Freescale Semiconductor
Application Note

Document Number: AN4129
Rev. 0, 01/2012

Building Custom Applications on MMA9550L/MMA9551L

by: Fengyi Li
Applications Engineer

1 Introduction

The Freescale MMA955xL Xtrinsic family of devices are high-precision, intelligent accelerometers built on the Freescale ColdFire version 1 core.

The MMA955xL family's microcontroller core enables the development of custom applications in the built-in FLASH memory. Programming and debugging tasks are supported by Freescale's CodeWarrior Development Studio for Microcontrollers (Eclipse based IDE).

The MMA9550L/MMA9551L firmware platform is specifically designed to ease custom application integration. Building custom applications on the built-in firmware platform provides an organized and more code-efficient sensing system to monitor development tasks, which reduces the application development cycle.

This document introduces the code architecture required for creating a custom application, the instructions for binding it with the Freescale firmware platform, and the available tools that support the custom application development on the MMA955xL family of devices.

Contents

2 Architecture

2.1 Top-level diagram

Custom applications are built on top of the Freescale firmware platform. The organization of the architecture is shown in Figure 1.

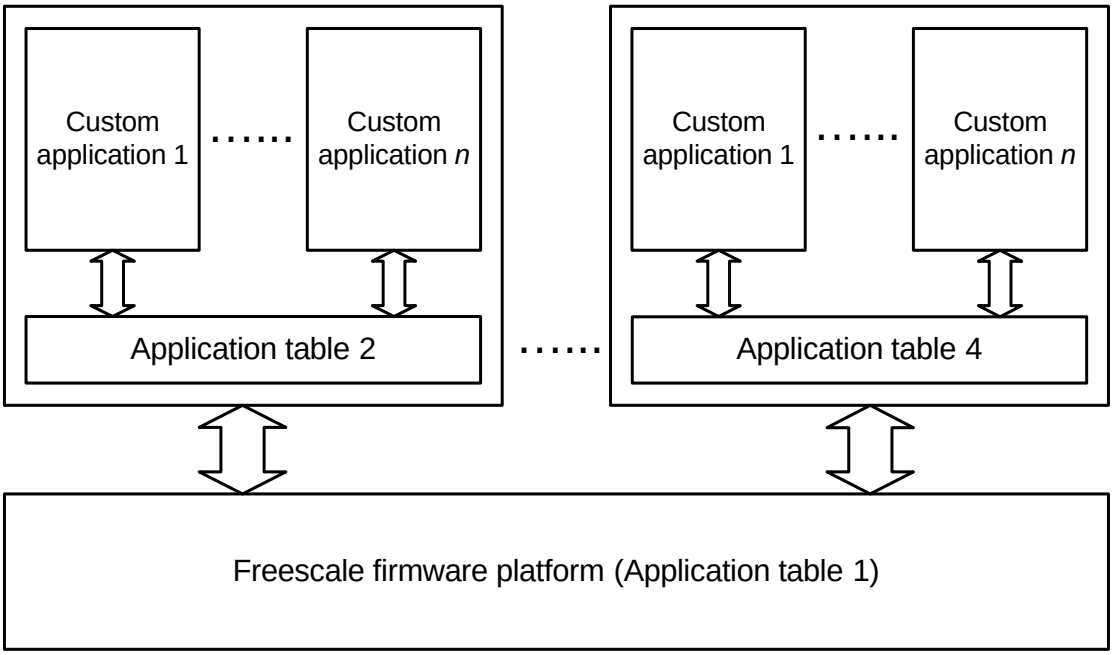

Figure 1. Architecture of custom applications and Freescale platform binding

At power on or reset, the Freescale bootloader searches for application tables in the FLASH memory. The bootloader recognizes the start of an application table by the special value 0x9550C0DE at a 512 byte device FLASH page boundary. The Freescale applications are in one application table, and up to three additional custom application tables can be included. For the same reason, users can stack up to three FLASH custom images in the custom FLASH.

To assist users with their applications development, the MMA955xL template is available on the Freescale website. Additional details are available in Section 3.1, “MMA955xL template,” on page 6. Corresponding to this block diagram, the source and header files of the MMA955xL template are organized in the same way. This template supports all MMA955xL family devices except the MMA9559L device which has its own template. The custom applications program is located in the **CustomAppx.c** and **CustomAppx.h** files. The code to combine all custom applications resides in **main.c** and **main.h** files.

