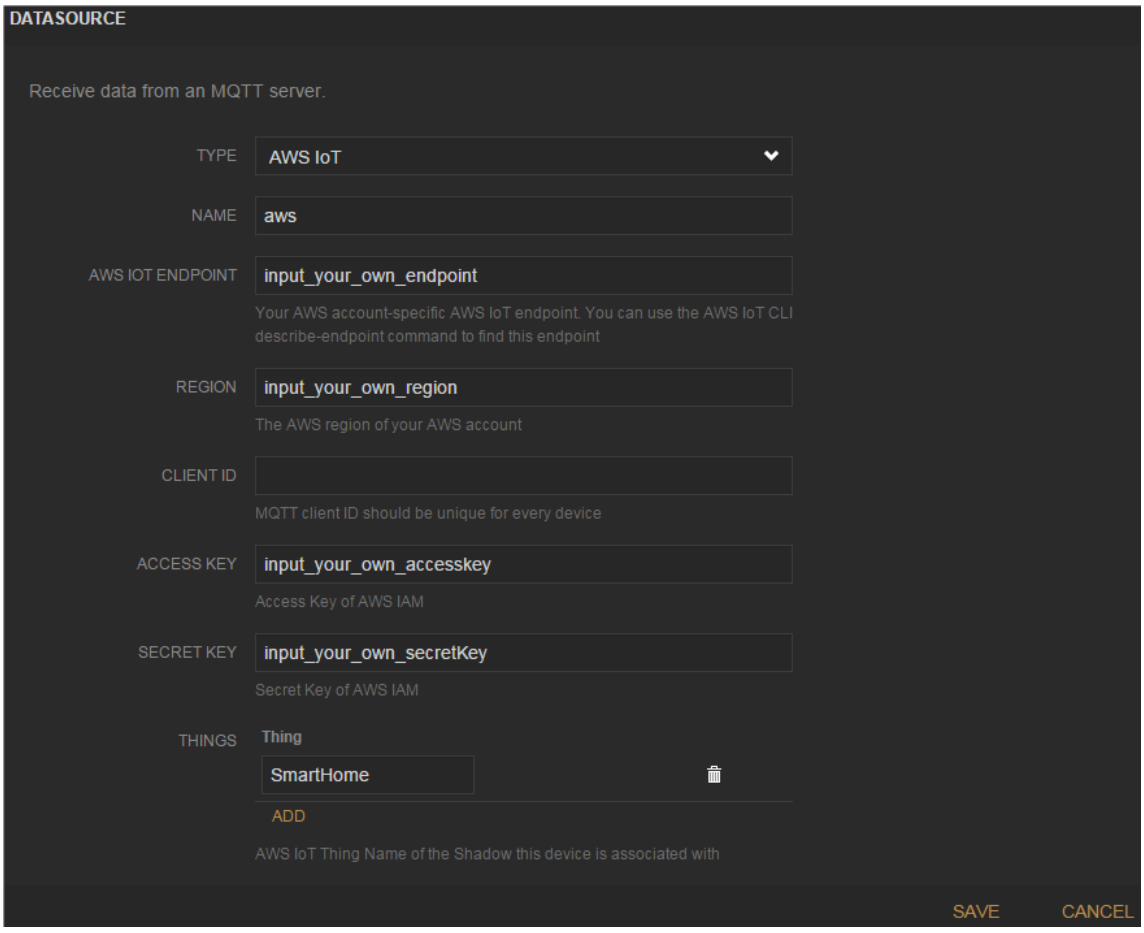
6. Open the [Web App] in a web browser and load the configuration file dashboard-smarthomesinglething.json obtained from [embARC/example/freertos/iot/aws/smarthome_demo]. The dashboard can be loaded automatically



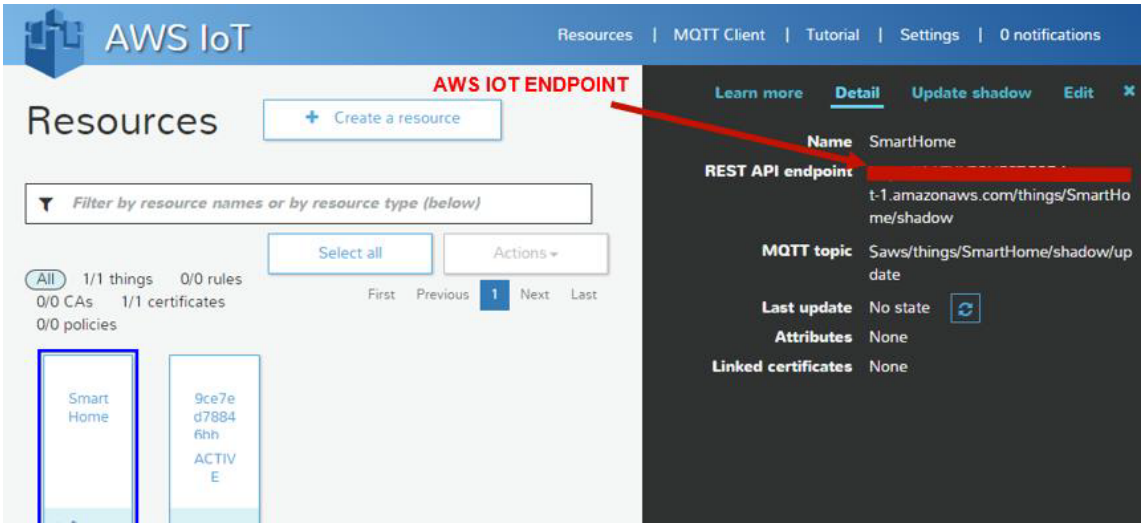
7. Click "ADD" to go to DATASOURCE page and fill up the forms.

a) TYPE: Choose AWS IoT.

b) NAME: Name is aws.



