
ARC labs handbook

Release 2018.09

Synopsys

2023

Contents:

1	Overview	1
1.1	Introduction	1
1.2	Supported Hardware Platform	2
1.3	Reference	2
2	Getting Started	3
2.1	Software Requirement	3
2.2	Install Software Tools	4
2.3	Final Check	7
2.4	Learn More	7
3	Hands-on labs	9
3.1	Basic labs	9
3.2	Advanced labs	49
3.3	Exploration	85
4	Appendix	93
4.1	Reference	93
5	Indices and tables	95

1.1 Introduction

This is a handbook for ARC labs which is a part of ARC university courses. The handbook is written to help students who attend the ARC university course. Anyone interested in DesignWare® ARC® processors can also take this handbook as a quick start-up to get started in DesignWare® ARC® processors development. In this handbook, all the basic elements of ARC are described in the labs with a 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 Computer Science, Computer Engineering, Electrical Engineering, or for professional engineers.

This handbook includes a series of labs (more labs will be added in the future), which are roughly classified into 3 levels:

- *Level 1: ARC basic*

The labs in this level cover the basic topics of DesignWare® ARC® processors. For example, the installation and usage of hardware and software tools, software or hardware development kits, the first hello world example, interrupt handling and internal timers of DesignWare® ARC® processors, and so on.

- *Level 2: ARC advanced*

The labs in this level cover the advanced topics of DesignWare® ARC® processors. For example, Real-Time Operating System (RTOS), customized linkage, compiler optimization, basic applications, DesignWare® ARC® processors DSP feature, and so on.

- *Level 3: ARC exploration*

The labs in this level cover some complex applications of DesignWare® ARC® processors. For example, Internet of Things (IoT) application, embedded machine learning, and so on.

Most of the 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 Free and open-source software (FOSS) embedded software stacks to DesignWare® ARC® processors.

For more details about embARC OSP, please see its [online docs](#).

1.2 Supported Hardware Platform

The following DesignWare® ARC® processors based hardware platforms are supported in this handbook.

- [ARC EM Starter Kit](#)
- [ARC IoT Development Kit](#)

You can click the above links to get the platform's data sheet and user manual as a reference.

1.3 Reference

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

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

2.1 Software Requirement

- **ARC Development Tools** Select MetaWare Development Toolkit or GNU Toolchain for ARC Processors from the following list according to your requirement.
 - MetaWare Development 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 or 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 with restrictions such as code-size limit of 32 Kilobytes and no runtime library sources. It is available for Microsoft Windows only.
 - GNU Toolchain for ARC Processors
 - * **Open Source ARC GNU IDE (2018.03)**. The GNU Toolchain for ARC Processors offers all of the benefits of open source tools such as 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. One USB cable is required and external debugger is not required.
- **Tera Term** or **PuTTY** for serial terminal connection. The serial configurations are 115200 baud, 8 bits data, 1 stop bit, and no parity (115200-8-N-1) by default.

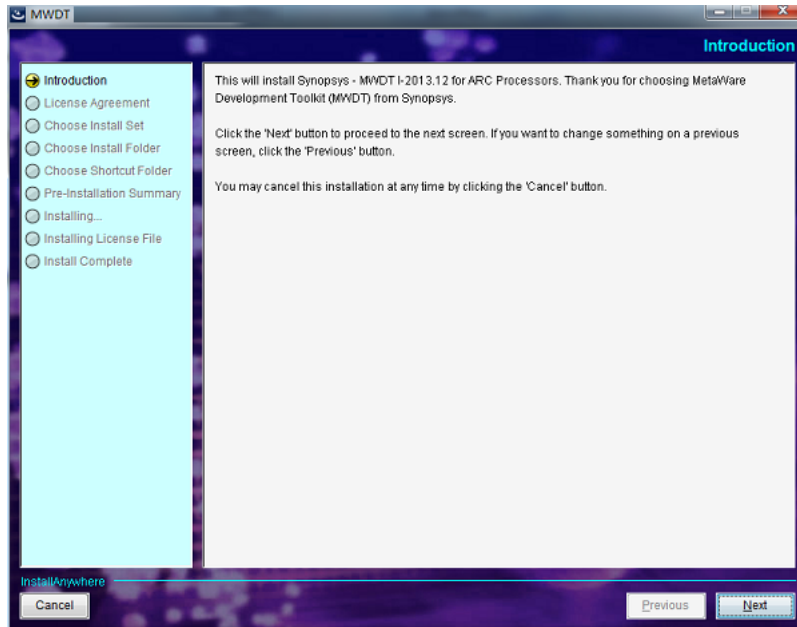
Note: If using embARC with GNU toolchain on Windows, please install **Zadig** to replace FTDI driver with WinUSB driver. See [How to Use OpenOCD on Windows](#) for more information. If you want to switch back to Metaware toolchain, make sure you switch back the usb-jtag driver from WinUSB to FTDI driver.

2.2 Install Software Tools

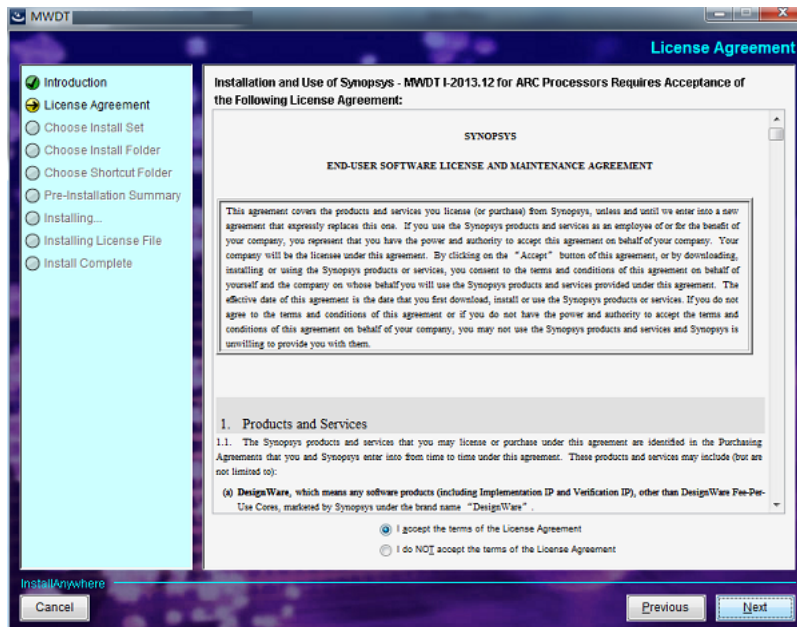
2.2.1 Install MetaWare Development Toolkit

Installing MetaWare Development Toolkit (2017.09).

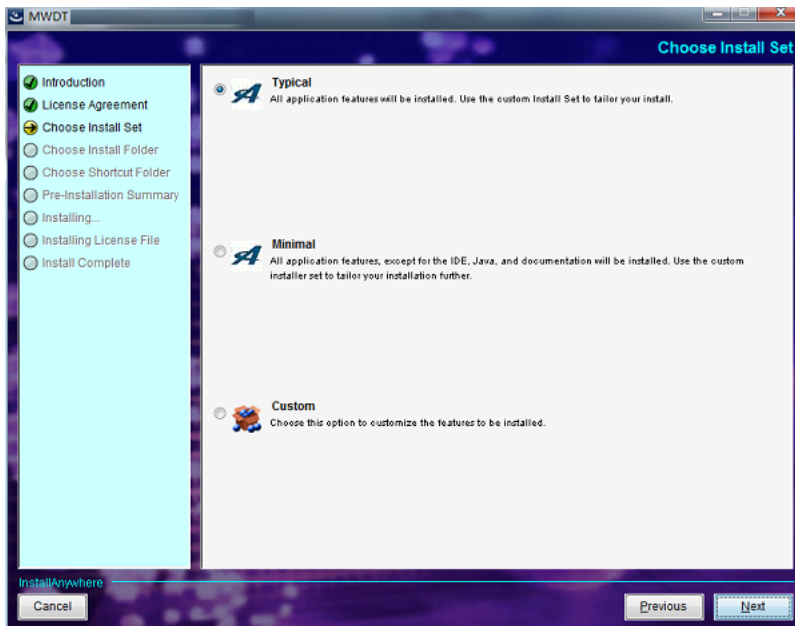
1. Double click the `mw_dekit_arc_i_2017_09_win_install.exe` and click **Next**.



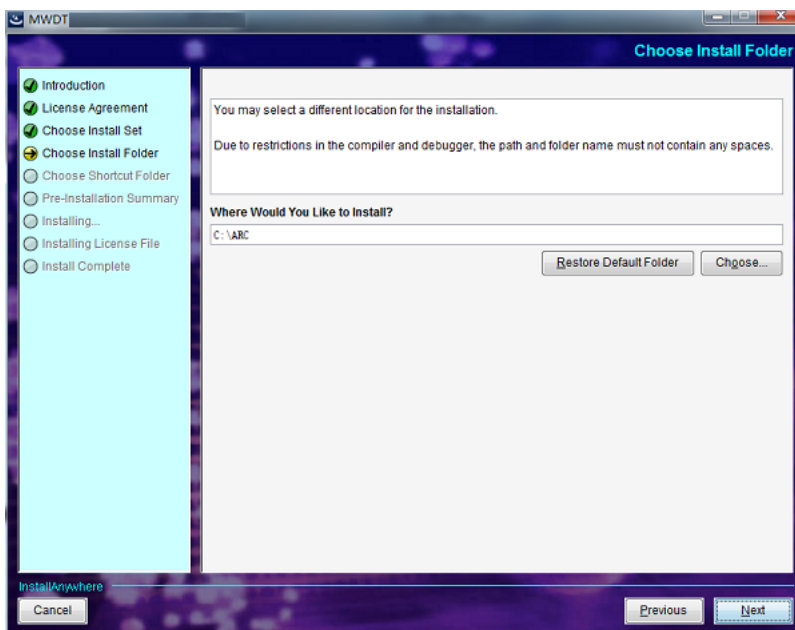
2. Select I accept and click **Next**.



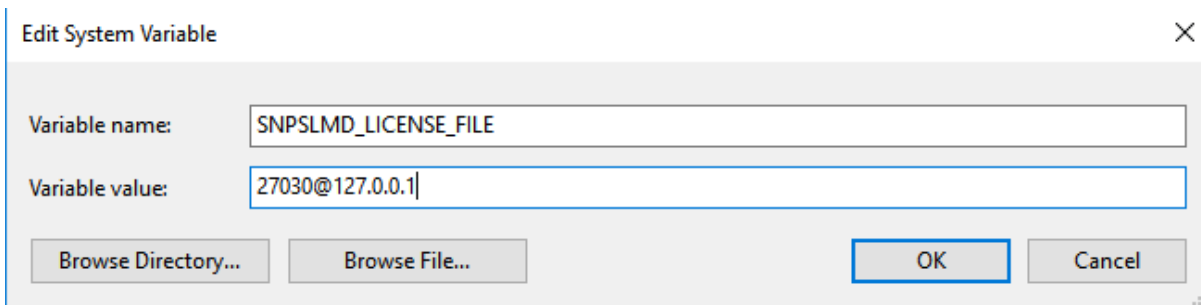
3. Select Typical installation and click **Next**.



4. Set the install path (make sure you use English letters without any space) and click **Next** until the installation is complete.



5. Set the license file (SNPSLMD_LICENSE_FILE) for MetaWare Development Toolkit. It can be a real file containing license or a license server.
 - For Windows, go to **Computer > Properties > Advanced > Environment Variables > System Variables > New**.



- For Linux, add SNPSLMD_LICENSE_FILE into your system variables.

6. Test the MetaWare Development Toolkit and the license

At the command prompt, compile and link in one step.

For example, find the queens.c in the demos folder of MetaWare Development Toolkit installation directory.

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

If you get the following message without any error, then the MetaWare Development Toolkit is successfully installed.

```
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 GNU Toolchain for ARC Processors

Click (<https://github.com/foss-for-synopsys-dwc-arc-processors/toolchain/releases>) to get the latest version of GNU Toolchain for ARC Processors.

To use and install GNU Toolchain for ARC Processors, please see (<http://embarc.org/toolchain/ide/index.html>).

It is recommended to install GNU Toolchain for ARC Processors 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 through git. There are scripts in this repo that you are need to set up your development environment, and Git is used to get this repo. If you do not have Git installed, see the beginning of the OS-specific instructions below for help.

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. Please use git to clone or download the ARC labs to the root folder of embARC OSP. If the download is successful, the following folder structure is displayed:

```
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 MetaWare Development Toolkit or GNU Toolchain for ARC Processors are added to the system variable **PATH** in your environment variables.
- It is recommended to install GNU Toolchain for ARC Processors to default location. Otherwise, you need to make additional changes as described.
 - If running and debugging embARC applications using **arc-elf32-gdb** and **OpenOCD for ARC**, make sure the path of **OpenOCD** is added to the **PATH** in your environment variables and modify **OPENOCD_SCRIPT_ROOT** variable in `<embARC>/options/toolchain/toolchain_gnu.mk` 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 makefile-based. *make/gmake* is provided in the MetaWare Development Toolkit (gmake) and GNU Toolchain for ARC Processors (make)

2.4 Learn More

For more details about embARC OSP, see [online docs](#)

3.1 Basic labs

3.1.1 How to use ARC IDE

MetaWare Development Toolkit

Purpose

- To learn MetaWare Development Toolkit
- To get familiar with the basic usage of the MetaWare Development Toolkit
- To get familiar with the features and usage of the MetaWare Debugger (mdb)

Requirements

The following hardware and tools are required:

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

Content

- Create a C project using the MetaWare Development Toolkit
- Import the code `CoreTest.c` from `embarc_osp/arc_labs/labs/lab_core_test`
- Configure compilation options to compile, and generate executable files.
- Start the debugger of MetaWare Development Toolkit and enter debug mode.

From two different perspectives of C language and assembly language, use the methods of setting breakpoint, single-step execution, full-speed executions, etc., combined with observing PC address, register status, global variable status, and profiling performance to analyze and debug the target program.

Principles

Use the MetaWare Development Toolkit to create projects and load code. In the engineering unit, configure the compilation options to compile 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
}
```

(continues on next page)

(continued from previous page)

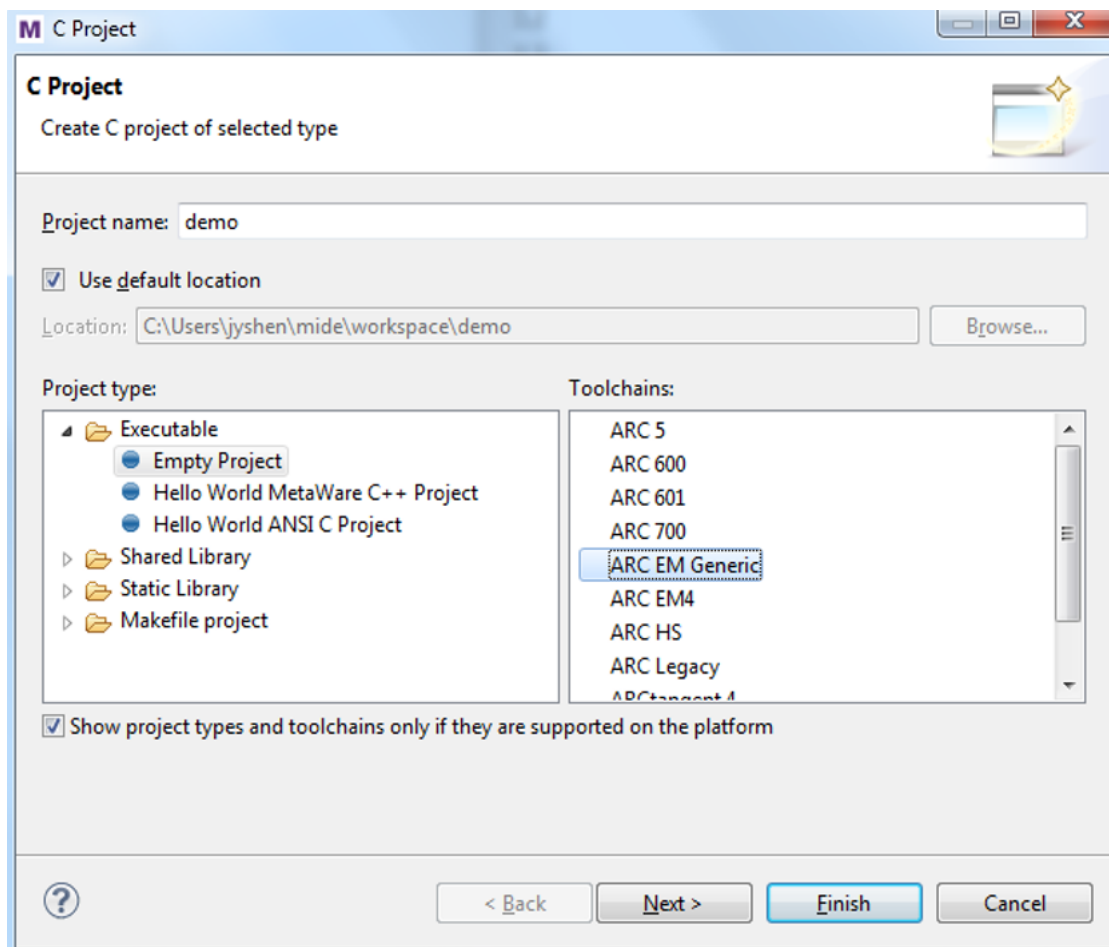
```
#endif