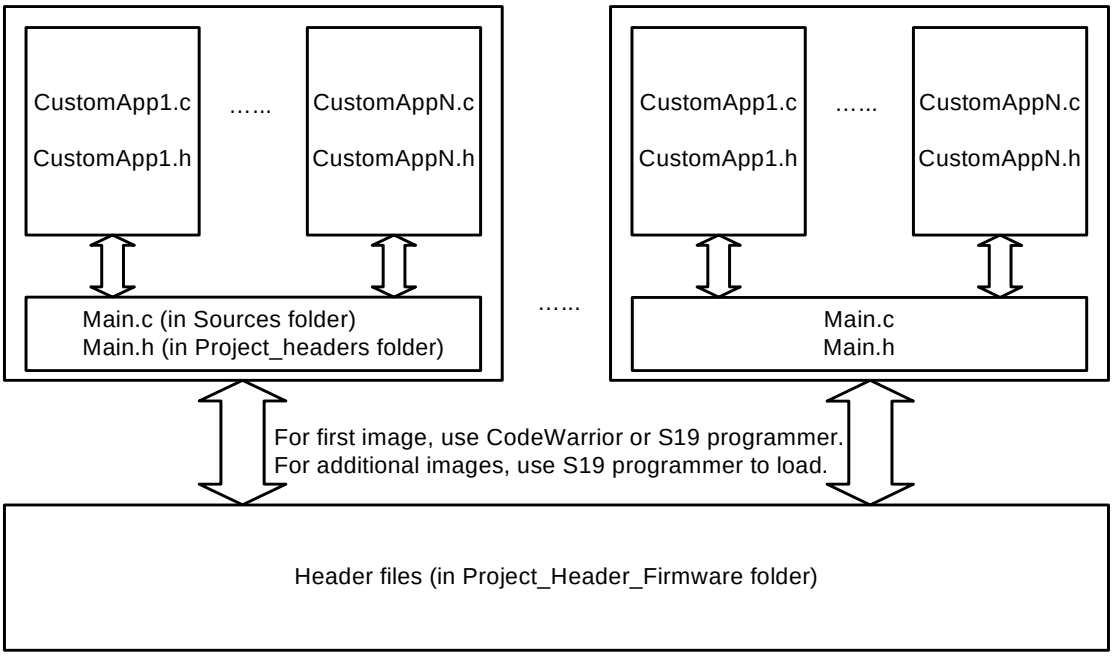

Figure 2. Project files in architecture drawing

Users can program the initial custom image into a MMA955xL device using CodeWarrior. Each CodeWarrior project should contain only one application table. For information about programming multiple custom images to the MMA955xL FLASH, please refer to the MMA955xL application notes at http://www.freescale.com/webapp/sps/site/prod_summary.jsp?code=MMA9550L.

2.2 Memory space

2.2.1 FLASH

MMA955xL family devices have a total of 16K FLASH memory. The FLASH contains 32 pages; each page size is 512 bytes. The space is shared between the user application space, which occupies the top region of the FLASH, and the Freescale applications and infrastructure, which occupy the bottom region. The actual number of bytes varies, based on the device, as shown in Figure 3.

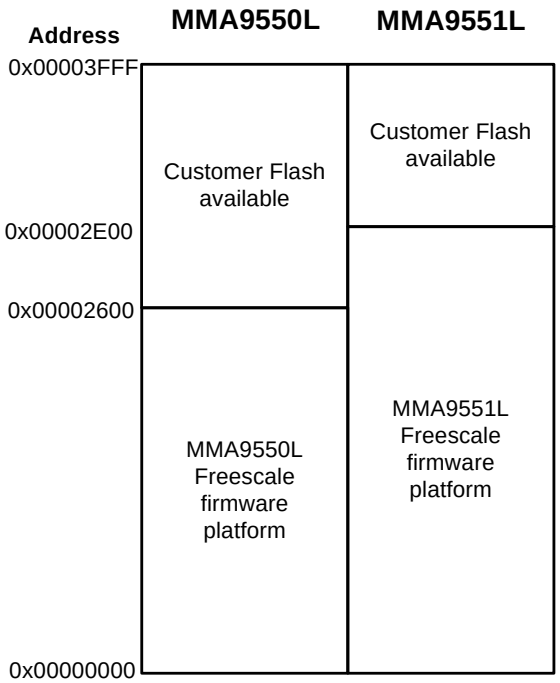

Figure 3. FLASH memory allocation

The memory space on the upper FLASH region is available for customer use. MMA9550L devices can accommodate 6656 bytes of customer code, and MMA9551L devices can accommodate 4608 bytes of customer code.

By default, the customer code starts at the first byte of the customer FLASH region. This value can be changed in the linker file **Project.lcf**. To modify the start address, if necessary, it is important to start the custom application on the page boundary to ensure that the code is recognizable to the Freescale bootloader.

2.2.2 RAM

MMA955xL family devices contain 2K RAM shared between the Freescale applications and infrastructure and the user applications. Space is allocated from the top of the RAM down, beginning with the Freescale stack space, followed by the user stack space, the user heap memory and then the Freescale heap memory. Both of the stacks are 192 bytes deep. The user heap memory size depends on the device. MMA9550L devices have 576 bytes of user memory; MMA9551L devices have 452 bytes. The size of the heap and stack is illustrated in Figure 4.