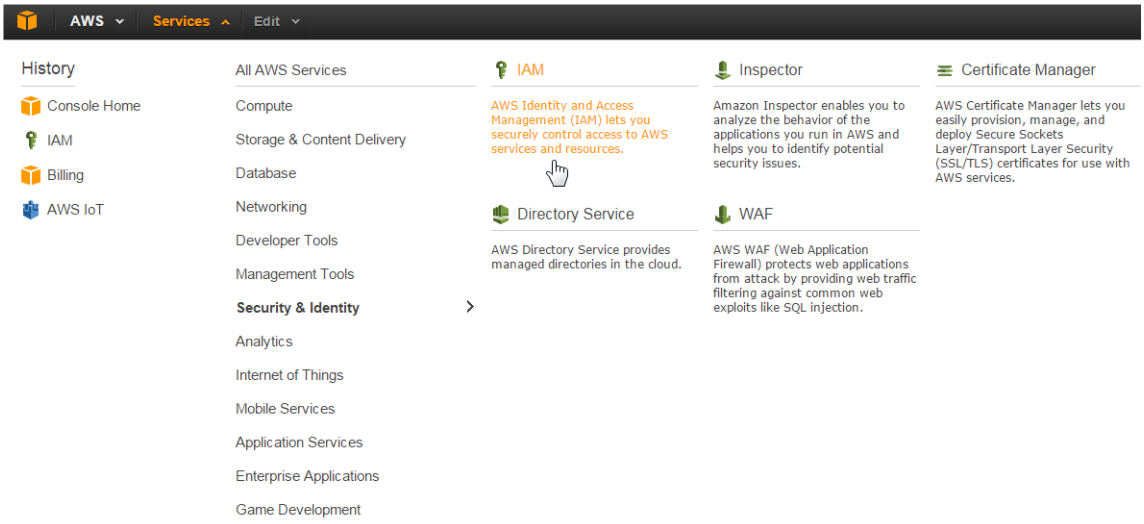
c) AWS IOT ENDPOINT: Go to AWS IoT console and click your smart home node "SmartHome". Copy the content `XXXXXXXXXXXXXX.iot.us-east-1.amazonaws.com` in REST API endpoint to AWS IOT ENDPOINT.



d) REGION: Copy the AWS region of your smart home node in REST API endpoint to REGION. For example, `https://XXXXXXXXXXXXX.iot.us-east1.amazonaws.com/things/SmartHome/shadow`. REGION is us-east-1.

e) CLIENT ID: Leave it blank as default.

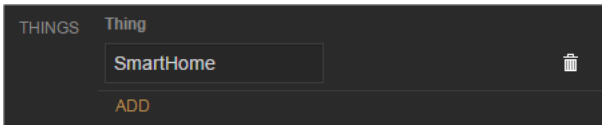
f) ACCESS KEY and SECRET KEY: Go to AWS Services page and click "IAM".



Go to User page and click "Create New Users". Enter User Names "AWSIoTUser". Then download user security credentials, Access Key ID and Secret Access Key. Copy Access Key ID to ACCESS KEY and Secret Access Key to SECRET KEY.

Go to User page and click “AWSIoTUser”. Click “Attach Policy” to attach “AWSIoTDataAccess” to “AWSIoTUser”.

g) THINGS: AWS IoT thing name “SmartHome”.



h) Click “Save” to finish the setting.

Building and running AWS IoT smart home example

1. The AWS IoT thing SDK for C has been ported to embARC. Check the above steps in order for your IoT application to work smoothly. Go to *embARC/example/freertos/iot/aws/smarthome_demo*. Modify aws_iot_config.h to match your AWS IoT configuration. The macro **AWS_IOT_MQTT_HOST** can be copied from the REST API endpoint in AWS IoT console. For example, `https://XXXXXXXXXXXXXX.iot.us-east-1.amazonaws.com/things/SmartHome/shadow`. **AWS_IOT_MQTT_HOST** should be `XXXXXXXXXXXXXX.iot.us-east-1.amazonaws.com`.

```
// Get from console
// =============================================
#define AWS_IOT_MQTT_HOST              "XXXXXXXXXXXXXX.iot.us-east-1.amazonaws.com" ///< Cus
#define AWS_IOT_MQTT_PORT              8883 ///< default port for MQTT/S
#define AWS_IOT_MQTT_CLIENT_ID         "csdk-SH" ///< MQTT client ID should be unique for ev
#define AWS_IOT_MY_THING_NAME          "SmartHome" ///< Thing Name of the Shadow this device
#define AWS_IOT_ROOT_CA_FILENAME       "rootCA.crt" ///< Root CA file name
#define AWS_IOT_CERTIFICATE_FILENAME   "cert.crt" ///< device signed certificate file name
#define AWS_IOT_PRIVATE_KEY_FILENAME   "privateKey.pem" ///< Device private key filename
// =============================================
```