return 0;
}
```

Steps

1. Create a project

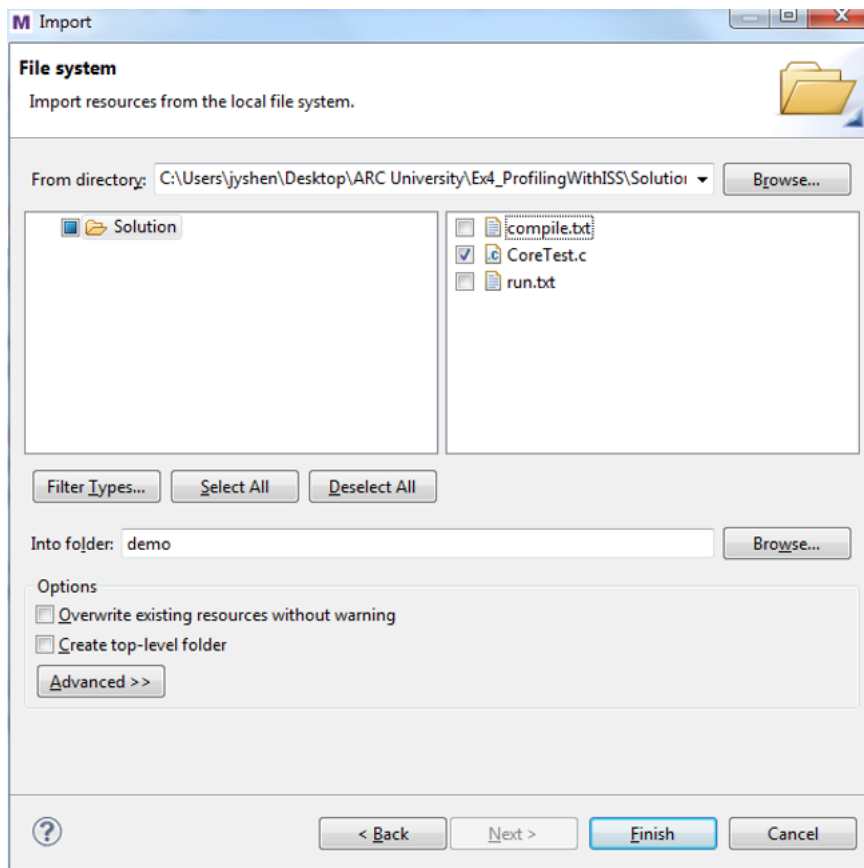
Open the MetaWare Development Toolkit, create an empty project called `demo`, and select the **ARC EM Generic** processor.



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

In the Project Explorer, click `demo` and select **Import**.

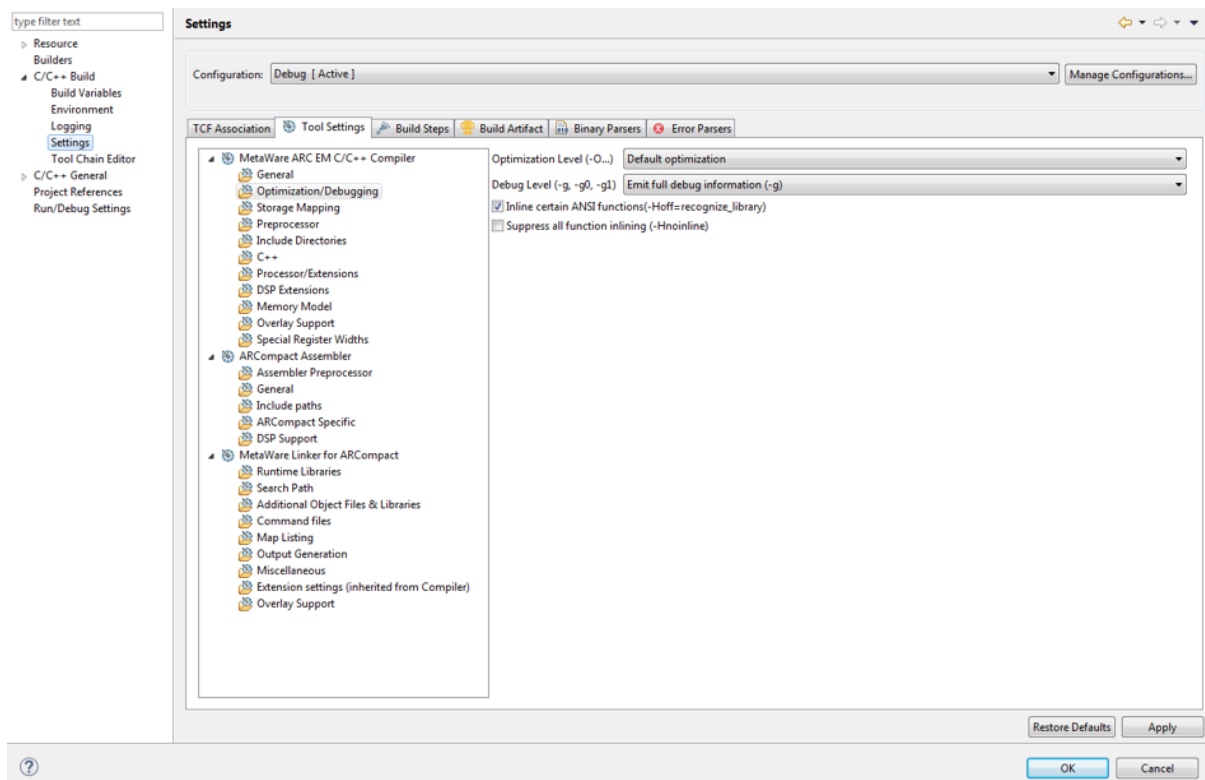
In the Import wizard, select **File system** from the **General** tab, then click **Next**. As shown in the following figure, in the From directory field, type or browse to select the directory contain the file `CoreTest.c`. Recent directories that have been imported from are shown on the From directory field's combo box. In the left pane, check a folder that will import its entire contents into the Workbench, and in the right pane check the file `CoreTest.c`.



Click **Finish** when done, the file `CoreTest.c` is now shown in the one of the navigation views in the project `demo`.

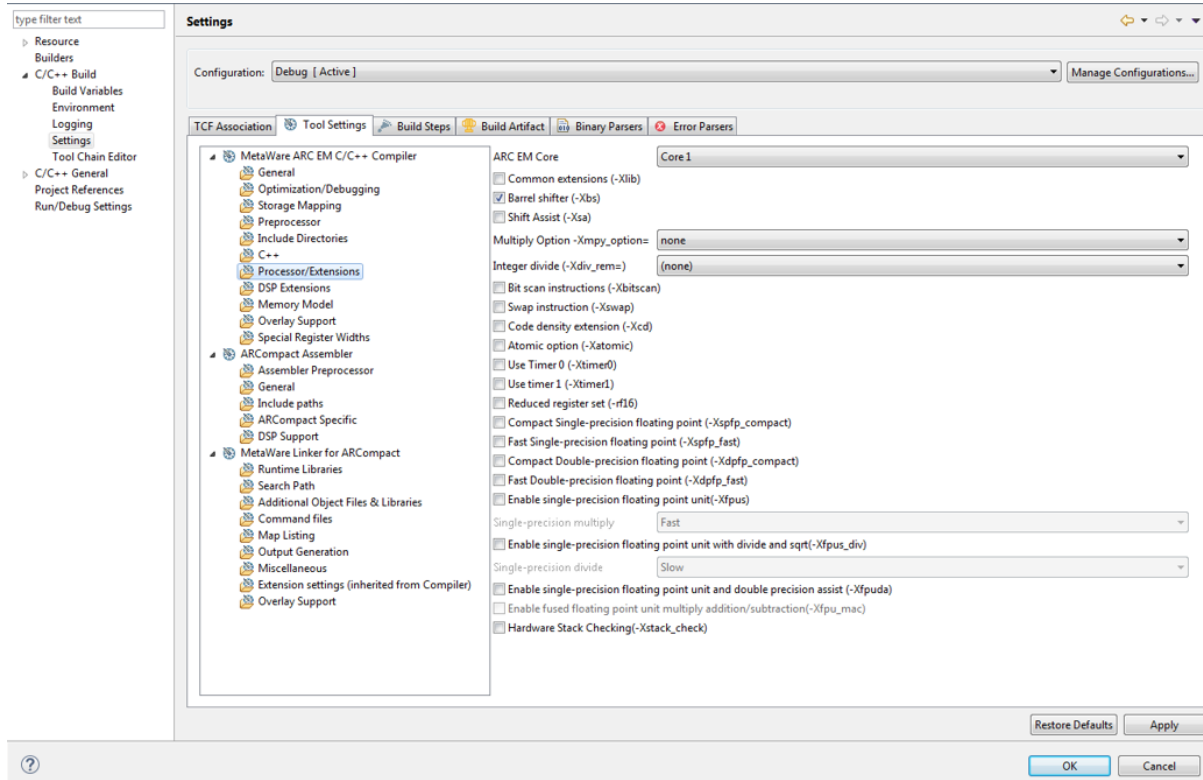
3. Set compilation options

From the Project Explorer view, right-click the project `demo` and choose Properties. Click **C/C++ Build > Settings > Tool Settings** menu options. The **Tool Settings** dialog opens.




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.

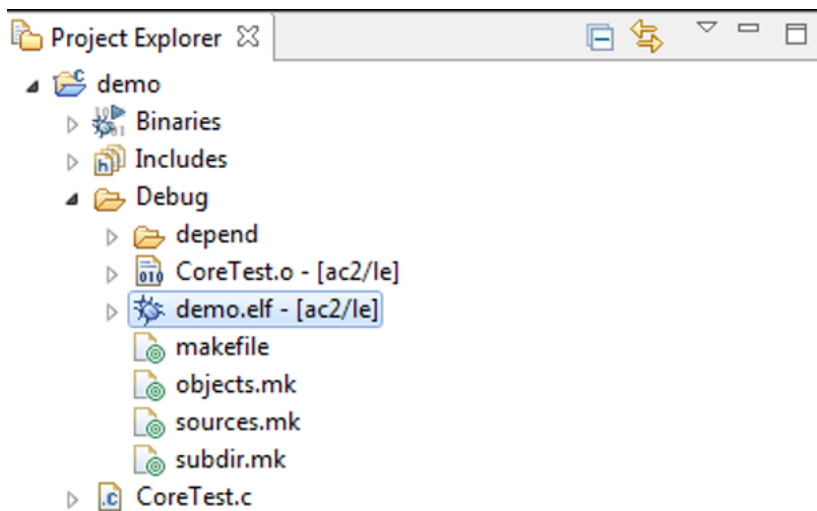
Select **Processor/Extensions** to set the compilation options corresponding to the target processor hardware properties, such as the version of the processor, whether to support extended instructions such as shift, multiplication, floating-point operations, and so on whether to include Timer0/1. As shown in the following figure, this setting indicates that the target processor supports common extended instructions.



Select **MetaWare ARC EM C/C++** and check the settings compilation options. Click **OK** when done.

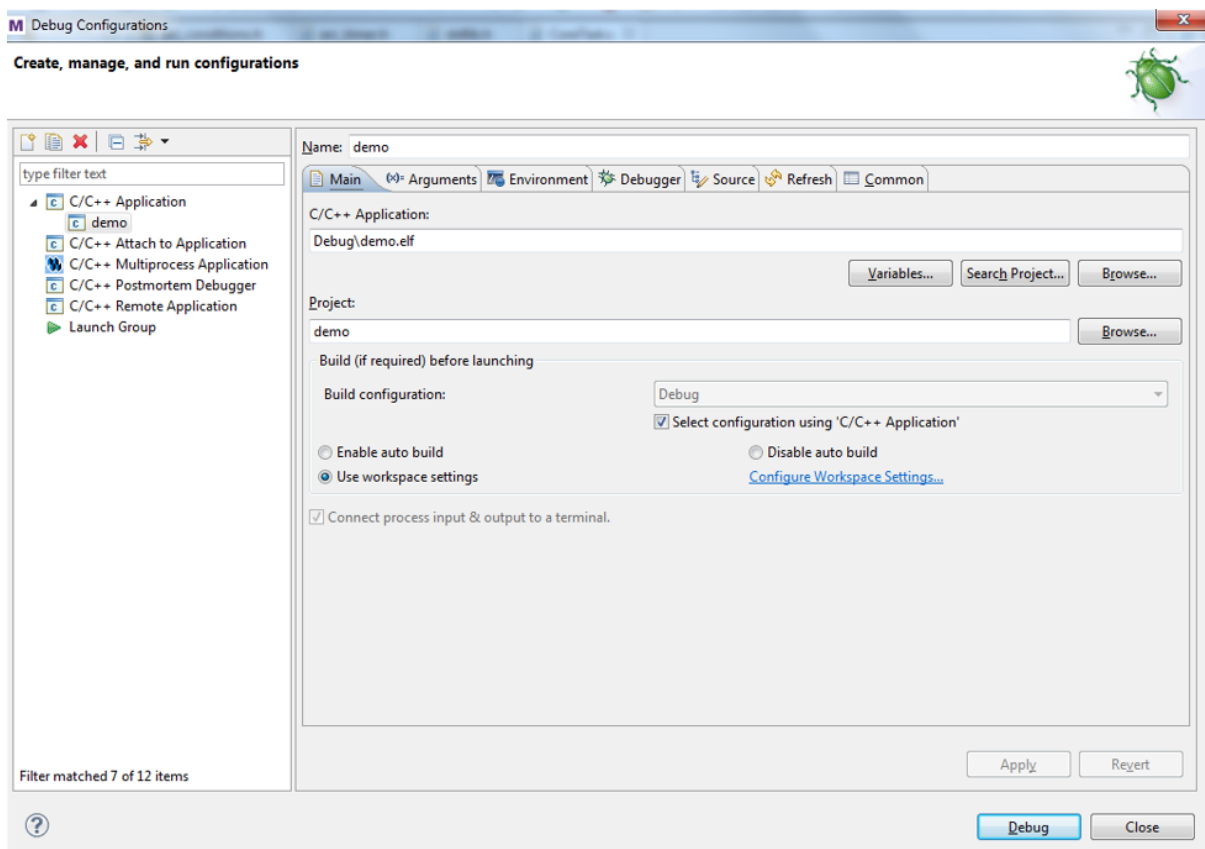
4. Build project

In the Project Explorer view, select project `demo`. Click **Project > Build Project** or click the icon  on the toolbar. In the MetaWare Development Toolkit main interface, you can see in the **Console** view the output and results of the build command. Click on its tab to bring the view forward if it is not currently visible. If for some reason it's not present, you can open it by selecting **Window > Show View > Console**. When the message `Finished building target: demo.elf` is displayed, the compilation is successful, and the compiled executable file `demo.elf` can be seen in the Project Explorer.



5. Set debug options

Click the **Run > Debug Configurations...** menu option to open the **Debug Configurations** dialog. Double-click **C/C++ Application** or right-click **New** to create a new launch configuration.

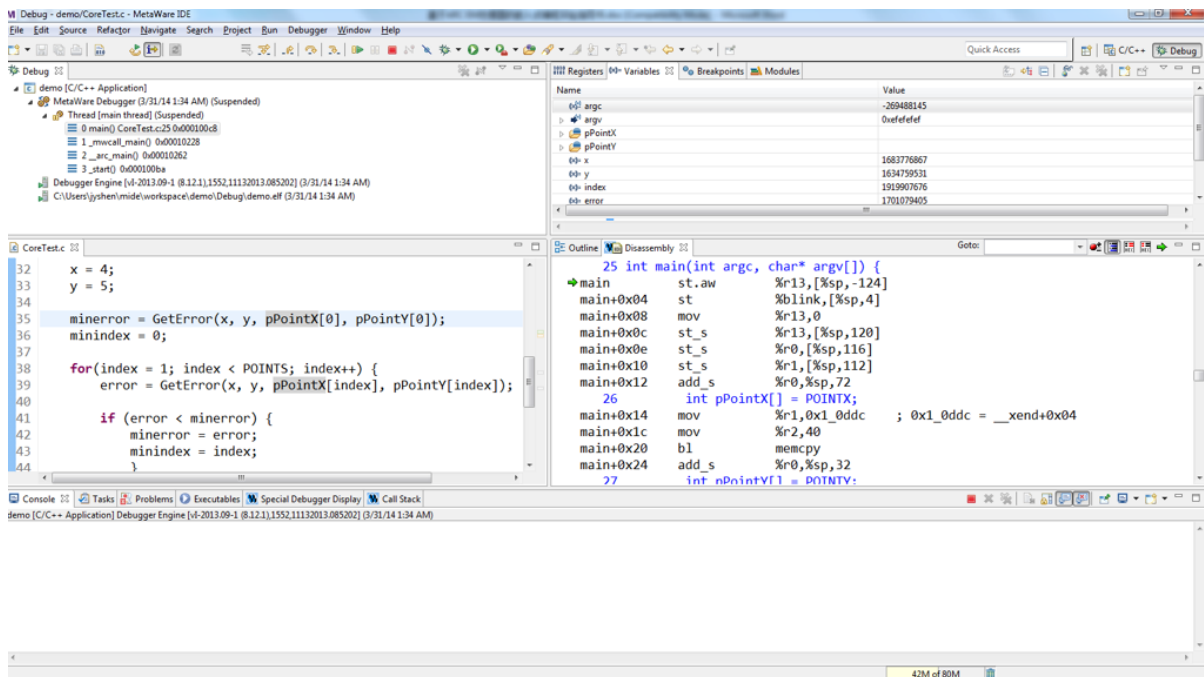


If a project is selected in the Project Explorer view all data is automatically entered, take a moment to verify its accuracy or change as needed. Here you do not need to make any changes, just click **Debug** to enter the debugging interface.






6. Debug executable file demo.elf

You may be prompted to switch to the **Debug** perspective. Click **Yes**.

The Debug perspective appears with the required windows open. And the windows can be source code window, assembly code window, register window, global variable window, breakpoint window, function window, and so on.






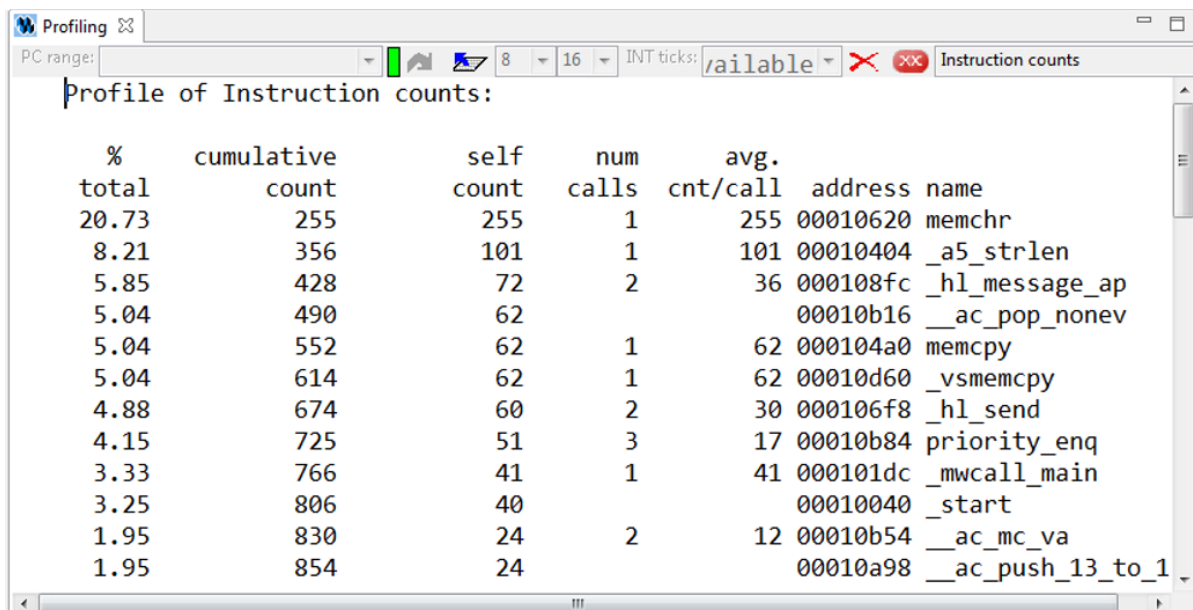
In the C code window, right-click the code line number on the left side of the window, select **Toggle Breakpoint** or double-click the line number to set a breakpoint on. In the assembly code window, double-click a line of code to set a breakpoint on. You'll see a blue circle there indicating the breakpoint is set.

After the breakpoint is set, click **Run > Resume** or you can use the **Resume** button  on the toolbar of the Debug view to run the program. The program runs directly to the nearest breakpoint. You can observe the current program execution and the relevant status information of the processor through the various windows as described in 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 . If you want to return to the main MetaWare Development Toolkit page, click C/C++ .

7. Code performance analysis using the debugger

Based on the previous project demo, open the **Compile Options** dialog box in step 3 and set the Optimization





Level to -O0 in the **Optimization/Debugging** column. Then click  to recompile the project, and click  to enter the debugging interface. Click **Debugger** in the main menu of the debugging interface, select **Disassembly** from the menu 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 .

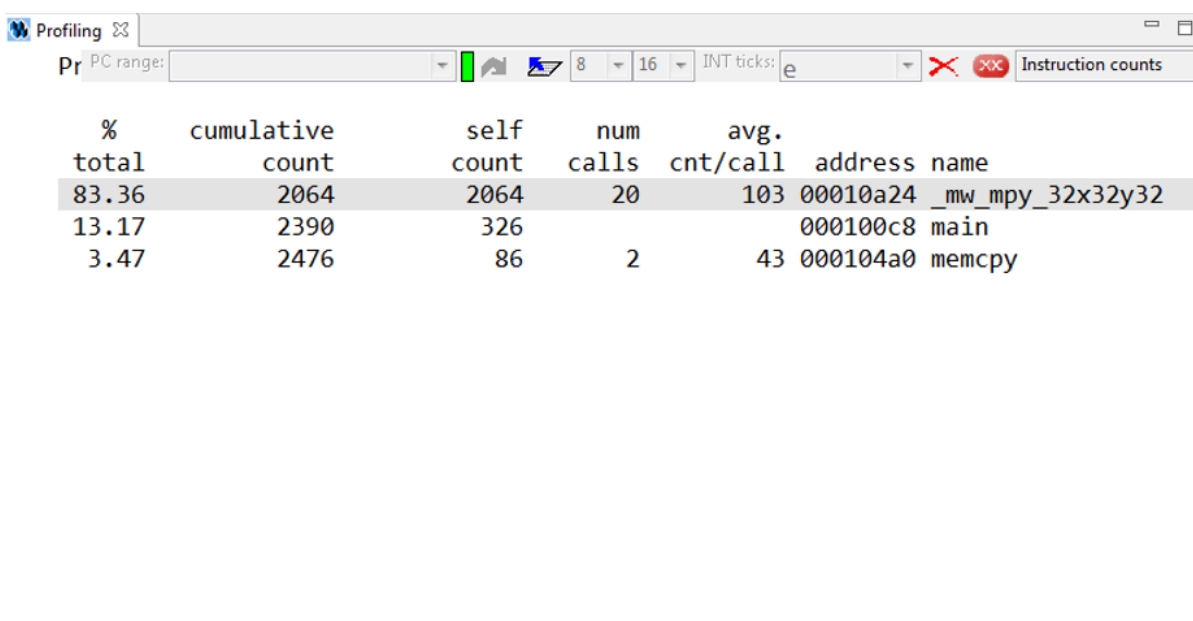


Profile of Instruction counts:

% total	cumulative count	self count	num calls	avg. cnt/call	address	name
20.73	255	255	1	255	00010620	memchr
8.21	356	101	1	101	00010404	_a5_strlen
5.85	428	72	2	36	000108fc	_hl_message_ap
5.04	490	62			00010b16	__ac_pop_nonev
5.04	552	62	1	62	000104a0	memcpy
5.04	614	62	1	62	00010d60	_vsmemcpy
4.88	674	60	2	30	000106f8	_hl_send
4.15	725	51	3	17	00010b84	priority_enq
3.33	766	41	1	41	000101dc	_mwcall_main
3.25	806	40			00010040	_start
1.95	830	24	2	12	00010b54	__ac_mc_va
1.95	854	24			00010a98	__ac_push_13_to_1

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.

Use this project as an example to continue to introduce the use of the **Profiling** window. The program is paused at the entrance of the main() function and the **Profiling** window opens. The main() function is the main object of performance analysis optimization. The content displayed in the **Profiling** window is some function information initialized by the processor before the main() function is executed. Click  in the **Profiling** window to clear the current information. If you click , nothing is displayed, and it indicates that the cleaning is successful. Set a breakpoint at the last statement of the main() function (either C statement or assembly statement), and click  in the toolbar above the debug interface to let the program run until it hits the breakpoint. Click  in the **Profiling** window, and only the information related to the main() function is displayed. Therefore, flexible setting of breakpoints, combined with the clear function, can perform performance analysis on the concerned blocks.



Pr PC range:

% total	cumulative count	self count	num calls	avg. cnt/call	address	name
83.36	2064	2064	20	103	00010a24	_mw_mpy_32x32y32
13.17	2390	326			000100c8	main
3.47	2476	86	2	43	000104a0	memcpy

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 spends 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 consider how you can use fewer instructions and implement multiplication more efficiently.

Exercises

Enable MPY extension instructions by setting Multiply Option `-Xmpy_option = wlh1` in step 3, it implements multiplication more efficiently with fewer instructions. Redo steps 4 - 7 to analyze with the debugger's Profiling function, observe the total number of instructions consumed by the main function, and compare it with the above Profiling result.

GNU Toolchain for ARC Processors

Purpose

- Learn the GNU Toolchain for ARC Processors
- Familiar with the GNU Toolchain for ARC Processors
- Familiar with the functions and usage of the GNU Toolchain for ARC Processors debugger

Requirements

The following hardware and tools are required:

- PC host
- GNU Toolchain for ARC Processors
- nSIM simulator or ARC board (EM Starter Kit / IoT Development Kit)
- `embarc_osp/arc_labs/labs/lab_core_test`

Content

- Create a C project using GNU Toolchain for ARC Processors
- Import the code `CoreTest.c` from `embarc_osp/arc_labs/labs/lab_core_test`
- Configure compilation options to compile, and generate executable files
- Start the GNU Toolchain for ARC Processors debugger to enter the debug mode

From two different perspectives of C language and assembly language, use the methods of setting breakpoint, single-step execution, full-speed executions, and so on combined with observing PC address, register status, global variable status, and profiling performance to analyze and debug the target program.

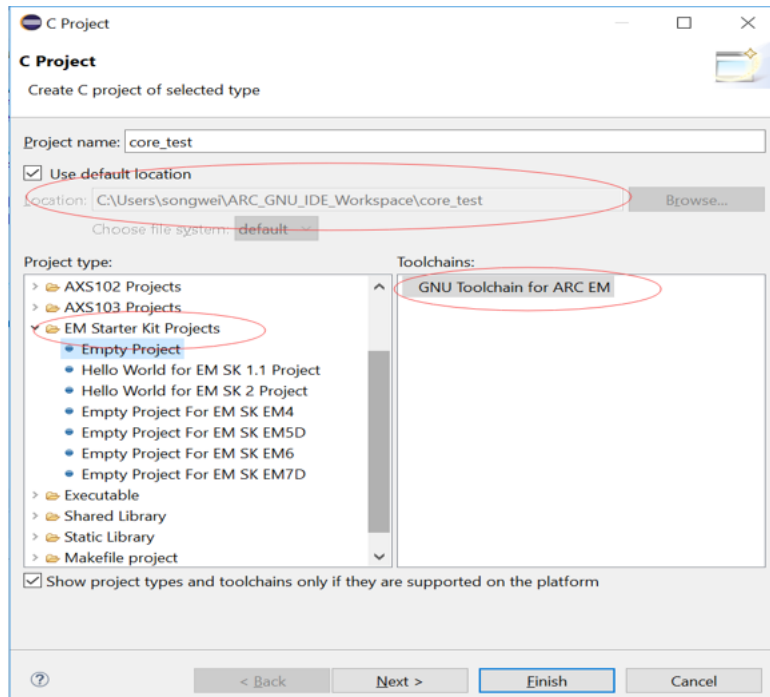
Principles

Use the GNU Toolchain for ARC Processors 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

1. Create a project

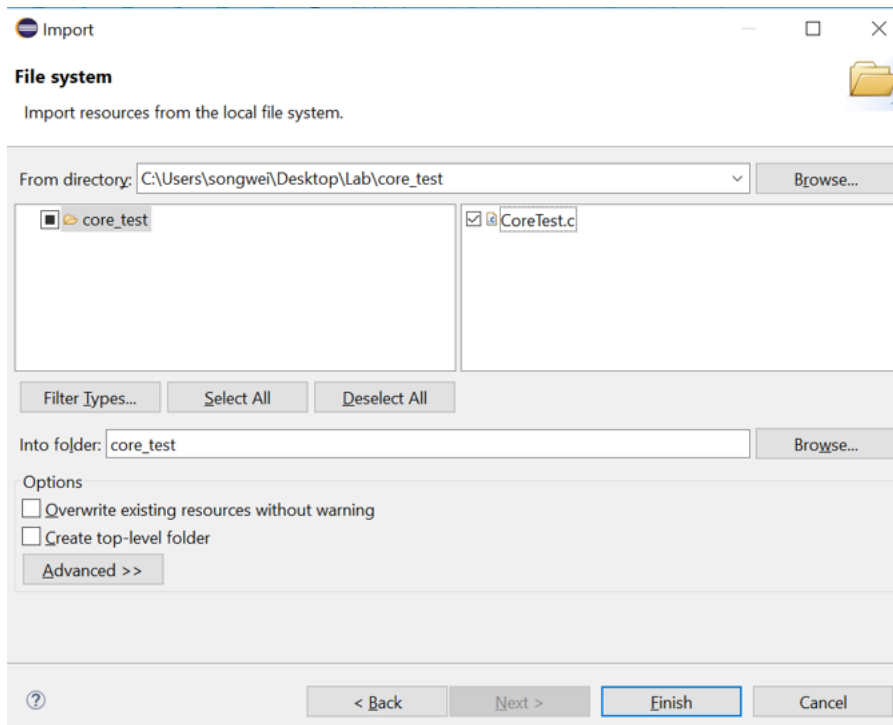
Open the GNU Toolchain for ARC Processors, create an empty project called `core_test`, and select **ARC EM series processor**.



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

In the Project Explorer, right-click `core_test`, and select **Import..**

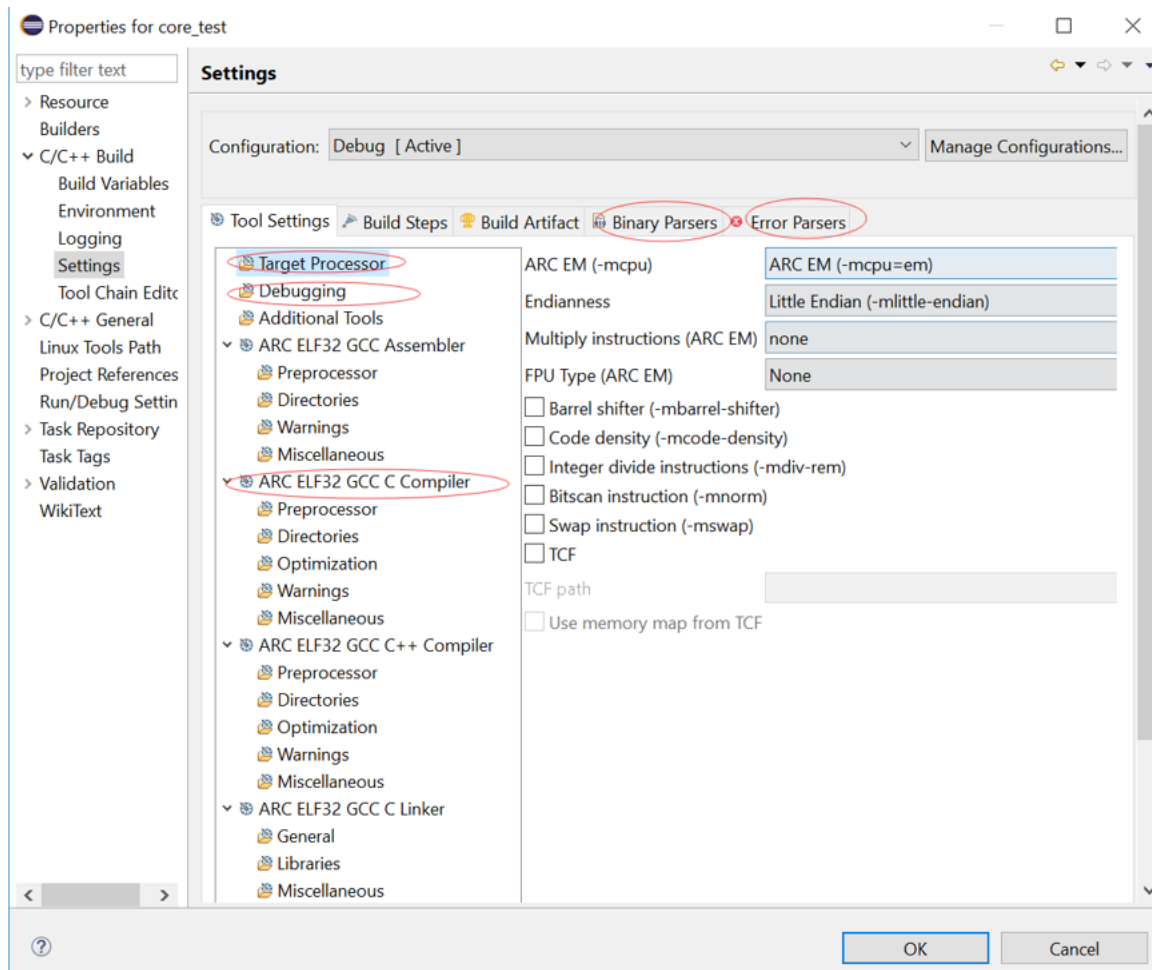
In the Import wizard, select **File system** from the **General** tab, then click **Next**. As shown in the following figure, in the From directory field, type or browse to select the directory contain the file `CoreTest.c`. Recent directories that have been imported from are shown on the From directory field's combo box. In the left pane, check a folder that imports the contents into the Workbench, and in the right pane check the file `CoreTest.c`.



Click **Finish** when done, the file `CoreTest.c` is now shown in the one of the navigation views in the project `core_test`.

3. Set compilation options

From the Project Explorer view, right-click the project `core_test` and choose Properties. Click **C/C++ Build > Settings > Tool Settings**. The **Tool Settings** dialog opens.



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, and so on whether to include Timer0/1.

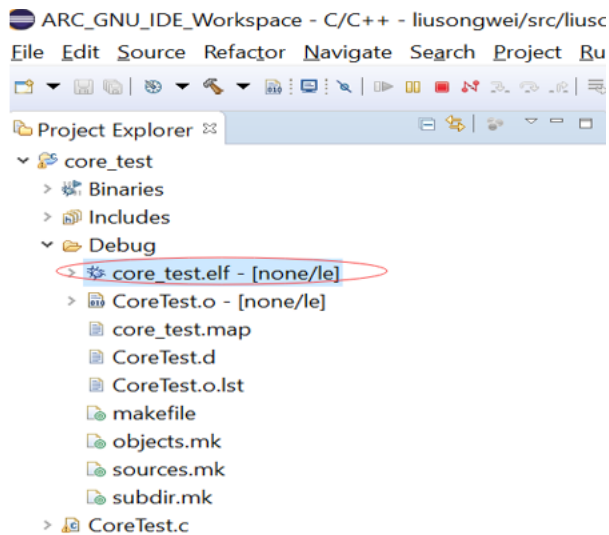
In step 1, you built the project using the engineering template of EMSK, the corresponding necessary options have been set by default. If there is no special requirement, check the setting compile options in the All options column and click **OK** when done.

4. Build project

In the Project Explorer view, select project `core_test`.

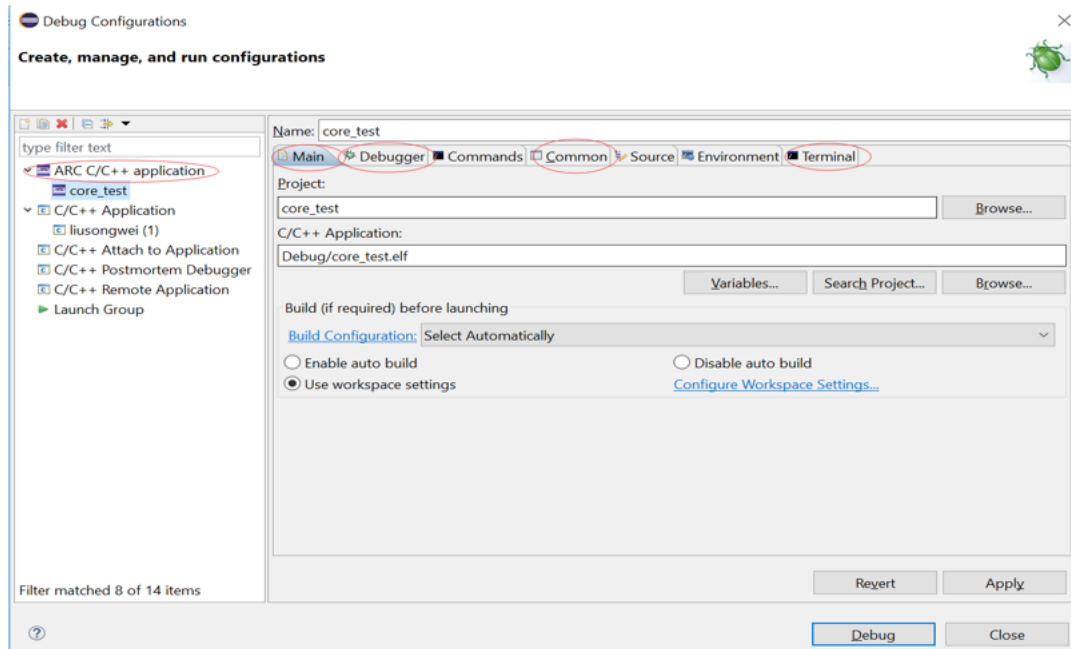


Click **Project > Build Project** or click the hammer icon. In the middle of the GNU Toolchain for ARC Processors main interface, you can see in the **Console** view the output and results of the build command. Click the tab to bring the view forward if it is not currently visible. If for some reason it is not present, you can open it by selecting **Window > Show View > Console**. When the message `Finished building target: Core_test.elf` is displayed, the compilation is successful, and the compiled executable file `Core_test.elf` can be seen in the Project Explorer.



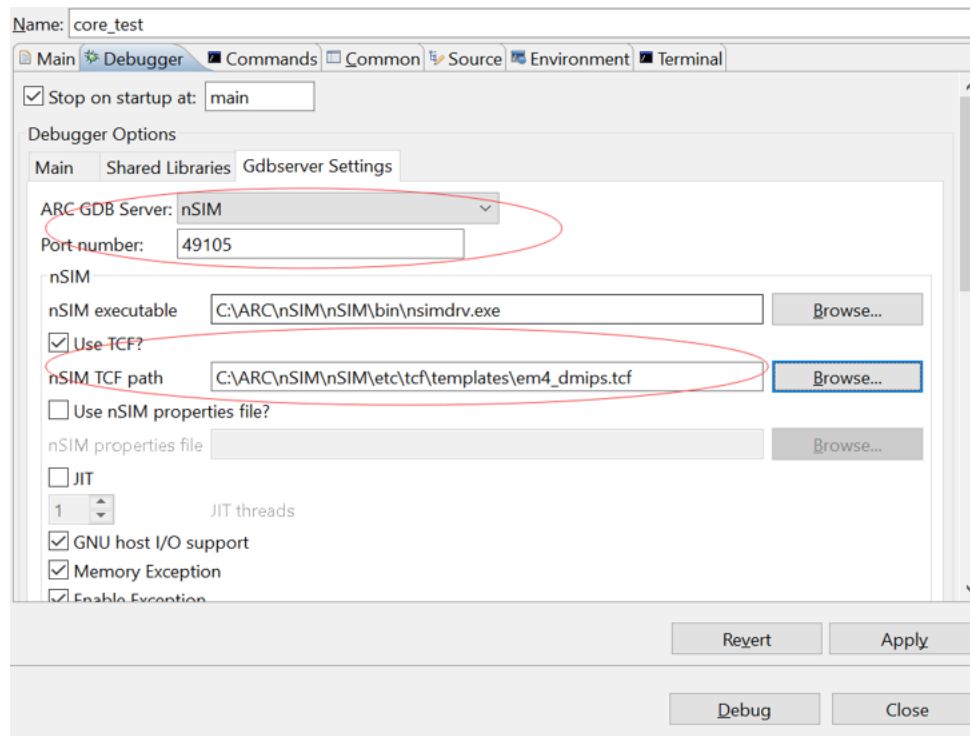
5. Set debugger options

Click the **Run > Debug Configurations...** menu option to open the **Debug Configurations** dialog. Double-click **C/C++ Application** or right-click **New** to create a new launch configuration.



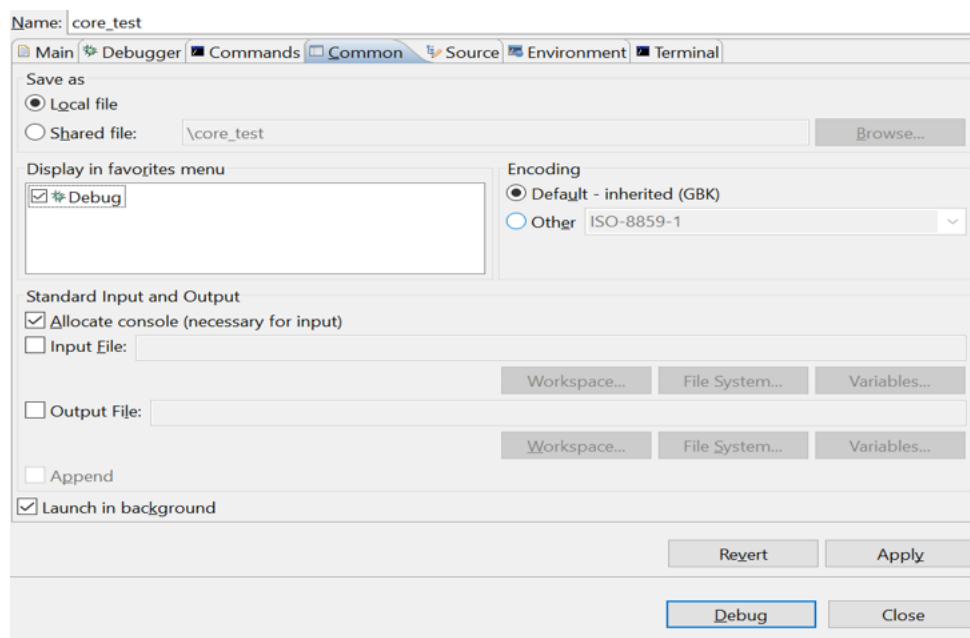
If a project is selected in the Project Explorer view all data is automatically entered, take a moment to verify its accuracy or change as needed. As you use nSIM simulator to simulate EMSK development board, you need to modify the settings of Debugger, Common, and Terminal (this is because nSIM cannot be called directly in GNU IDE. Still needs GDB Server for indirect calls). The specific settings are as follows:

- Set Debugger->Gdbserver Settings



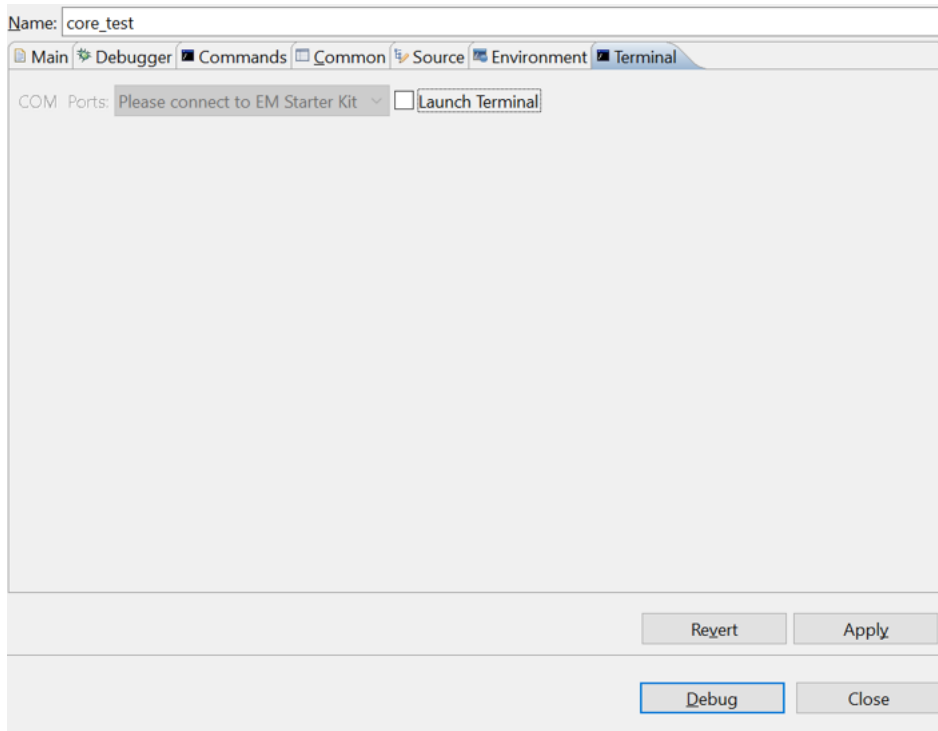
Select nSIM as the **ARC GDB Server**, and the default **port number** is 49105. Note that **Use TCF** is enabled. Otherwise, the nSIM cannot work normally. The TCF start file is under *nSIM/nSIM/etc/tcf/templates* (the default installation path). If you have downloaded the MetaWare IDE, the default nSIM path is *C:\ARC\nSIM\nSIM\etc\tcf\templates*, and you can select a TCF file from this folder (depending on the version of the board you are simulating and the kernel model), as shown earlier.

- Pay attention to Debug in Common



- Terminal settings

If you are using the EM Starter Kit, the terminal automatically selects the correct port number, and if you are using the emulator without a port, uncheck it as shown in the following figure.

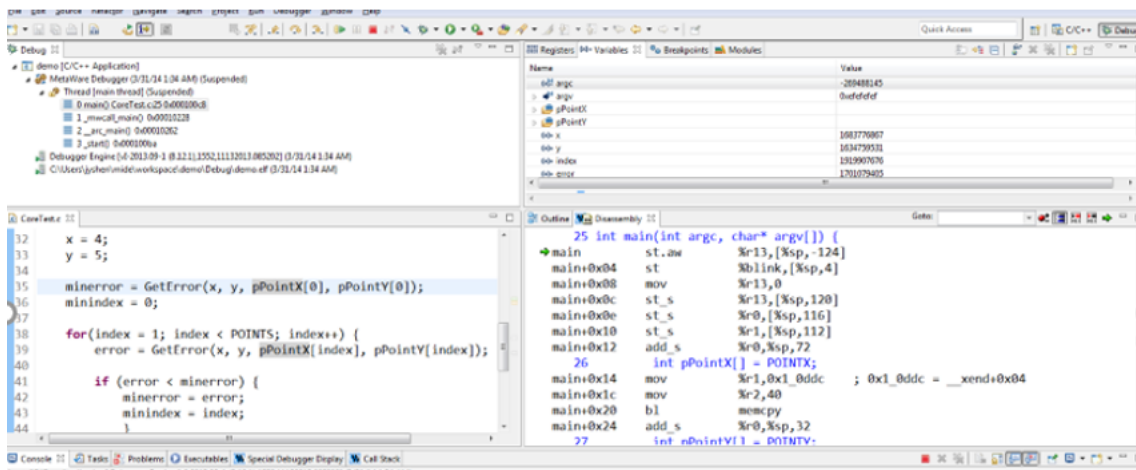


When you are done, click **Debug** to enter the debugging interface.







6. Debug executable file core_test.elf



You may be prompted to switch to the **Debug** perspective. Click **Yes**.

The Debug perspective appears with the source code window, assembly code window, register window, global variable window, breakpoint window, function window, and so on.



In the C code window, right-click the code line number on the left side of the window, select **Toggle Breakpoint** or double-click the line number to set a breakpoint on. In the assembly code window, double-click a line of code to set a breakpoint on. A blue circle is seen indicating the breakpoint is set.

After the breakpoint is set, click **Run > Resume** or you can use the **Resume** button  on the toolbar of the Debug view to run the program. The program runs directly to the nearest breakpoint. You can observe the current program execution and the relevant status information of the processor through the various windows as described in 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 . If you want to return to the main GNU Toolchain for ARC Processors page, click

C/C++  .

7. Code performance analysis using the debugger

Same as the code performance analysis method of MetaWare Development Toolkit.

For the use of these two IDEs, you can refer to the Help documentation in the respective IDE, or you can view the on-line documentation provided by Synopsys.

3.1.2 How to use embARC OSP

Purpose

- To know the concept of embARC OSP
- To know how to run examples in embARC OSP
- To know how to debug the examples in embARC OSP
- To know how to create application in embARC OSP

Requirements

The following hardware and tools are required:

- PC host
- GNU Toolchain for ARC Processors / MetaWare Development Toolkit
- ARC board (EM Starter Kit / IoT Development Kit)
- embARC OSP packages

For the detailed tool requirements of embARC OSP, see [Software Requirement](#).

Content

- A brief introduction of embARC OSP
- Get embARC OSP and run and debug the provided examples
- Create an embARC OSP application

Principles

1. IoT OS/Platform

As more and more devices are connected and become more complex, the tools 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 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, see embARC OSP [online 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](#). The repository consists of scripts and other things to you need to setup your development environment, and use Git to get this repo. If you do not have Git installed, see the beginning of the OS-specific instructions for help.

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

- http download

You may also try to get the latest release of embARC OSP 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 Windows or a **terminal** in Linux and change directory to the root of embARC OSP.

Use 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, 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 EM Starter Kit, 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 on the EM Starter Kit) when bit 3 and bit 4 of SW1 switch is off to run a self-test. By doing so, board information is sent by UART and displayed 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 does not have usable LEDs except some status LEDs, the following output log is displayed through UART.

[illegible]

Debug the examples

Use 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=arcem1ld 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 Development Toolkit, the mdb (MetaWare debugger) is used and it is a GUI interface. You can refer the MetaWare toolchain user manual for details.

For GNU Toolchain for ARC Processors, the command-line based `gdb` is used. You need to have some basic knowledge of `gdb` debug.

Create your own application

Create your own application in embARC OSP.

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

1. Create 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 command-line. If not specified, the default value is `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 command-line. If not specified, the default value is 22 for board `emsk`.
- Change the core configuration: change the value of **`CUR_CORE`** to **`arcem7d`** This option can also be given in command-line. If not specified, the default is `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 nshell 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.
 - 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 EM Starter Kit 2.2 hardware configuration to ARC EM7D (no need to set to IoT Development Kit), and connect it to your PC. Open PuTTY or Tera-term, and connect to the right COM port. Set the baud rate to **115200 bps**.
- Enter `make run` in the command-line to run this application.

Exercises

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

3.1.3 How to use ARC board

Purpose

- To get familiar with the usage of ARC board and on-board peripherals
- To know how to program and debug the ARC board and on-board peripherals

Requirements

The following hardware and tools are required:

- PC host
- GNU Toolchain for ARC Processors / MetaWare Development Toolkit
- ARC board (EM Starter Kit / IoT Development Kit)
- `embarc_osp/arc_labs/labs/lab5_emsk/embarc_osp/arc_labs/labs/lab5_iotdk`

Content

- A brief introduction of ARC board and on-board peripherals
- Based on embARC OSP, program the GPIO to control some on-board peripherals

Note: About the detailed usage of embARC OSP, see [How to use embARC OSP](#)

Principles

EM Starter Kit

About the brief introduction of EM Starter Kit, see to [embARC OSP Documentation](#)

There are LEDs, DIP switches, and buttons on EM Starter Kit, this lab shows how to program the GPIO to control these on-board peripherals of EM Starter Kit. The code for this lab is located in `embarc_osp/arc_labs/labs/lab5_emsk`. In the code, the on-board buttons and DIP switches' values are read, and whether LEDs are on or off depend on the value of the buttons and DIP switches.

IoT Development Kit

About the brief introduction of IoT Development Kit, see [embARC OSP Documentation](#)

As there are no LED or other easy-to-use peripherals on IoT Development Kit, this lab shows how to control a LED through the arduino interface of IoT Development Kit. The code for this lab is located in `embarc_osp/arc_labs/labs/lab5_iotdk`. In the code, the external connected LED blinks periodically.

Steps

EM Starter Kit

1. Connect EM Starter Kit to your computer, select **em7d** configuration and open UART terminal.
2. Compile and run the `embarc_osp/arc_labs/lab5_emsk` example with the following commands:

```
cd /arc_labs/lab5_emsk
make BOARD=emsk BD_VER=22 CUR_CORE=arcem7d TOOLCHAIN=gnu run
```

3. Press the buttons (L or/and R) and toggle the DIP switches (3 or/and 4), then check the result in UART terminal, and watch the changes of on-board LEDs.



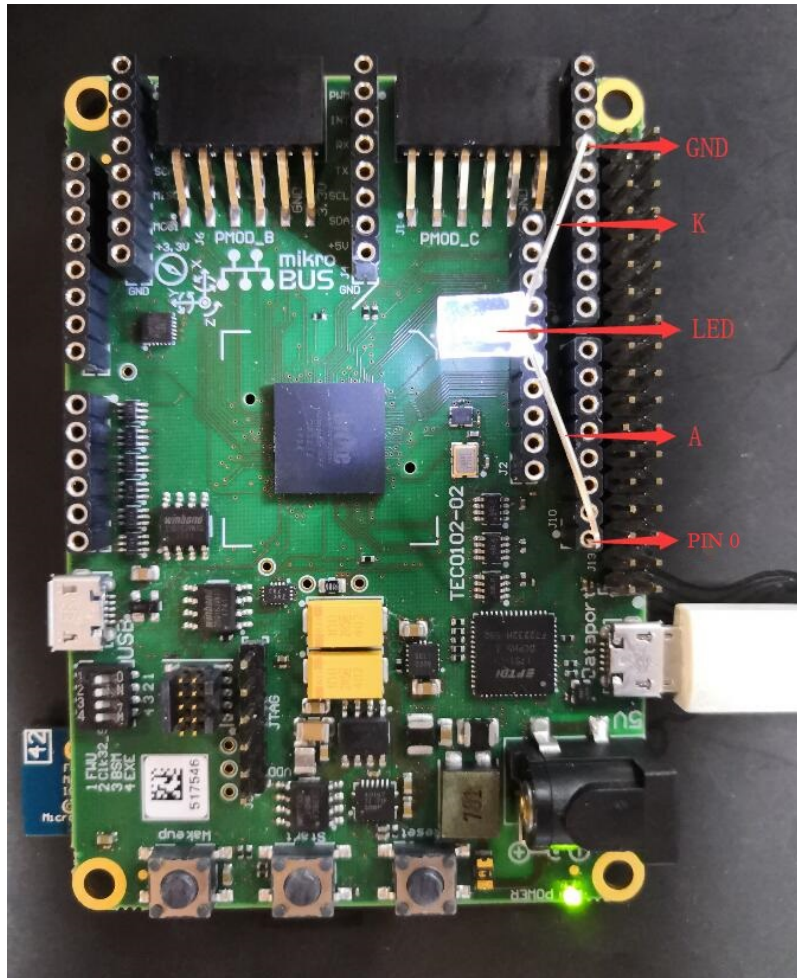
IoT Development Kit

IoT Development Kit has an arduino interface, here select arduino digital pinout **ARDUINO_PIN_0(iotdk_gpio4b_2[0])** to control LED.

1. Find a LED, connect the LED anode pin to **ARDUINO_PIN_0**, connect the LED cathode pin to **GND** of IoT Development Kit.
2. Connect the USB cable to the USB data port of IoT Development Kit and the computer.
3. Compile and run the `embarc_osp/arc_labs/lab5_iotdk` example with the following commands:

```
cd /arc_labs/lab5_iotdk
make BOARD=iotdk TOOLCHAIN=gnu run
```

4. Watch the changes of external connected LED.



Note: The connection between LED and IoT Development Kit is just for test. A $1k\Omega$ resistor should be added in series connection to limited the current and prevent damage to I/O pin.

Exercises

Try to create you own application to control the peripherals of ARC board

Note: The ARC IoT Development Kit is powered over USB. Note that the ARC IoT Development Kit needs to be powered by an external power adapter if additional devices are connected to the extension interfaces. External power supply must be 5V DC (A 12V power supply will most probably damage your board).

3.1.4 ARC features: AUX registers and timers

Purpose

- To know the auxiliary registers and processor timers of DesignWare® ARC® processors
- To learn how to program auxiliary registers to control the processor timers

Requirements

The following hardware and tools are required:

- PC host
- GNU Toolchain for ARC Processors / MetaWare Development Toolkit
- ARC board (EM Starter Kit / IoT Development Kit)
- `embarc_osp/arc_labs/labs/lab_timer`

Content

- Through reading the corresponding Build Configuration Register (BCR) auxiliary registers of processor timers to get the configuration information
- Through programming the auxiliary registers to initialize, start and stop the timer (here `TIMER0` is used)
- By reading the count value of processor timers, get the execution time of a code block

Principles

Auxiliary Registers

The auxiliary register set contains status and control registers, which by default are 32 bits wide to implement the processor control, for example, interrupt and exception management and processor timers. These auxiliary registers occupy a separate 32-bit address space from the normal memory-access (that is load and store) instructions. Auxiliary registers accessed using distinct Load Register (LR), Store Register (SR), and Auxiliary EXchange (AEX) instructions.

The auxiliary register address region `0x60` up to `0x7F` and region `0xC0` up to `0xFF` is reserved for the Build Configuration Registers (BCRs) that can be used by embedded software or host debug software to detect the configuration of the ARCV2-based hardware. The Build Configuration Registers contain the version of each ARCV2-based extension and also the build-specific configuration information.

In embARC OSP, `arc_builtin.h` provides API (`arc_aux_read` and `arc_aux_read`) to access the auxiliary registers.

Processor Timers

The processor timers are two independent 32-bit timers and a 64-bit real-time counter (RTC). **Timer0** and **Timer1** are identical in operation. The only difference is that these timers are connected to different interrupts. The timers cannot be included in a configuration without interrupts. Each timer is optional and when present, it is connected to a fixed interrupt; interrupt 16 for timer 0 and interrupt 17 for timer 1.

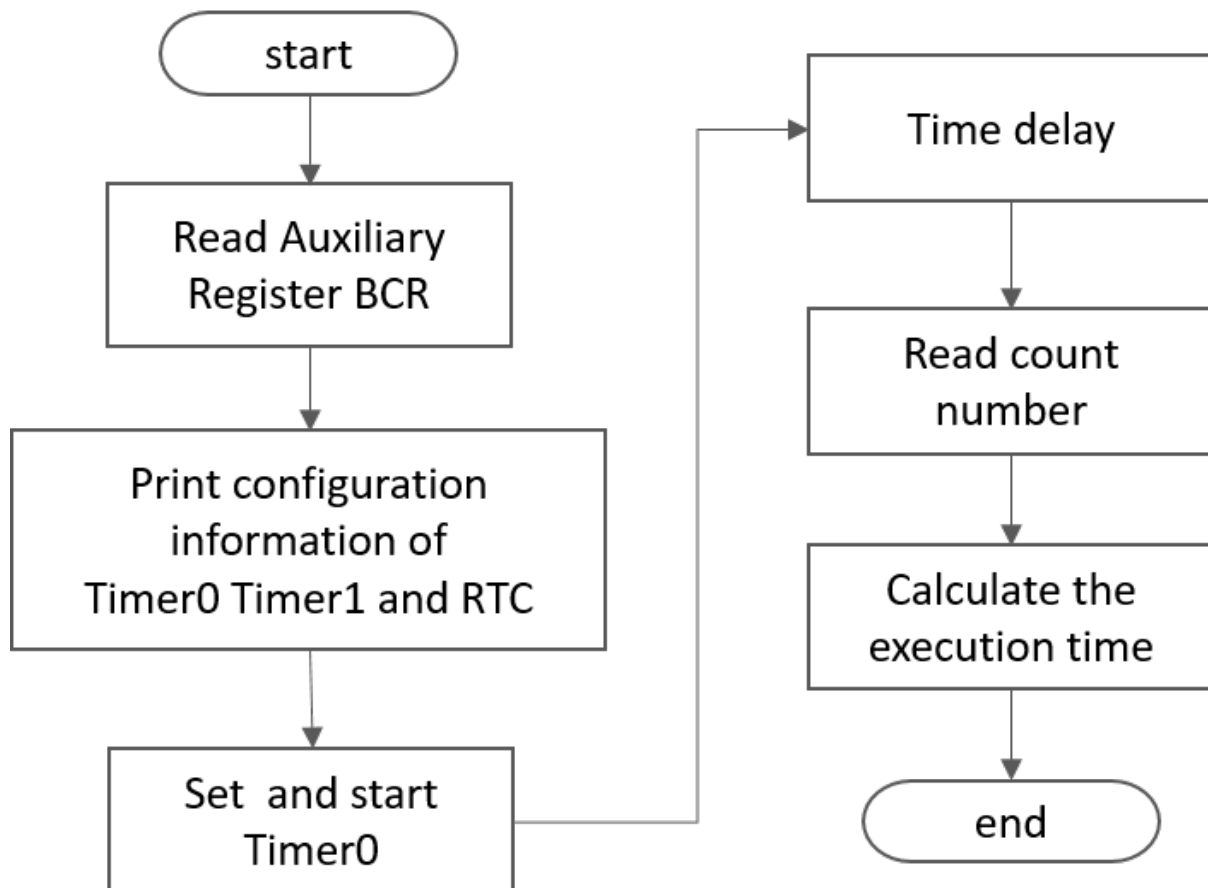
The processor timers are connected to a system clock signal that operates even when the ARCV2-based processor is in the sleep state. The timers can be used to generate interrupt signals that wake the processor from the SLEEP state. The processor timers automatically reset and restart their operation after reaching the limit value. The processor timers can be programmed to count only the clock cycles when the processor is not halted. The processor timers can also be programmed to generate an interrupt or to generate a system Reset upon reaching the limit value. The 64-bit RTC does not generate any interrupts. This timer is used to count the clock cycles atomically.

Through the BCR register `0x75`, you can get the configuration information of processor timers

In embARC OSP, `arc/arc_timer.h` provides API to operate the processor timers.

Program flow chart

The code's flow is shown below:



The code can be divided into 3 parts:

- Part1 : read the BCR of internal timers to check the features
- Part2 : promgram Timer0 by auxiliary registers with the embARC OSP provided API
- Part3 : read the counts to Timer 0 to measure a code block's execution time

Steps

1. Build and Run

```
$ cd <embarc_root>/arc_labs/labs/lab_timer
# for emsk
$ make BOARD=emsk BD_VER=22 CUR_CORE=arcem7d TOOLCHAIN=gnu run
# for iotdk
$ make BOARD=iotdk TOOLCHAIN=gnu run
```

2. Output

[illegible]

(continues on next page)

(continued from previous page)

```

      \__|_| |_| |_|_|_|_| \__\_| \__\__|
-----

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

```

Exercises

1. Try to program TIMER1
2. Try to create a clock with a tick of 1 second

3.1.5 ARC features: Interrupts

Purpose

- To introduce the interrupt handling of DesignWare® ARC® processors
- To know how to use the interrupt and timer APIs already defined in embARC OSP

Requirements

The following hardware and tools are required:

- PC host
- GNU Toolchain for ARC Processors / MetaWare Development Toolkit
- ARC board (EM Starter Kit / IoT Development Kit)
- embarc_osp/arc_labs/labs/lab_interrupt

Content

- Through `embarc_osp/arc_labs/labs/lab_interrupt/part1` to learn the basics of interrupt handling of DesignWare® ARC® processors and the interrupt API provided by embARC OSP
- Through `embarc_osp/arc_labs/labs/lab_interrupt/part2` to learn the interrupt priority and interrupt nesting of DesignWare® ARC® processors and corresponding API of embARC OSP

Principles

1. Interrupt

An interrupt is a mechanism in processor to respond to special interrupt signals emitted by hardware or software. Interrupts can be used by processor to perform a specific function after some specific event happens and then return to normal operation. For this purpose there are many different types of interrupts possible to be issued by hardware and software and each interrupt can have it's own functions called Interrupt Service Routine (ISR). ISR is a function (sequence of commands) to deal with the immediate event generated by a given interrupt.

2. Interrupt unit of DesignWare® ARC® processors

The interrupt unit of DesignWare® ARC® processors has 16 allocated exceptions associated with vectors 0 to 15 and 240 interrupts associated with vectors 16 to 255. The ARCv2 interrupt unit is highly programmable and supports the following interrupt types:

- Timer — triggered by one of the optional extension timers and watchdog timer
- Multi-core interrupts — triggered by one of the cores in a multi-core system
- External — available as input pins to the core
- Software-only — triggered by software only

The interrupt unit of DesignWare® ARC® processors has the following interrupt specifications:

- **Support for up to 240 interrupts**
 - User configurable from 0 to 240
 - Level sensitive or pulse sensitive
- **Support for up to 16 interrupt priority levels**
 - Programmable from 0 (highest priority) to 15 (lowest priority)
- The priority of each interrupt can be programmed individually by software
- **Interrupt handlers can be preempted by higher-priority interrupts**
 - Optionally, highest priority level 0 interrupts can be configured as “Fast Interrupts”
 - Optional second core register bank for use with Fast Interrupts option to minimize interrupt service latency by minimizing the time needed for context saving
- Automatic save and restore of selected registers on interrupt entry and exit for fast context switch
- User context saved to user or kernel stack, under program control
- Software can set a priority level threshold in STATUS32.E that must be met for an interrupt request to interrupt or wake the processor
- **Minimal interrupt / wake-up logic clocked in sleep state**
 - Interrupt prioritization logic is purely combinational
- Any Interrupt can be triggered by software

The interrupt unit can be programmed by auxiliary registers. For more details, See DesignWare® ARC® processors ISA.

3. Interrupt API in embARC OSP

In embARC OSP, a basic exception and interrupt processing framework is implemented in embARC OSP. Through this framework, you can handle specific exceptions or interrupts by installing the desired handlers. This can help you analyze the underlying details of saving and operating registers. See [here](#) for details.

The interrupt and exception related API are defined in `arc_exception.h`.

Steps

Part I: implement a customized timer0 interrupt handling

1. Build and Run

```
$ cd <embarc_root>/arc_labs/labs/lab4_interrupt/part1
# for emsk
$ make BOARD=emsk BD_VER=22 CUR_CORE=arcem7d TOOLCHAIN=gnu run
# for iotdk
$ make BOARD=iotdk TOOLCHAIN=gnu run
```

2. Output

```
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

...
```

3. Code analysis

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

- Interrupt service function:

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

This code is a standard example of an interrupt service routine: 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, the count variable `t0` is incremented by one.

- Main function

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

(continues on next page)

(continued from previous page)

```
arc_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);
arc_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 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, the timer and timer interrupts are configured.