2. Use USB cable to connect the EMSK board. Set the baud rate of the terminal emulator to 115200.

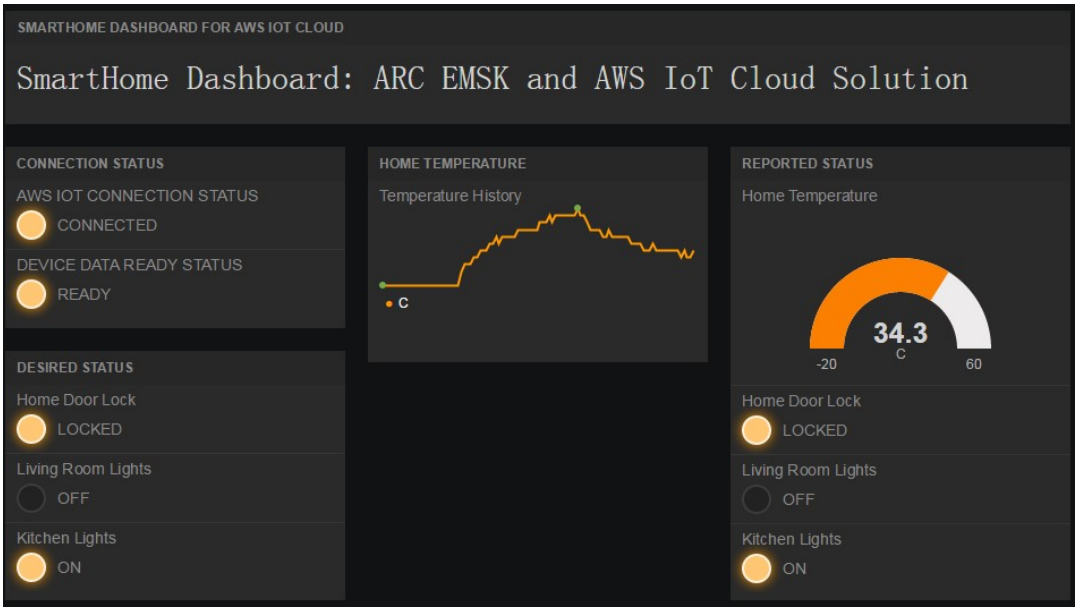
3. Insert the SD Card into the EMSK board SD Card slot. Run the AWS IoT application using JTAG. Go to *embARC/example/freertos/iot/aws/smarthome_demo* in command line, input the compile command as follow:

```
make TOOLCHAIN=gnu BD_VER=22 CUR_CORE=arcem7d run
```

4. FreeRTOS-based runtime environment can be loaded automatically. Wait for WiFi initialization and connection establishment (30 seconds or less) until the “WiFi connected” message is shown in the terminal emulator. “Network is ok” will be shown after the certificate files cert.crt, privateKey.pem and rootCA.crt are validated. The information in “reported”: {} is the state of the EMSK-based smart home node. “Updated Accepted !!” means the connection works between the smart home node and AWS IoT cloud.

```
Shadow Init
Shadow Connect
FrontDoor is open
Turn off KitchenLights
Turn off LivingRoomLights
Update Shadow: {"state":{"reported":{"temperature":29.600000,"DoorLocked":false,
"KitchenLights":false,"LivingRoomLights":false}}, "clientToken":"csdk-SH-0"}
********************************************************************************
*********

Delta - FrontDoor state changed to 1
FrontDoor is locked
Delta - KitchenLights light state changed to 1
Turn on KitchenLights
Update Accepted !!
Update Shadow: {"state":{"reported":{"temperature":29.600000,"DoorLocked":true,"
KitchenLights":true,"LivingRoomLights":false}}, "clientToken":"csdk-SH-1"}
********************************************************************************
*********

Update Accepted !!
Update Shadow: {"state":{"reported":{"temperature":29.600000,"DoorLocked":true,"
KitchenLights":true,"LivingRoomLights":false}}, "clientToken":"csdk-SH-2"}
********************************************************************************
*********
```

4. Interact using EMSK and Dashboard. You can press the button L/R/X to see the led changes on board and also on dashboard web app. You can also click the lights of DESIRED STATUS pane on the dashboard app, and see the led changes on board and dashboard web app.



Exercises

This application is designed to show how to connect only 1 EMSK and AWS IoT Cloud using embARC. Try to add more nodes and implement a Multi-nodes AWS IoT Smarthome Demo.

Note: Related demo codes you can find [here]

CHAPTER

FOUR

APPENDIX

CHAPTER

FIVE

INDICES AND TABLES

- genindex
- search