This code uses the emBARC OSP API to program **Timer0**. 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, turn them off first. This work is done by the functions `int_disable` and `arc_timer_stop`.

Then 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, enable the **Timer0** and interrupts that are previously turned off. This work is done by the functions `int_enable` and `arc_timer_start`. The implementation of the `arc_timer_start` function is the same as the reading and writing of the auxiliary registers in `lab_timer`. You 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 `arc_timer_start`). Configure the interrupt time to 1ms, do a simple calculation (the formula is the expression after `COUNT`).

In this example, the loop body only serves as an effect display. Delay function in the loop body to print the time per second is called.

Note: Since nSIM is only simulated by computer, there may be time inaccuracy when using this function. You can use the EMSK to program the program in the development board. In this case, the time is 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 the function entered, clear the global variable `t0`. The interrupt interval is set to 1ms in the above `arc_timer_start`, assume that every time `t0` is incremented, the time has passed 1ms.

Wait through the `while(t0<ms)` sentence, so that the full ms delay with higher precision is received.

Part II: interrupt priority and interrupt nesting

1. Build and Run

```
$ cd <embarc_root>/arc_labs/labs/lab4_interrupt/part2
# for emsk
$ make BOARD=emsk BD_VER=22 CUR_CORE=arcem7d TOOLCHAIN=gnu run
# for iotdk
$ make BOARD=iotdk TOOLCHAIN=gnu run
```

2. Output

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

3. Code analysis

The code for PART II can be divided into two parts: the interrupt service routine and the main function.

- Interrupt service function

```
static void timer0_isr(void *ptr)
{
    arc_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");
    }
}
```

(continues on next page)

(continued from previous page)

```

    hits++;
}

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

    timer_flag = 1;
}

```

Through the above code, when timer0's interrupt comes in and is serviced, different output messages are sent by ISR according to the value of *timer_flag*, which is only be set in timer1's ISR *timer1_isr*. This means timer0's interrupt is preempted by timer1's interrupt as it has a higher interrupt priority.

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

- main function

```

int main(void)
{
    arc_timer_stop(TIMER_0);
    arc_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");

    int_enable(INTNO_TIMER0);
    int_enable(INTNO_TIMER1);

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

    while(1)
    {
        if((hits >= 5) && (nesting_flag == 1)) {
            arc_timer_stop(TIMER_0);
            arc_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);

            arc_timer_start(TIMER_0, TIMER_CTRL_IE | TIMER_CTRL_NH, ↵
↵MAX_COUNT);
            arc_timer_start(TIMER_1, TIMER_CTRL_IE | TIMER_CTRL_NH, ↵
↵MAX_COUNT/10);
        }
    }
}

```

(continues on next page)

(continued from previous page)

```

    } else if((hits >= 10) && (nesting_flag == 0)) {
        arc_timer_stop(TIMER_0);
        arc_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);

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

```

First, the timer 0 and timer 1 are configured and install with corresponding ISR. Then in the while loop, the interrupt priority of timer 0 and timer 1 are periodically changed to make the interrupt nesting happen.

Exercises

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

3.1.6 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

Requirements

The following hardware and tools are required:

- PC host
- GNU Toolchain for ARC Processors / MetaWare Development Toolkit
- ARC board (EM Starter Kit / IoT Development Kit)
- SD card
- `example/baremetal/bootloader`

Simple Bootloader

This simple bootloader is designed to work as a secondary/simple bootloader for embARC OSP, it loads `boot.hex` or `boot.bin` on SD Card and run that program. The example 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

Content

1. Build and run the `example/baremetal/bootloader`
2. Download the generated `bootloader.bin` into flash
3. Build a self-boot application 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 (starting from 0x17f00000) 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 in the following table:

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

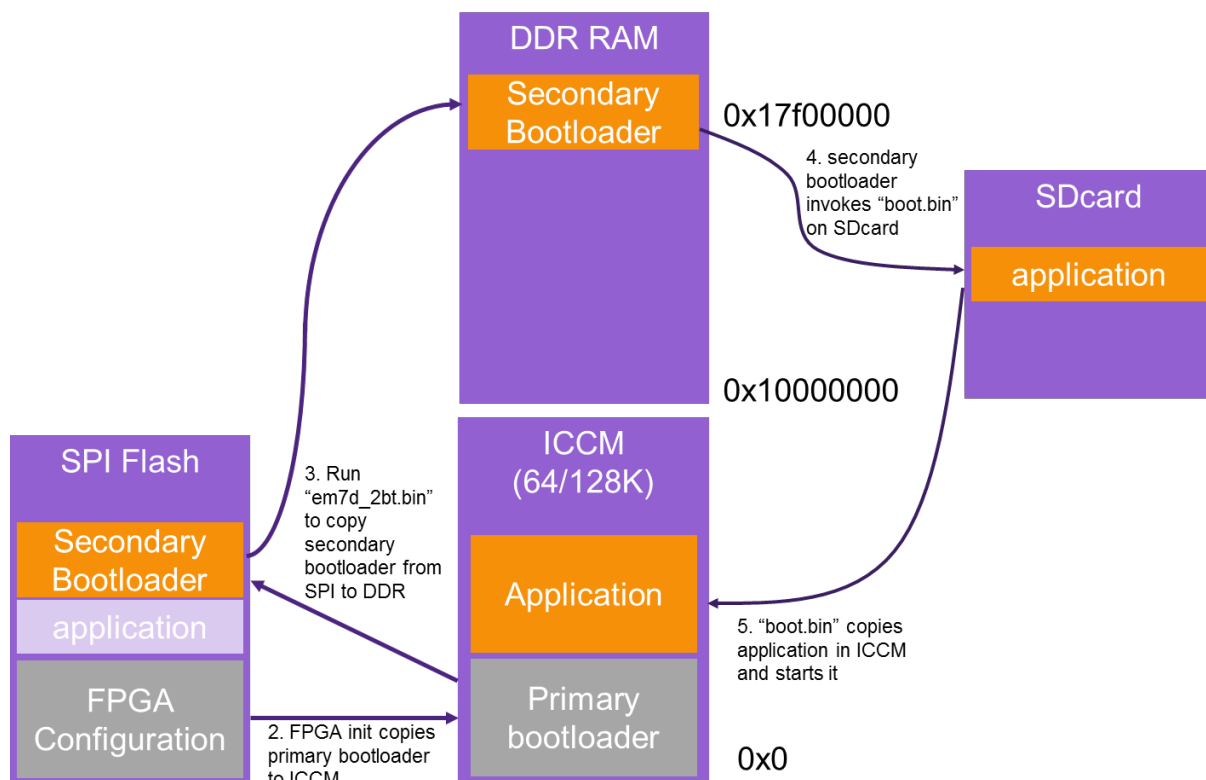
EM Starter Kit

The EM Starter Kit 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 auto-loads 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 (DDR).
4. Run secondary bootloader from main memory to load application from the SD card into ICCM/DDR memory.
5. Run the application from ICCM/DDR memory.



IoT Development Kit

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

simple bootloader can be written to both flash to load an application from the TF card. The startup sequence for IoT Development Kit is listed below:

1. Power on or reset event
2. Boot from on-chip eflash or extern boot SPI flash decided by the FWU switch
3. Run simple bootloader to load application from the TF card into ICCM
4. Run the application from ICCM memory

How to flash the ARC board

Note: In this lab, we do not use MCUBoot, so we need to disable MCUBoot, we should set **USE_MCUBOOT = 0** in makefile.

EM Starter Kit

- Generate a secondary bootloader binary file

```
$ cd <embarc_root>/example/baremetal/bootloader
$ make BOARD=emsk BD_VER=22 CUR_CORE=arcem7d TOOLCHAIN=gnu bin
```

- **Program the secondary bootloader binary file into SPI Flash**

- Insert SD card to your PC, and copy the binary file `obj_emsk_22/gnu_arcem7d/emsk_bootloader_gnu_arcem7d.bin` to SD card root folder, and rename it to `em7d_2bt.bin`
- Insert the SD card to EMSK Board, choose the right core configuration, build and run the `<embARC>/example/baremetal/bootloader` example, then press any button to stop auto boot process, and enter to ntshell command mode
- **Use ntshell command *spirw* to program the `em7d_2bt.bin` into spiflash**
 - * Run *spirw* to show help
 - * Run *spirw -i* to check SPI Flash ID, it should be **Device ID = ef4018**
 - * Run *spirw -w em7d_2bt.bin 0x17f00000 0x17f00004* to program spiflash
 - * Check the output message to see if it has been programmed successfully

```

COM3:115200baud - Tera Term VT
File Edit Setup Control Window Help
ntshell command load(load .bin file from SD card to ram at specified address) was registered!
ntshell command go(run the program at the specified address) was registered!
NTShell Task StartUp
COM1>spirw -h
usage: spirw <-opt>
--Option Available--
-h/-?          show help
-i read Device ID
-r filename     read image from spi to file on host, address and size have been taken
               from header in spi flash
-w file ram_addr start_addr write binary [file] to spi flash, the image will be loaded to ram_addr
               r and run from start_addr
*****
SPI Flash Command Done
COM1>spirw
usage: spirw <-opt>
--Option Available--
-h/-?          show help
-i read Device ID
-r filename     read image from spi to file on host, address and size have been taken
               from header in spi flash
-w file ram_addr start_addr write binary [file] to spi flash, the image will be loaded to ram_addr
               r and run from start_addr
COM1>spirw -i
Device ID = ef4018
*****
SPI Flash Command Done
COM1>spirw -w em7d_2bt.bin 0x17f00000 0x17f00004
** Writing file em7d_2bt.bin to address 0x17f00000 start at 0x17f00004
   Erased 37 sectors from 0xd80000 to 0xda4fff

** Writing SPI flash **
- data
write 0xd8001c - 0xd8101c check_sum:57126
write 0xd8101c - 0xd8201c check_sum:ac3a2
write 0xd8201c - 0xd8301c check_sum:102aac
write 0xd8301c - 0xd8401c check_sum:1575b3
write 0xd8401c - 0xd8501c check_sum:1a91ea
write 0xda201c - 0xda301c check_sum:efb304
write 0xda301c - 0xda401c check_sum:f43f00
write 0xda401c - 0xda470c check_sum:f5015c
   Written 149260 bytes: header=28 and image=149232 image from 0xd80000 to 0xda470b

** Boot image header **
head: 0x68656164
cpu: 0x0
start: 0xd8001c
size: 0x246f0
ram addr: 0x17f00000
start: 0x17f00004
checksum: 0xf5015c
*****
SPI Flash Command Done

```

Successfully programmed to spiflash

- If programmed successfully, when the board is reset, make sure Bit 4 of the on-board DIP switch is ON to enable secondary bootloader run
- If the SD card already contains the *boot.bin* in it, the bootloader automatically loads it from SD card. If not, it enters to ntshell mode
- You can goto the next step to generate the *boot.bin* for proper application you want to be auto-loaded in SD card

```

COM22:115200baud - Tera Term VT
File Edit Setup Control Window Help
Firmware Feb 22 2017, v2.3
Bootloader Feb 22 2017, v1.1
ARC EM7D, core configuration #1

ARC IDENTITY = 0x43
RF_BUILD = 0x2
TIMER_BUILD = 0x1010b05
ICCM_BUILD = 0xa05
DCCM_BUILD = 0x10905
I_CACHE_BUILD = 0x225105
D_CACHE_BUILD = 0x215105

SelfTest PASSED

Boot image has been found
  start = 0xd8001c
  size = 0x4030c
  ram addresss = 0x17f00000
  start address = 0x17f00004
Reload cfg button pressed(C)
*****
**      Synopsys, Inc.      **
**      ARC EM Starter kit  **
**                          **
** Comprehensive software stacks **
** available from embARC.org **
**                          **
*****
Firmware Feb 22 2017, v2.3
Bootloader Feb 22 2017, v1.1
ARC EM7D, core configuration #1

ARC IDENTITY = 0x43
RF_BUILD = 0x2
TIMER_BUILD = 0x1010b05
ICCM_BUILD = 0xa05
DCCM_BUILD = 0x10905
I_CACHE_BUILD = 0x225105
D_CACHE_BUILD = 0x215105

SelfTest PASSED

Boot image has been found
  start = 0xd8001c
  size = 0x4030c
  ram addresss = 0x17f00000
  start address = 0x17f00004
Loading 262924 bytes from SPI: 0xd8001c to RAM: 0x17f00000 - completed
Starting application from 0x17f00004
embARC Build Time: Mar 16 2017, 17:34
Compiler Version: ARC GNU, 6.2.1 20160824
FatFS initialized successfully!
boot.json open error. use default bootloader

Press any button on board to stop auto boot in 5 s

```

- Generate `boot.bin` using any embARC example, it's RAM start address should be `0x10000000`. Use bootloader to run it

- **Known Issues**

- Boot rom of EMSK1.x is not able to load secondary bootloader on SPI Flash, you need a modified EMSK1.x mcs file to enable this function, send request in forum about this mcs file.

IoT Development Kit

- Generate a secondary bootloader binary file

- **Program the secondary bootloader binary file into SPI Flash**

- Insert SD card to your PC, and copy the binary file `obj_iotdk_10/mw_arcem9d/simple_bootloader_mw_arcem9d.bin` to SD card Root, and rename it to `simple_bootloader.bin`
- copy the file `example/bootloader/boot.json` to SD card root, and change the `boot_file` value to `boot.bin`, and change the `ram_startaddress` to `536870912(0x20000000)`

```

1  {
2      "ntshell": false,
3      "boot_file": "boot.bin",
4      "ram_startaddress": 536870912,
5      "wifi": {
6          "wifi_sel": "PMWIFI",
7          "SSID": "embARC",
8          "PSK": "qazwsxedc",
9          "DHCP": true,
10         "version": 4,
11         "address": [192, 168, 1, 123],
12         "gateway": [192, 168, 1, 1],
13         "mask": [255, 255, 255, 0],
14         "MAC": [21, 21, 21, 21, 21]
15     },
16     "app_cfg": "lwm2m.json"
17 }

```

- Insert the SD card to iotdk Board, remove `APPL_DEFINES += -DUSE_APPL_MEM_CONFIG` in `makefile`, build and run the `<embARC>/example/baremetal/bootloader` example, and enter to `ntshell` command mode.
- Use `ntshell` command `flash` to program the `simple_bootloader.bin` into both flash
 - * Run `flash -h` to show help
 - * Run `flash -eflash simple_bootloader.bin` to program eflash
 - * Run `flash -bootspi simple_bootloader.bin` to program bootspi flash
 - * Check the output message to see if it was programmed successfully

```

VT COM11 - Tera Term VT
File Edit Setup Control Window Help

COM1>flash -h
Usage: flash [OPTION]...
Write bin file to flash(eflash or bootspi flash)
  -h/H/?    Show the help information
Examples:
  flash -eflash test.bin    Write bin file to eflash
  flash -bootspi test.bin   Write bin file to bootspi flash
  flash -h                  Show the help information
COM1>flash -eflash simple_bootloader.bin
COM1>flash -bootspi simple_bootloader.bin
COM1>

```

- If the SD card already contains the `boot.bin` and `boot.json` in it, the bootloader automatically loads it from SD card, if not, it enters to `ntshell` mode
- You can goto the next step to generate the `boot.bin` for proper application you want to be auto-loaded in SD card

```

embARC Build Time: Oct 10 2018, 10:03:54
Compiler Version: Metaware, 4.2.1 Compatible Clang 4.0.1 (branches/release_40)
FatFS initialized successfully!
boot_file:boot.bin
app_cfg:lwm2m.json
ram:0x20000000
ntshell:0
Start loading boot.bin from sdcard to 0x20000000 and run...
led out: f, ff
led out: f0, ff
led-----

  PoweredBy
  embARC

embARC Build Time: Sep 28 2018, 09:18:00
Compiler Version: Metaware, 4.2.1 Compatible Clang 4.0.1 (branches/release_40)
ed out: 0, 0
led out: ffff, 0
led out: 0, 0
led out: ffff, 0
led out: 0, 0

```

- Generate `boot.bin` using any embARC example, its RAM start address should be `0x20000000`. Use bootloader to run it

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 Advanced labs

3.2.1 Memory map and linker

Purpose

- To get familiar with memory layout in compilation process
- To learn how to use linker

Requirements

The following hardware and 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**.
- Using “pragma code” to specify a new name of section in which the code of function reside.
- Mapping this code section into specified memory location with linker.
- 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 compilation process, the linker automatically maps 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 MetaWare IDE, create an empty C project called `lab_linker` and select ARC EM series processor. Import the `main.c` and `link.cmd` files from the `embarc_osp/arc_labs/labs/lab8_linker` directory into the project.

Open `main.c` file in MetaWare IDE, use “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++)
```

(continues on next page)

(continued from previous page)

```
        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, see MetaWare C/C++ Programmer's Guide for the ccac Compiler.

Edit the linker command file

Open 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. Add one new GROUP in SECTIONS, and mapping section “modify_seg” into MyBlock.

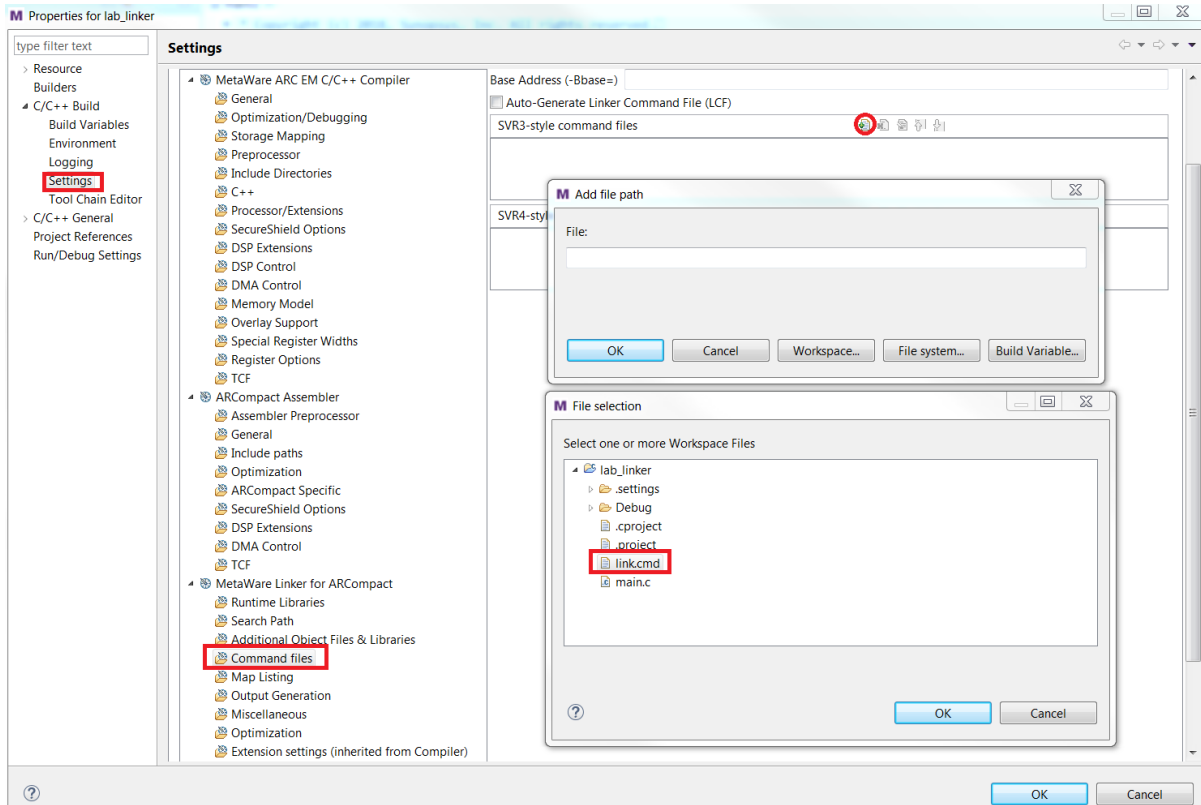
```
MEMORY {
    // Note: overlap of code and data spaces is not recommended since it makes
    //      Address validity checking impossible with the debugger and simulator
    MyBlock: ORIGIN = 0x00002000, LENGTH = 32K
    MEMORY_BLOCK1: ORIGIN = 0x0010000, LENGTH = 64K
    MEMORY_BLOCK2: ORIGIN = 0x0020000,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

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

Note: About format and syntax of the linker command file, see MetaWare ELF Linker and Utilities User's Guide.

Add the linker command file into the project

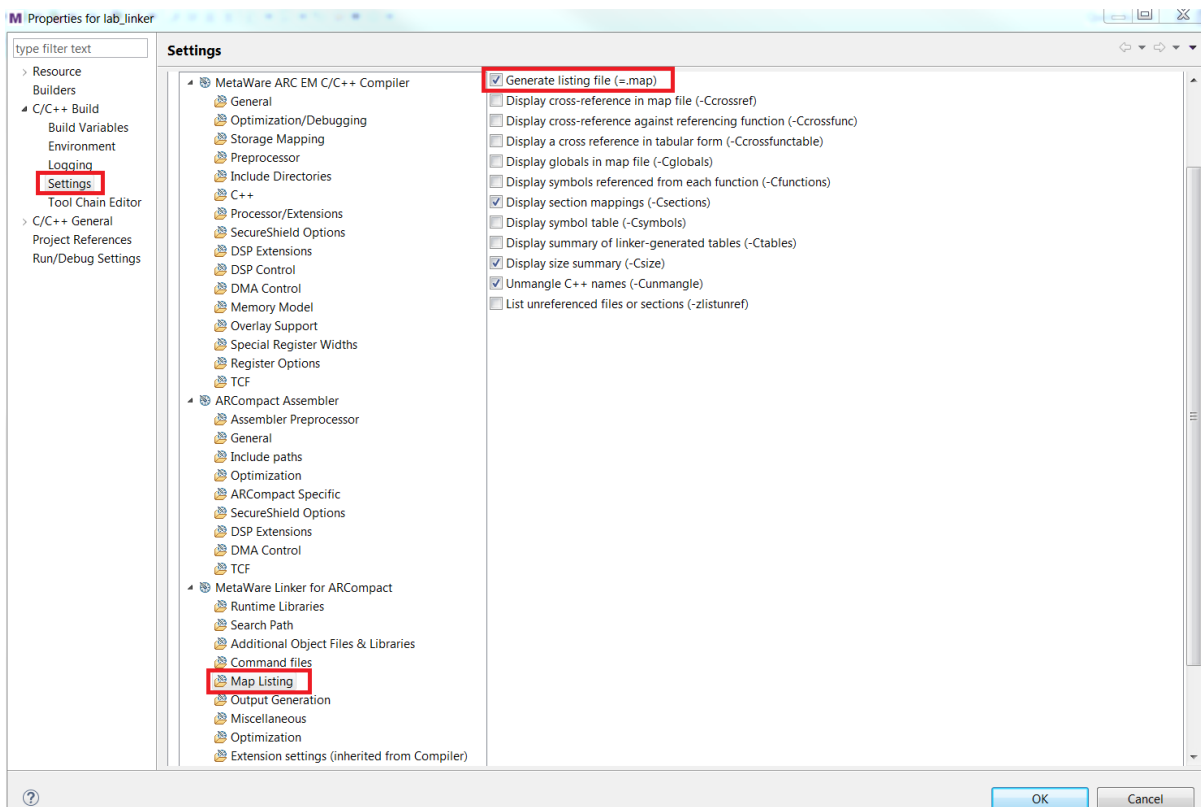
Right-click the current project lab_linker and select Properties. Click **C/C++ build > Settings > Tool Settings** to open the linker option settings page.



Select Command files to add linker.cmd file into this project.

Check the result

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



Build the lab_linker project, then open the lab_linker.map file.

SECTION SUMMARY

OUTPUT/ INPUT	TYPE SECTION	START ADDRESS	END 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

Search SECTIONS SUMMARY, then you can check the size and location of *modify_seg* section, it resides in *MyBlock*, similar to you setting in the linker command file.

Exercises

Check the memory mapping info of *modify_seg* section by using *elfdump* tool.

3.2.2 A WiFi temperature monitor

Purpose

- To learn how to build a wireless sensor terminal based on the embARC OSP package
- To know how to use ESP8266 module and AT commands
- To learn more about the usage of FreeRTOS operating system

Requirements

The following hardware and tools are required:

- PC host
- GNU Toolchain for ARC Processors / MetaWare Development Toolkit
- ARC board (IoT Development Kit)
- embARC OSP package
- `embarc_osp/arc_labs/labs/lab_esp8266_wifi`

Content

Through this lab, you get a preliminary understanding of ESP8266 WiFi module and the AT command.

The lab is based on the embARC OSP package and the supports of the popular WiFi module, ESP8266. During the lab, you first use the AT command to set the ESP8266 to the server mode. Then you can use your laptop or mobile phone to access ESP8266 by IP address. You get a static webpage transmitted via TCP protocol.

Principles

ESP8266

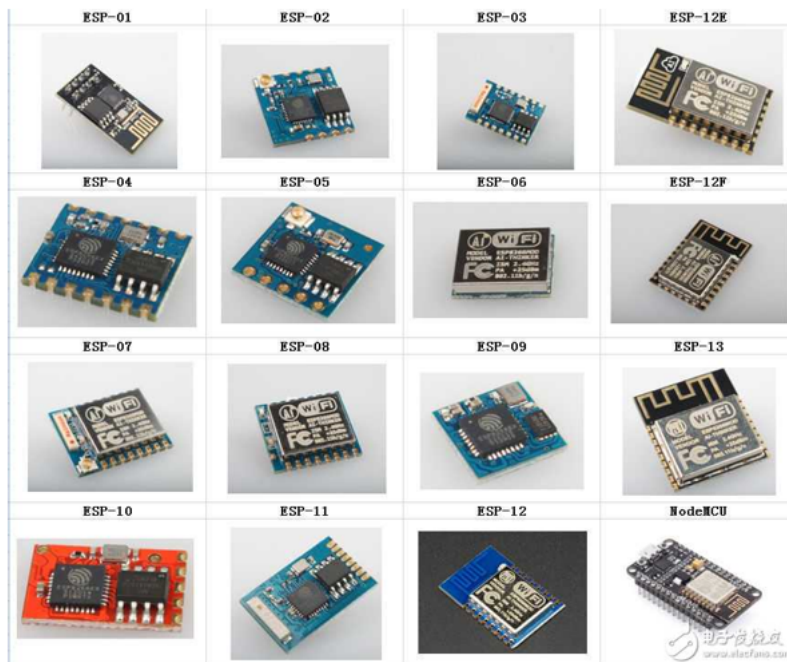
The ESP8266 is an ultra-low-power WiFi chip with industry-leading package size and ultra-low power technology. It is designed for mobile devices and IoT applications, facilitating the connection between user devices to IoT environments.

The ESP8266 is available with various encapsulations. On-board PCB antenna, IPEX interface, and stamp hole interface are supported.

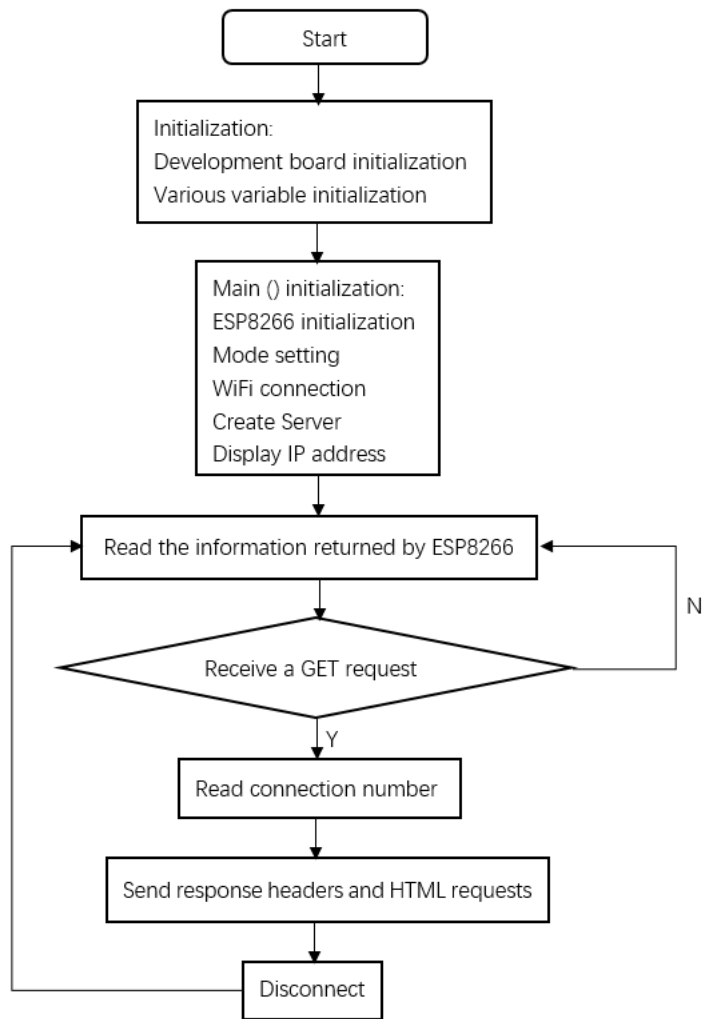
ESP8266 can be widely used in smart grid, intelligent transportation, smart furniture, handheld devices, industrial control, and other IoT fields.

Ai-Thinker company has developed several WiFi modules based on ESP8266, including ESP01 and ESP01S which are used in this lab.

Note: See [embARC doc](#) to learn how to connect it with your board.



Program structure is shown below



Code is shown below

```

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

#include "board.h"
#include "esp8266.h"

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

#define WIFI_SSID    "\"embARC\""
#define WIFI_PWD     "\"qazwsxedc\""

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>";
  
```

(continues on next page)

(continued from previous page)

```

static char http_server_buf[1024];

int main(void)
{
    char *conn_buf;

    //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, http_server_buf);
        EMBARC_PRINTF("Searching for WIFI %s ..... \n", WIFI_SSID);
        board_delay_ms(100, 1);
    } while (strstr(http_server_buf, 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);
        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);

    while (1) {
        memset(http_server_buf, 0, sizeof(http_server_buf));
        at_read(esp8266->p_at, http_server_buf, 1000);
        board_delay_ms(200, 1);
    }
}

```

(continues on next page)

(continued from previous page)

```
//EMBARC_PRINTF("Alive\n");

if (strstr(http_server_buf, http_get) != NULL) {
    EMBARC_PRINTF("===== send_
↪=====\\n");

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

    conn_buf = strstr(http_server_buf, 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);

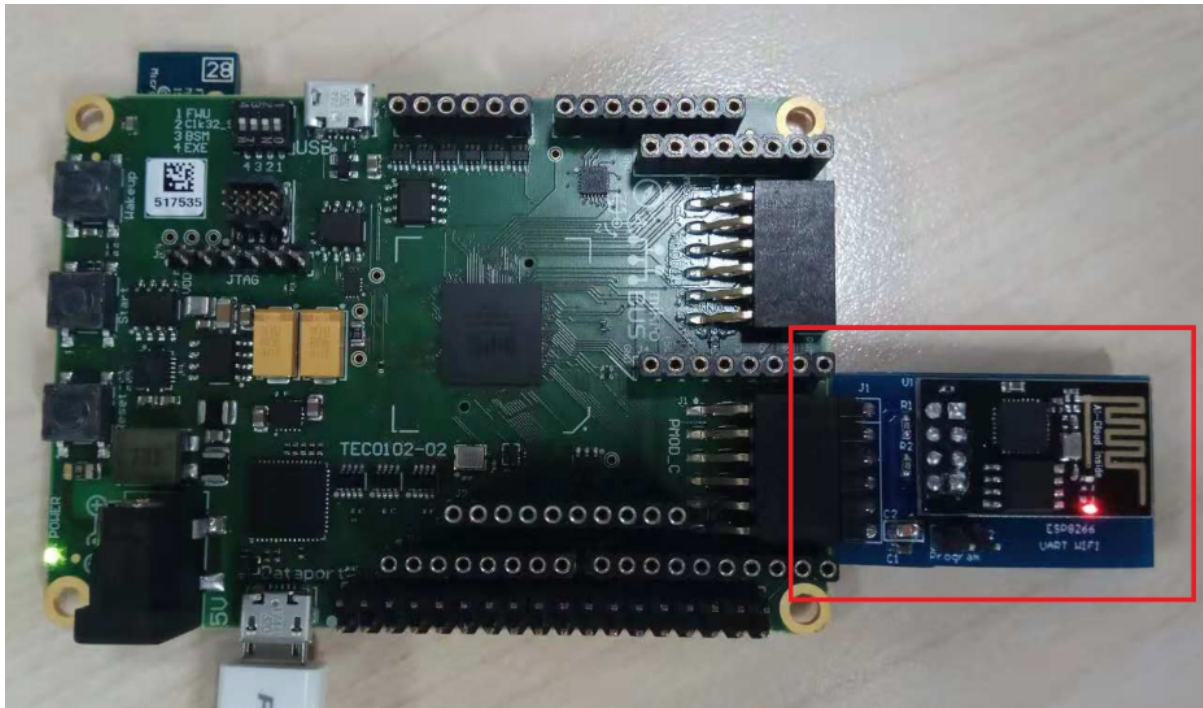
    esp8266_CIPCLOSE(esp8266, conn_buf);

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

return E_OK;
}
```

Steps

Hardware connection (as shown below)



Modify the code

Change the WiFi account and password set in the code to connect the particular wifi(as shown below).

```
#define WIFI_SSID    "\"embARC\""
#define WIFI_PWD     "\"qazwsxedc\""
```

Compile and download

Compile and download the program, after downloading successfully, the relevant download information is displayed in the command window(as shown in the following example).

```
[DIGILENT] This device supports JTAG7 scan formats.
[DIGILENT] Device enumeration: #0 is `IoTDK`=DesignWare ARC SDP.
[DIGILENT] We choose device : #0 `IoTDK' from 1 possible devices.
[DIGILENT] Product=507 variant=1 fwid=56 firmware-version=10a.
[DIGILENT] It is possible to set the JTAG speed.
[DIGILENT] Current speed is 10000000 Hz.
[DIGILENT] Attempting to set speed to 8000000 Hz.
[DIGILENT] Speed was set to 7500000 Hz.
[DIGILENT] Suppress these messages with environment variable DIG_VERBOSE=0.
Initializing. System name is ARC_DLL; my DLL was C:/ARC/MetaWare/arc/bin/freertos.
freertos: there are 10 task priorities.
```

At this point, feedback information is shown on your serial port console, representing the process of the board establishing connection with http server with AT command (showing below).

```
embARC Build Time: Nov 22 2018, 14:35:34
Compiler Version: Metaware, 4.2.1 Compatible Clang 4.0.1 (branches/release_40)
===== Init =====
[at_parser_init]57: obj->psio 0x800066c8 -> 0x80001330
[at_send_cmd]87: command is NULL, send AT test command
[at_send_cmd]131: at_out: "AT"
" (4)
[at_get_reply]154: "AT

OK" (9)
===== Set Mode =====
```

(continues on next page)

(continued from previous page)

```

[at_send_cmd]131: at_out: "AT+CWMODE_CUR?
" (16)
[at_get_reply]154: "
AT+CWMODE_CUR?
+CWMODE_CUR:1

OK" (38)
CWMODE_CUR = 1
[at_send_cmd]131: at_out: "AT+CWMODE_CUR=3
" (17)
[at_get_reply]154: "
AT+CWMODE_CUR=3

OK" (24)
===== Connect WiFi =====
[at_send_cmd]131: at_out: "AT+CWLAP
" (10)
[at_get_reply]154: "
AT+CWLAP
+CWLAP: (0,"synopsys-guest",-71,"6c:f3:7f:a8:a1:21",1,-27,0)
+CWLAP: (5,"Synopsys",-70,"6c:f3:7f:a8:a1:22",1,-27,0)
+CWLAP: (0,"synopsys-guest",-94,"d8:c7:c8:43:5b:81",1,-19,0)
+CWLAP: (5,"Synopsys",-95,"d8:c7:c8:43:5b:83",1,-21,0)
+CWLAP: (0,"iFuture",-94,"d4:68:ba:06:65:4a",1,-16,0)
+CWLAP: (4,"iFuture_City",-93,"d4:68:ba:0e:65:09",3,-4,0)
+CWLAP: (3,"embARC",-62,"5e:e0:c5:4f:df:80",6,32767,0)

OK" (416)
Searching for WIFI "embARC" .....
WIFI "embARC" found! Try to connect
[at_send_cmd]131: at_out: "AT+CWMODE_CUR=1
" (17)
[at_get_reply]154: "
AT+CWMODE_CUR=1

OK" (24)
[at_send_cmd]131: at_out: "AT+CWJAP_CUR="embARC","qazwsxedc"
" (35)
[at_get_reply]154: "
AT+CWJAP_CUR="embARC","qazwsxedc"
WIFI DISCONNECT
WIFI CONNECTED
WIFI GOT IP

OK" (88)
WIFI "embARC" connect succeed
===== Connect Server =====
[at_send_cmd]131: at_out: "AT+CIPMUX=1
" (13)
[at_get_reply]154: "
AT+CIPMUX=1

OK" (20)
[at_send_cmd]131: at_out: "AT+CIPSERVER=1,80
" (19)
[at_get_reply]154: "
AT+CIPSERVER=1,80
no change

OK" (37)
===== Show IP =====

```

(continues on next page)

(continued from previous page)

```
[at_send_cmd]131: at_out: "AT+CIFSR
" (10)
[at_get_reply]154: "
AT+CIFSR
+CIFSR:STAIP,"192.168.137.236"
+CIFSR:STAMAC,"5c:cf:7f:0b:5f:9a"

OK" (84)
```

Access server

The serial port feedback information above shows that the board has successfully connected to the target WiFi through ESP8266. It is set to the server mode by using the AT command, and the IP address of the server is also given.

At this point, use a PC or mobile phone to connect to the same WiFi, open a browser(recommend Google Chrome for PC), and enter the IP address to see the static HTTP page. Notice the IP address that you enter should be the same IP address shown in *Show IP* section at your serial port console. The content of your serial port console and browser is shown below:

```
===== send =====

The message is:
0,CONNECT
1,CONNECT

+IPD,0,384:GET / HTTP/1.1
Host: 192.168.137.236
Connection: keep-alive
Upgrade-Insecure-Requests: 1
User-Agent: Mozilla/5.0 (Windows NT 10.0; Win64; x64) AppleWebKit/537.36 (KHTML,
↪like Gecko) Chrome/70.0.3538.102 Safari/537.36
Accept: text/html,application/xhtml+xml,application/xml;q=0.9,image/webp,image/
↪apng,*/*;q=0.8
Accept-Encoding: gzip, deflate
Accept-Language: zh-CN,zh;q=0.9

Send Start
[at_send_cmd]131: at_out: "AT+CIPSEND=0,44
" (17)
[at_get_reply]154: "AT+CIPSEND=0,44

OK" (22)
[at_get_reply]154: "
>
Recv 44 bytes

SEND OK" (30)
[at_send_cmd]131: at_out: "AT+CIPSEND=0,93
" (17)
[at_get_reply]154: "
AT+CIPSEND=0,93

OK" (24)
[at_get_reply]154: "
>
Recv 93 bytes

SEND OK" (30)
[at_send_cmd]131: at_out: "AT+CIPSEND=0,93
```

(continues on next page)

(continued from previous page)

```

" (17)
[at_get_reply]154: "
AT+CIPSEND=0,93

OK" (24)
[at_get_reply]154: "
>
Recv 93 bytes

SEND OK" (30)
[at_send_cmd]131: at_out: "AT+CIPCLOSE=0
" (15)
[at_get_reply]154: "
AT+CIPCLOSE=0
0,CLOSED

OK" (32)
Send Finish
===== send =====

The message is:

+IPD,1,353:GET /favicon.ico HTTP/1.1
Host: 192.168.137.236
Connection: keep-alive
User-Agent: Mozilla/5.0 (Windows NT 10.0; Win64; x64) AppleWebKit/537.36 (KHTML,
↳like Gecko) Chrome/70.0.3538.102 Safari/537.36
Accept: image/webp,image/apng,image/*,*/*;q=0.8
Referer: http://192.168.137.236/
Accept-Encoding: gzip, deflate
Accept-Language: zh-CN,zh;q=0.9

Send Start
[at_send_cmd]131: at_out: "AT+CIPSEND=1,44
" (17)
[at_get_reply]154: "AT+CIPSEND=1,44

OK" (22)
[at_get_reply]154: "
>
Recv 44 bytes

SEND OK" (30)
[at_send_cmd]131: at_out: "AT+CIPSEND=1,93
" (17)
[at_get_reply]154: "
AT+CIPSEND=1,93

OK" (24)
[at_get_reply]154: "
>
Recv 93 bytes

SEND OK" (30)
[at_send_cmd]131: at_out: "AT+CIPSEND=1,93
" (17)
[at_get_reply]154: "
AT+CIPSEND=1,93

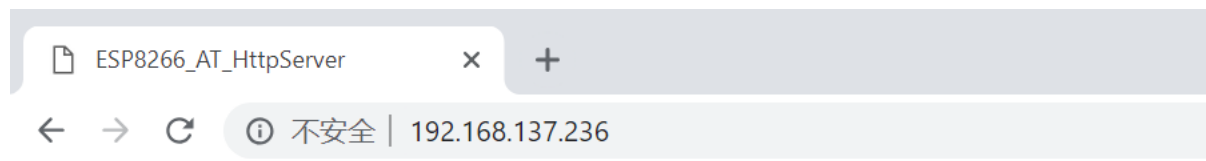
OK" (24)

```

(continues on next page)

(continued from previous page)

```
[at_get_reply]154: "  
>  
Recv 93 bytes  
  
SEND OK" (30)  
[at_send_cmd]131: at_out: "AT+CIPCLOSE=1"  
" (15)  
[at_get_reply]154: "  
AT+CIPCLOSE=1  
1,CLOSED  
  
OK" (32)  
Send Finish
```



Welcome to this Website

This Website is used to test the AT command about HttpServer of ESP8266.

Exercises

Referring to the embARC documents, using ESP8266 and TCN75 temperature sensor to build http server to make the page display the sensor temperature in real time.

3.2.3 BLE Communication

Purpose

- To get familiar with the wireless communication in IoT
- To get familiar with the usage of RN4020 BLE module on IoT Development Kit
- To learn the usage of APIs of RN4020 driver in embARC OSP

Requirements

The following hardware and tools are required:

- PC host
- A smartphone which supports BLE
- ARC GNU toolchain/MetaWare Development Toolkit
- ARC board (IoT Development Kit)
- embARC OSP package

- `embarc_osp/arc_labs/labs/lab6_ble_rn4020`

Content

The communication between smartphone and IoT Development Kit board with RN4020 BLE module.

- Setup RN4020 BLE module by using API of RN4020 driver.
- Connect smartphone and RN4020 by BLE, and check the data send by IoT Development Kit in smartphone.
- Send data from smartphone to IoT Development Kit board, and print this data value in terminal.

Principles

RN4020 BLE module is controlled by the user through input/output lines (that is 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, you might need to set up the RN4020 module as follows.

1. Configure UART which is 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. Run the command **SF, 1** to reset to the factory default configuration
4. Run the command **SN, IoT DK** to set the device name to be “IoT DK”
5. Run the command **SS, C0000001** to enable support of the Device Information, Battery Service, and User-Defined Private Service
6. Run the command **SR, 00002000** to set the RN4020 module as a server
7. Run the command **PZ** to clear all settings of the private service and the private characteristics
8. Run the command **PS, 11223344556677889900AABBCCDDEEFF** to set the UUID of user-defined private service to be 0x11223344556677889900AABBCCDDEEFF
9. Run the command **PC, 010203040506070809000A0B0C0D0E0F, 18, 06** to add private characteristic 0x010203040506070809000A0B0C0D0E0F to current private service. The property of this characteristic is 0x18 (writable and could notify) and has a maximum data size of 6 bytes.
10. Run the command **R, 1** to reboot the RN4020 module and to make the new settings effective
11. Run 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);
```

(continues on next page)

(continued from previous page)

```
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

Run 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

Run 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, see RN4020 Bluetooth Low Energy Module User's Guide.

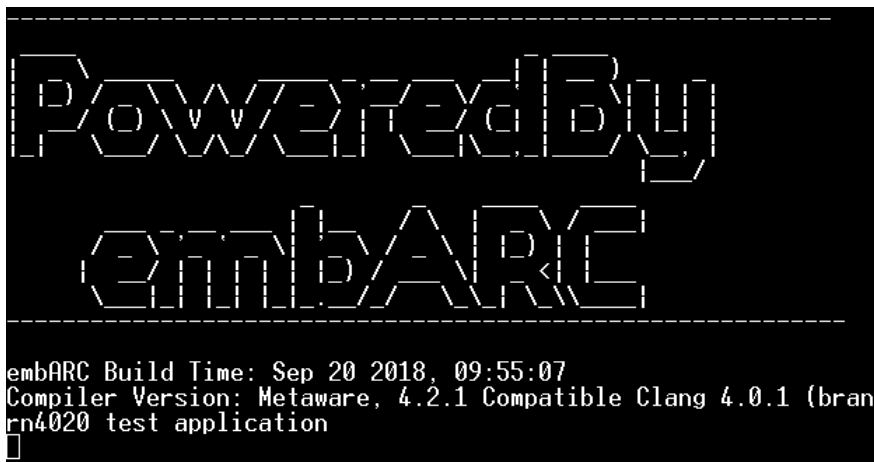
Steps

Run project

Open the serial terminal emulator in computer (for example, Tera Term), set as **115200 baud, 8 bits data, 1 stop bit and no parity**, and connect to the IoT Development Kit board.

Open cmd from the folder *embarc_osp/arc_labs/labs/lab6_ble_rn4020*, input the command as follows:

```
make BOARD=iotdk TOOLCHAIN=gnu run
```

A screenshot of a serial terminal window. At the top, there is a logo that reads "Powered By" in a large, stylized, outlined font, followed by "embARC" in a similar but smaller font. Below the logo, there is a dashed horizontal line. Underneath the line, the following text is displayed: "embARC Build Time: Sep 20 2018, 09:55:07", "Compiler Version: Metaware, 4.2.1 Compatible Clang 4.0.1 (bran", and "rn4020 test application".

```
Powered By
embARC
-----
embARC Build Time: Sep 20 2018, 09:55:07
Compiler Version: Metaware, 4.2.1 Compatible Clang 4.0.1 (bran
rn4020 test application
```

Then the output is displayed in the serial terminal. □

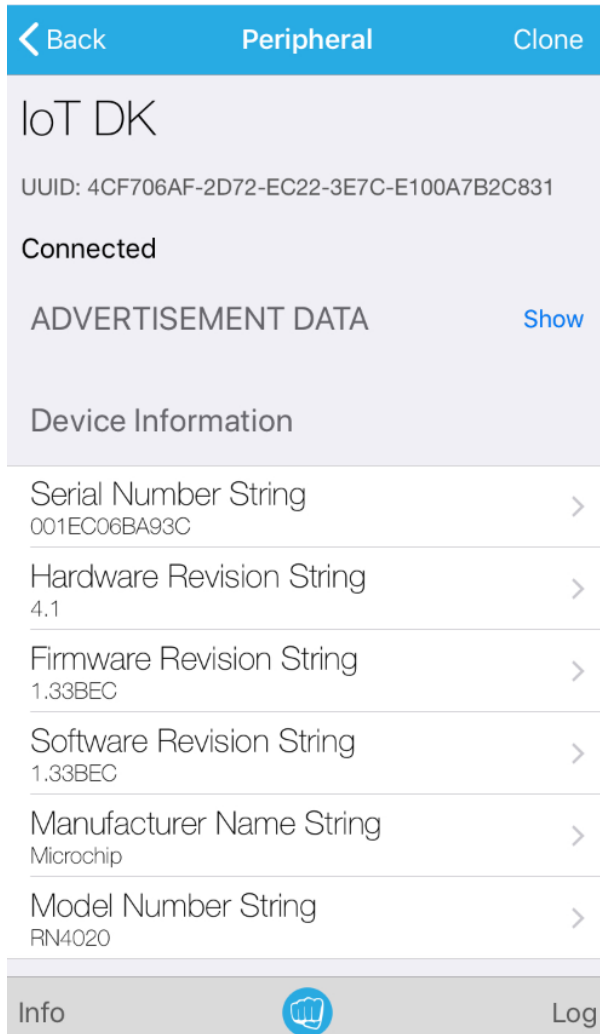
Connection

Open the BLE browser APP in smartphone (for example, LightBlue in IOS), and scan for BLE peripherals, connect the “IoT DK” device. Then the output is displayed in the serial terminal.

A screenshot of a serial terminal window, similar to the one above. It shows the "Powered By embARC" logo and build information. Below the dashed line, the text "embARC Build Time: Sep 20 2018, 09:55:07", "Compiler Version: Metaware, 4.2.1 Compatible Clang 4.0.1 (branches/release_40)", and "rn4020 test application" is displayed. The word "connected" is shown on the next line, highlighted with a red rectangular box. A small square cursor is visible at the bottom left.

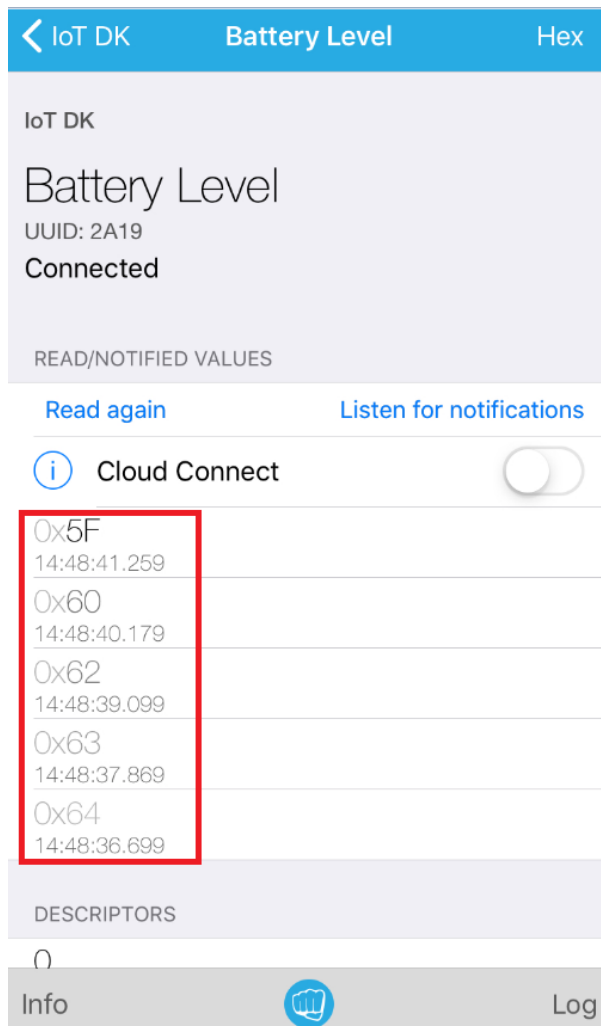
```
Powered By
embARC
-----
embARC Build Time: Sep 20 2018, 09:55:07
Compiler Version: Metaware, 4.2.1 Compatible Clang 4.0.1 (branches/release_40)
rn4020 test application
connected
□
```

And the device information in displayed BLE browser APP.

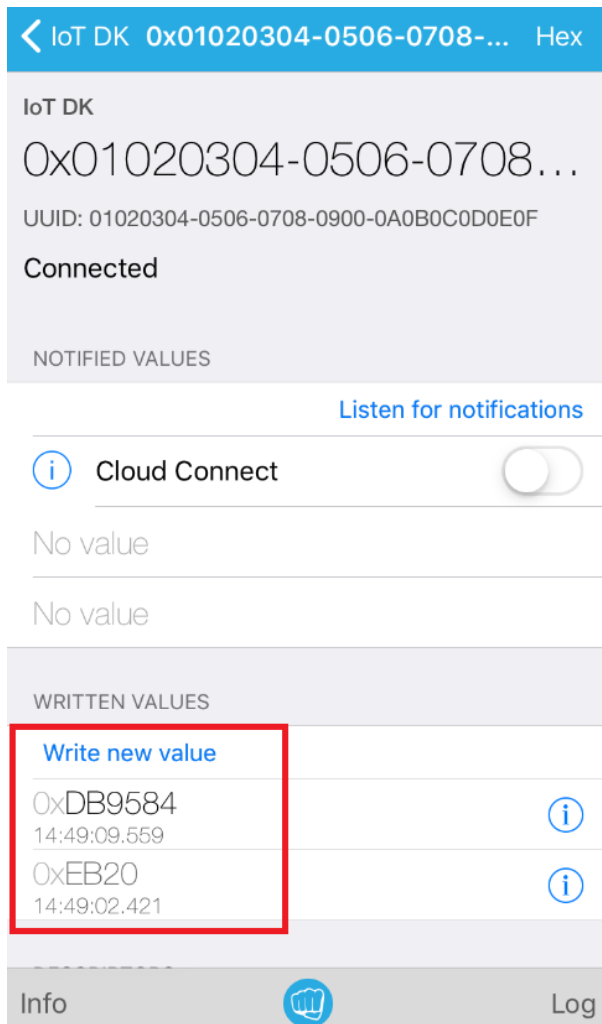


Communication

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



Write data in BLE browser APP. Check the received data in PC serial terminal.



Exercises

Try to use the received data in IoT Development Kit board, and do some control by using GPIO. (for example, LED on/off)

3.2.4 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

Requirements

The following hardware and tools are required:

- PC host
- GNU Toolchain for ARC Processors / 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 creates 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. You should know the basic functions of FreeRTOS.

Principles

Background

Operating system is software that controls basic hardware and software resources and provides access to them as a service for applications. In this sense applications that are used are said to be run on top of or inside the operating system.

There are different kind of operating systems and many definitions of operating systems depending on the available features. One of the main features of every operating system is how it organizes several tasks (programs) to work together. Some operating systems execute only one task at the time (these are called single-tasking) other allow multiple programs to work together (multi-tasking). Most common desktop operating systems are multi-tasking (Linux, Windows, and so on).

As processors on which programs are executed are sequential devices, technically only single program can be run at a time on a processor. However, multi-tasking does periodical switching between several tasks creating an illusion that these tasks work in parallel. The part of operating system that does this work is called scheduler. Scheduler is a routine that decides the order of execution of several tasks running on operating system.

Depending on scheduler multi-tasking algorithm operating systems are classified on real-time and non-real-time. In desktop operating systems (Linux, Windows) the usual approach of scheduler is to try to distribute processor time evenly between running application, so that each uses fair amount of resources. However, this approach has significant drawback which is unpredictable times when specific task are running. On the other hand, some applications (especially embedded) are time constrained and thus require deterministic execution of tasks. For example, if embedded system is controlling industrial machinery and software application is controlling some operation in the machine, which should be done at specific times disregarding of what other operations are pending. For this purpose, schedulers in some operating systems are made in a way to start tasks and predefine times. Such operating systems are called real-time operating systems (RTOS), because each task (application) running in RTOS can specify specific time (in milliseconds or other real time unit) at which it should be started. To organize this for several tasks, scheduler uses priorities set for tasks, so that if two applications requested to be called at the same time, the one with higher priority gets the resources.

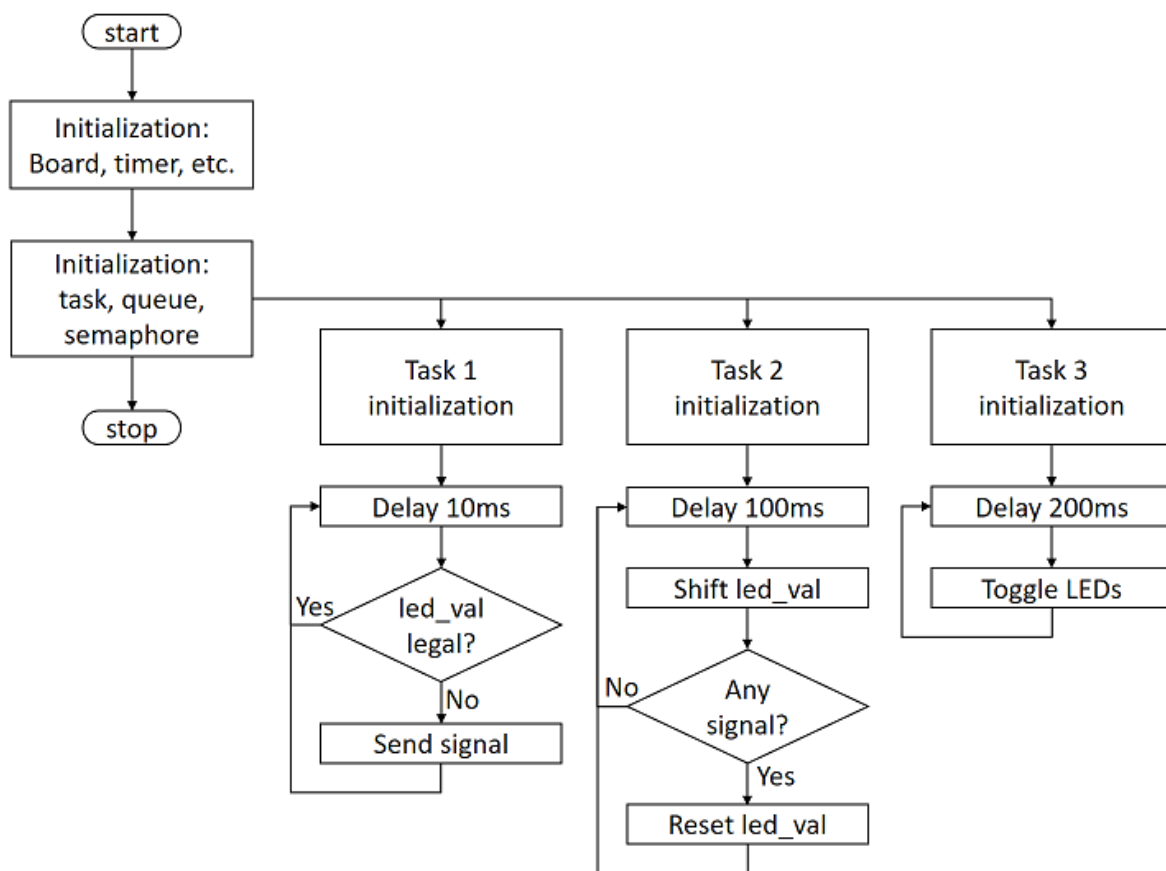
As resources becoming abundant for modern micro processors, the cost to run RTOS becomes increasingly insignificant. RTOS also provides event-driven mode for better utilization of CPU with efficiency.

FreeRTOS is an implementation of RTOS specially designed to be compact, easy to use and freely available (under GPL license with several exceptions). FreeRTOS source code is available for download at <http://freertos.org> and for different processor it could be ported (architecture specific code needs to be changed) so that it can operate on the specific processor. embARC OSP includes FreeRTOS port for DesignWare® ARC® processors that can be used to run applications using RTOS. FreeRTOS contains all the basic features of RTOS: tasks, scheduler, notifications, semaphores, mutexes, and so on.

Design

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

The following is the flow chart of the program:



Realization

The following is the example code of system , including various initialization and task time delay.

```

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

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

(continues on next page)

(continued from previous page)

```

#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);

    return 0;
}

```

(continues on next page)

(continued from previous page)

```

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 uncompleted code

The code is at `embarc_osp/arc_labs/labs/lab9_freertos`, uses an UART terminal console and run the code, the following message from program is displayed:

```
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 implies 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, the new function to learn is `xQueueReceive()`. This is a FreeRTOS API to pop and read an item from queue. See FreeRTOS documentation 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. See FreeRTOS documentation 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()`. See FreeRTOS documentation 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 cannot do these two things at the 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 noodle, each fork is placed between two adjacent philosophers. It is hard to eat noodles with one fork, so all philosophers need two forks in order to eat.

Write a program with proper console output to simulate this process.

3.2.5 ARC DSP: Compiler Optimizations

Purpose

- To understand Metaware compiler DSP extension options and optimization level
- To learn how to use Metaware compiler to optimize regular C code with DSP instructions

Requirements

The following hardware and tools are required:

- PC host
- MetaWare Development Toolkit
- ARC board (EM Starter Kit / IoT Development Kit)
- `embarc_osp/arc_labs/labs/dsp_lab_compiler_opt`

Content

An example code below contains a function “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;
}
```

Use Metaware compiler to optimize the C code with and without DSP extension options, and analyze the assembly code.

Principles

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

To optimize code with DSP extensions, two sets of compiler options are used throughout the lab: DSP Extensions options and optimization level.

DSP Extensions Options

Use embARC OSP build system to compile the code. 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=iotdk BD_VER=10 CUR_CORE=arcem9d TOOLCHAIN=mw ADT_COPT="-Hfxapi -Xdsp2  
↪ " OLEVEL=02
```

Options that are used in the lab are:

- `-Xdsp[1/2]`:
Enable DSP instructions
- `-Xdsp_complex, -Xdsp_divsqrt`:
Enable complex arithmetic DSP, divide, and 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 and transforms
- `-Hfxapi`: Use the Fixed Point API support library
Used with `-Xdsp`. Provides low level intrinsic support for ARC 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: Because ARC is configurable processor, different cores can contain different extensions on hardware level. Therefore, options set for compiler should match underlying hardware. On the other hand, if specific hardware feature is present in the core but compiler option is not set, it cannot be used effectively, if used at all. IOTDK Core default options are presented in tcf file.

Optimization level

MetaWare compiler has different optimization levels, which enables or disables various optimization techniques included in the compiler. You can 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 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.

Steps

1. Compiling with option -O0, DSP extensions will be specified in TCF file

Below is the list of options used when launching gmake:

```
gmake BOARD=iotdk BD_VER=10 CUR_CORE=arcem9d TOOLCHAIN=mw OLEVEL=O0
```

You can use the following command to generate disassembly code, and check assembly code for function “test”.

```
elfdump -T -S <your_working_directory>/obj_iotdk_10/mw_arcem9d/  
dsp_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
```

2. Compiling with DSP extensions, with -O2

Compile with:

```
gmake BOARD=iotdk BD_VER=10 CUR_CORE=arcem9d TOOLCHAIN=mw OLEVEL=O2
```

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.

3. Compiling with DSP extensions, with -O3

Compile with:

```
gmake BOARD=iotdk BD_VER=10 CUR_CORE=arcem9d TOOLCHAIN=mw OLEVEL=O3
```

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[]) {

```

Exercises

Remove “<<1” from test function and see changes in the output instructions.

3.2.6 ARC DSP: Using FXAPI

Purpose

- To understand what is ARC Fixed-point API (FXAPI)
- To learn how to use FXAPI to optimize DSP programs

Requirements

The following hardware and tools are required:

- PC host
- MetaWare Development Toolkit
- ARC board (EM Starter Kit / IoT Development Kit)
- `embarc_osp/arc_labs/labs/dsp_lab_fxapi`

Content

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.

Principle

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

Complex number multiplication

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

Is done using formula:

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

In this lab example multiplication and accumulation of two arrays of complex numbers are 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 can be defined that stores real and imaginary parts of the complex number. Therefore, 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;
```

(continues on next page)

(continued from previous page)

```

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;
result.imag = (short) acci;

return result;
}

```

The example keeps 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, therefore, 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 IoT Development Kit board for performance comparison

To compare performance of these two functions a simple application is created that performs 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 eight times, and after each loop a LED on board turns-on. In the result, LED strip on board works as a “progress bar” showing the process of looped multiplications.

The main performance check loop is shown in the following example. 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.

Steps

To test the following example, some modification of the code is required to have two loops with and without DSP. Firstly you must build DSP libraries for this particular configuration of IOTDK:

```

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

```

IoT Development Kit tcf file can be found in `embarc_osp/board/iotdk/configs/10/tcf/arcem9d.tcf`

Both examples are to be compiled with DSP extensions.

1. Run program without FXAPI

Build with the command:

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

With high optimization level functions using “short” type is compiled to use 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       %acc1,%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]
```

2. Run program with FXAPI

Rename `main.c.fxapi` to `main.c`, then execute the command:

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

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]
```

3.2.7 ARC DSP: Using DSP Library

Purpose

- To understand what is ARC DSP library
- To learn how to use DSP library to optimize DSP programs

Requirements

The following hardware and tools are required:

- PC host
- MetaWare Development Toolkit
- ARC board (EM Starter Kit / IoT Development Kit)
- `embarc_osp/arc_labs/labs/dsp_lab_dsp_lib`

Content

This lab uses matrix multiplication as an example where DSP library helps to efficiently use DSP extensions with shorter code. To use DSP Library and compare the execution speed of the programs with and without DSP library.

Principle

In this lab two implementations of matrix multiplication are shown: One manual implementation and the other using the DSP library.

Matrix multiplication

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

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

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

Implementation without DSP

The following example shows the implementation of matrix multiplication of two matrices containing “short” values. By convention, matrices here are implemented as 1D arrays with row-first indexing, where element `a_ik` is indexed as `aik`

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

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

/* ***** */

/* Matrix manipulation functions */
```

(continues on next page)

(continued from previous page)

```

/* 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;
    }

    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++) {

```

(continues on next page)

(continued from previous page)

```

                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, implementing matrix multiplication using DSP library requires initialization of matrix arrays (1D) and call to `dsp_mat_mult_q15`. The overall code is 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) that the result in `c` is not rounded 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
#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[]) {

```

(continues on next page)

(continued from previous page)

```

    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;
}

//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];
//            }
//        }
//    }
//}

```

(continues on next page)

(continued from previous page)

```
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");
}
```

Using IoT Development Kit board for performance comparison

Note: Create an IoT Development Kit 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. Run the example and compare computational delay with and without DSPLIB.

Steps

Firstly you must build DSP libraries for this particular configuration of IOTDK:

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

IoT Development Kit tcf file can be found in `embarc_osp/board/iotdk/configs/10/tcf/arcem9d.tcf`

Both examples are to be compiled with DSP extensions.

1. Run program without DSP library

Build with the command:

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

2. Run program with DSP library

Rename `main.c.dsplib` to `main.c`, then execute the command:

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

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 does not 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) is less efficient than direct use of DSPLIB.

3.3 Exploration

3.3.1 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

Required Hardware

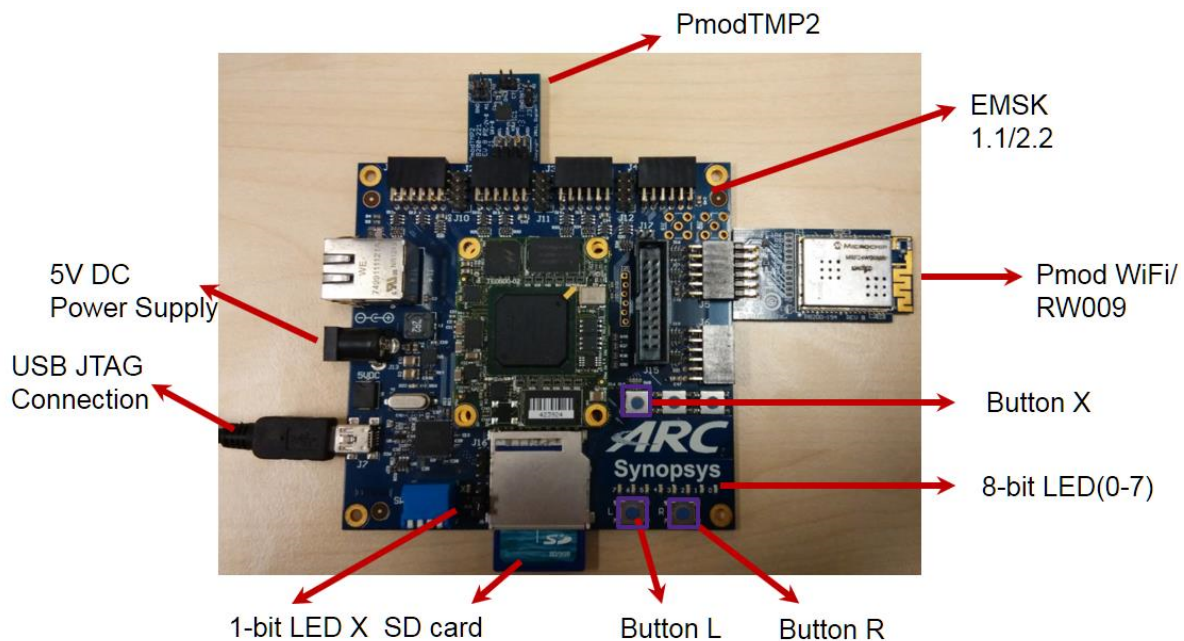
- [DesignWare ARC EM Starter Kit (EMSK)]
- [Digilent PMOD WiFi (MRF24WG0MA)]
- [Digilent PMOD TMP2]
- SD Card
- WiFi Hotspot (default SSID: **embARC**, Password: **qazwsxedc**, WPA/WPA2 encrypted)

Required Software

- MetaWare or ARC GNU Toolchain
- Serial port terminal (e.g. Putty, Tera-term or Minicom)

Hardware Connection (EMSK Board)

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



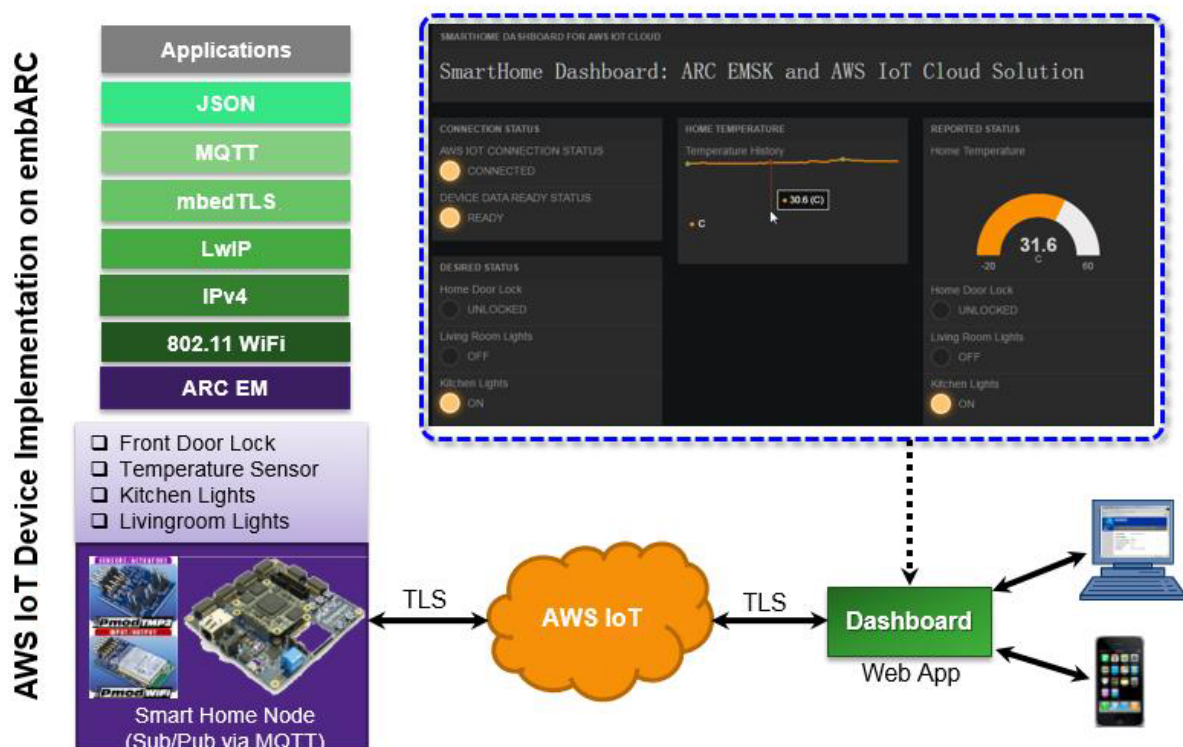
- Configure your hardware with proper core configuration.

- The hardware resources are described at 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 cloud platform, devices can securely interact with cloud applications and other devices. AWS IoT also supports MQ 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 on-board 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 a controlling platform that communicates with the EMSK Board with MQTT protocol. A HTML5 Web APP is designed to provide a dash board in order to monitor and control smart home nodes.

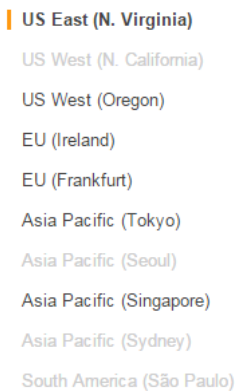
Steps

Creating and setting smart home node

1. Create an AWS account at [\[Amazon AWS Website\]](#). Amazon offers various account levels, including a free tier for AWS IoT.
2. Login AWS console and select AWS IoT.

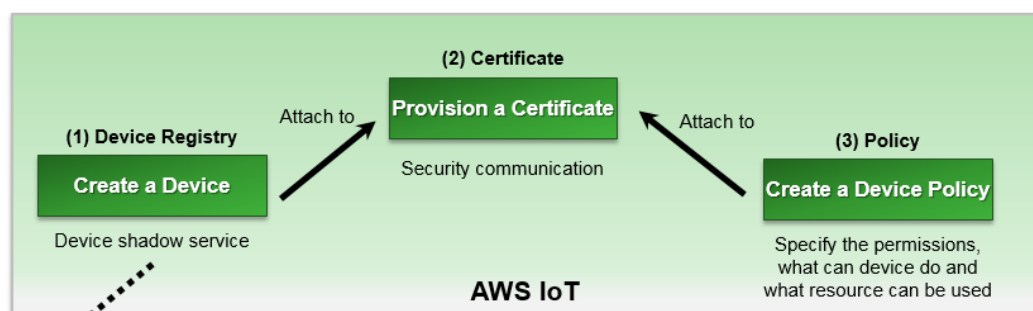


3. Select an appropriate IoT server in the top right corner of the AWS IoT console page. As an example US East (N. Virginia) server is selected, you may select other server as you see fit.



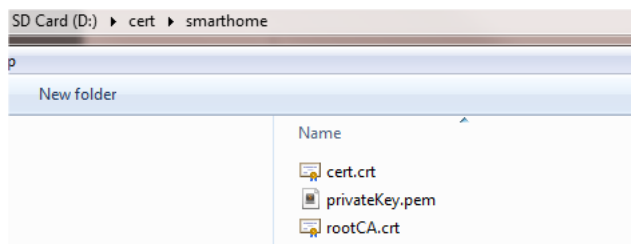
4. Create your smart home node in the thing registry and generate X.509 certificate for the node. Create an AWS IoT policy. Then attach your smart home node and policy to the X.509 certificate.

Note: for more details, see [\[Using a Smart Home IoT Application with EMSK\]](#)

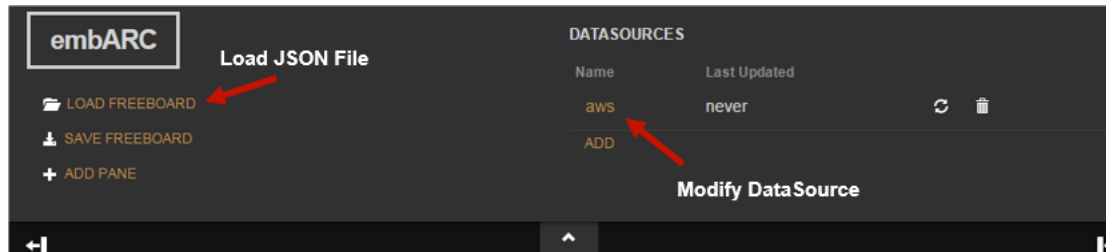


Topics

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 the forms.
- TYPE: Choose AWS IoT.
 - NAME: Name is aws.

DATASOURCE

Receive data from an MQTT server.

TYPE: **AWS IoT**

NAME: **aws**

AWS IOT ENDPOINT: **input_your_own_endpoint**
Your AWS account-specific AWS IoT endpoint. You can use the AWS IoT CLI describe-endpoint command to find this endpoint

REGION: **input_your_own_region**
The AWS region of your AWS account

CLIENT ID:
MQTT client ID should be unique for every device

ACCESS KEY: **input_your_own_accesskey**
Access Key of AWS IAM

SECRET KEY: **input_your_own_secretKey**
Secret Key of AWS IAM

THINGS: **Thing**

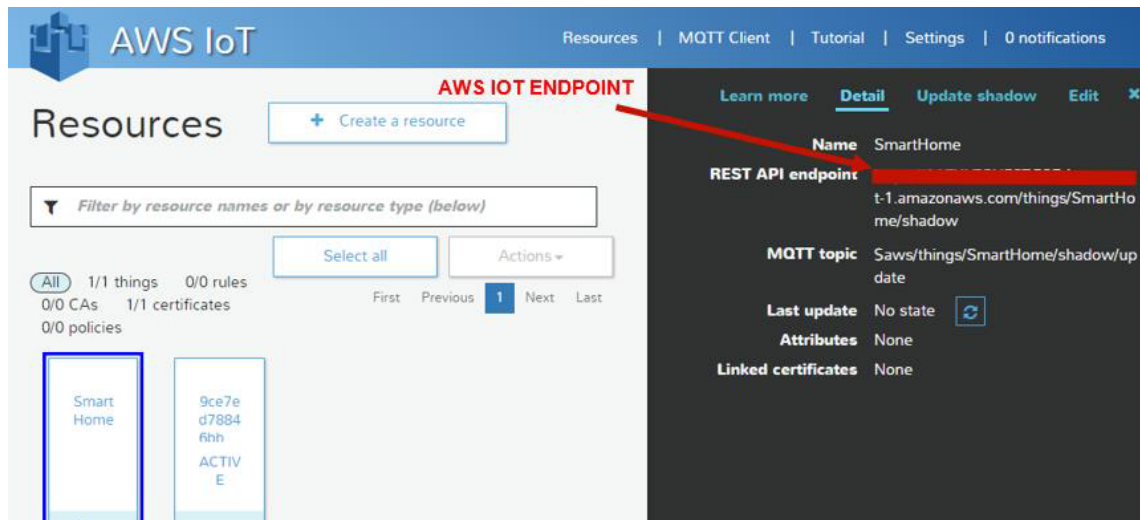
SmartHome

ADD

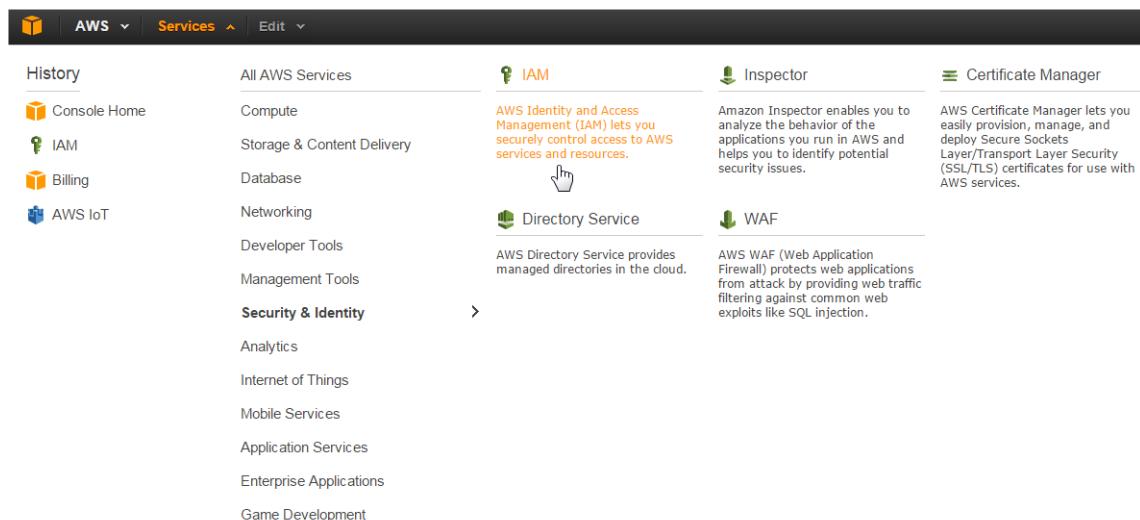
AWS IoT Thing Name of the Shadow this device is associated with

SAVE CANCEL

- AWS IOT ENDPOINT: Go to AWS IoT console and click your smart home node “SmartHome”. Copy the content `XXXXXXXXXXXXX.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://XXXXXXXXXXXXXXXXX.iot.us-east-1.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**.

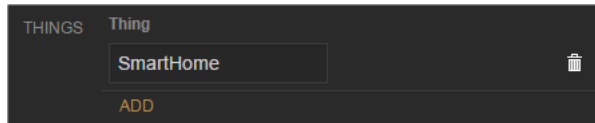


Go to user page and click **Create New Users**. Enter User Names **AWSIoTUser**. Then download user security credentials, Access Key ID, and Secret Access Key. Copy Access Key ID to ACCESS KEY and Secret Access Key to SECRET KEY.



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://XXXXXXXXXXXXXXXXX.iot.us-east-1.amazonaws.com/things/SmartHome/shadow>. **AWS_IOT_MQTT_HOST** should be `XXXXXXXXXXXXXXXXX.iot.us-east-1.amazonaws.com`.

```
// Get from console
// =====
#define AWS_IOT_MQTT_HOST      "XXXXXXXXXXXXXXXXX.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
```

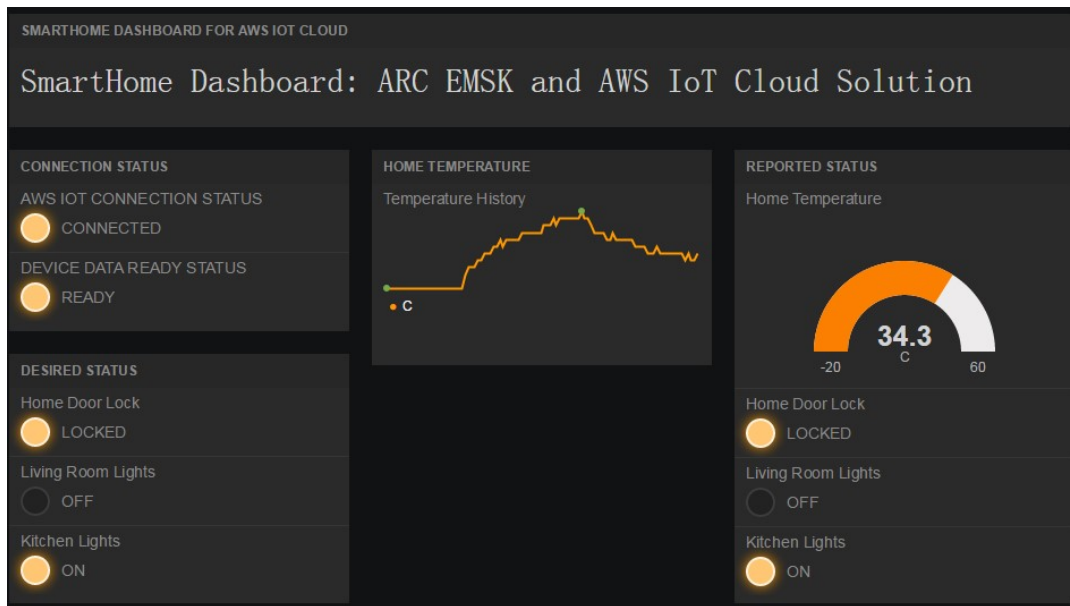
2. Use USB cable to connect the EMSK board. Set the baud rate of the terminal emulator to 115200.
3. Insert the SD Card into the EMSK board SD Card slot. Run the AWS IoT application using JTAG. Go to *embARC/example/freertos/iot/aws/smarthome_demo* in command-line, run the following command:

```
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 displayed in the terminal emulator. “Network is ok” is displayed 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"}
*****
```

5. Try out functions of EMSK and Dashboard. You can press the button L/R/X to see LED toggling on board, and the status of LEDs also changes on dashboard web app. You can also click the lights of *DESIRED STATUS* pane on the dashboard app, and check the reactions of LEDs status 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: You could find related demo codes [\[here\]](#)

4.1 Reference

1. Online docs
2. ARC EM Starter Kit Webpage
3. ARC IoT Development Kit Webpage
4. Github Repository of embARC Open Software Platform (OSP)

CHAPTER 5

Indices and tables

- `genindex`
- `search`