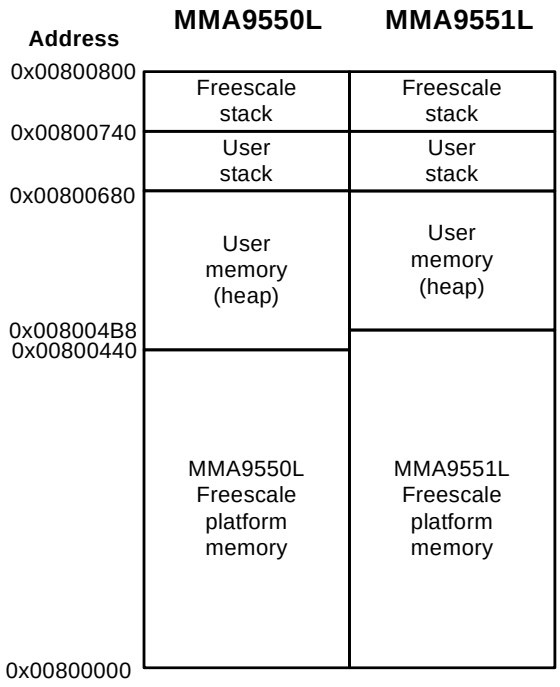

Figure 4. RAM memory allocation

When planning the memory space of their code, users need to understand that the RAM space for variables is assigned based on the variable's definition. The CodeWarrior compiler assigns *local* variables to the stack space, which does not store the variable content after exiting. In order to preserve the variable content, users must allocate these variables into the heap by calling the Freescale API function `request_ram_ptr ()`.

NOTE

Users must *not* use global variables nor static variables. These variable types are not supported and will corrupt the firmware's normal operation.

Memory allocation details are covered in Section 4.2, "Define RAM memory for custom applications" and in the MMA955xL Software Reference Manual.

3 Tools to build custom projects

NOTE

The term Function Block Identifier or FBID as referenced in this document has been changed to Application Identifier or APP_ID. The user interface illustrated in this document has not been updated to reflect this change.

3.1 MMA955xL template

The MMA955xL template project is available to download from the Freescale website and can be used as a starting point for MMA9550L and MMA9551L custom application development. A different template is available for MMA9559L.

This template project contains one application that reads the accelerometer data, the use of this data and the gesture function status. It toggles GPIO7 pin at 24.4 Hz when certain conditions are met. This template also includes the power consumption level, event FIFO and data FIFO setups.

The following section lists the steps of how to use this template, including download, importing and exporting the template, modifying code if necessary, building the template, and downloading and debugging the code.

3.1.1 Download the template

The MMA955xL template is available here:
http://www.freescale.com/webapp/sps/site/prod_summary.jsp?code=MMA9550L

3.1.2 Import the MMA955xL template in CodeWarrior IDE

1. Open CodeWarrior 10.1 ID and create a workspace for the program.

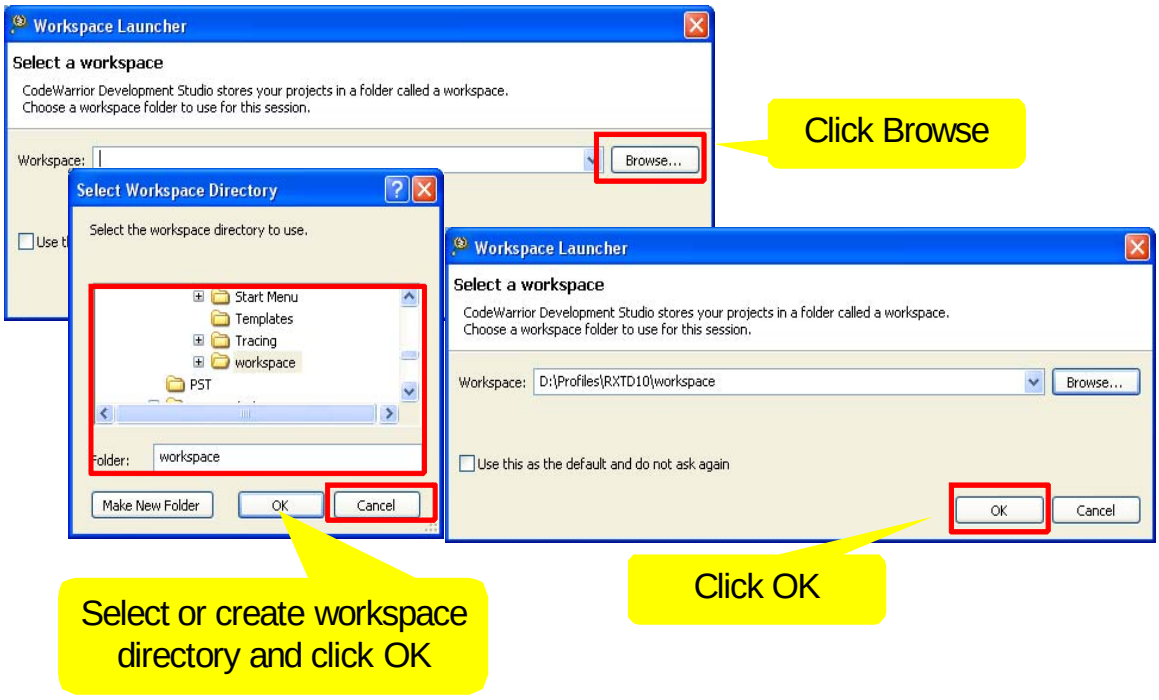

Figure 5. Create a workspace

2. Open the workbench from the welcome page

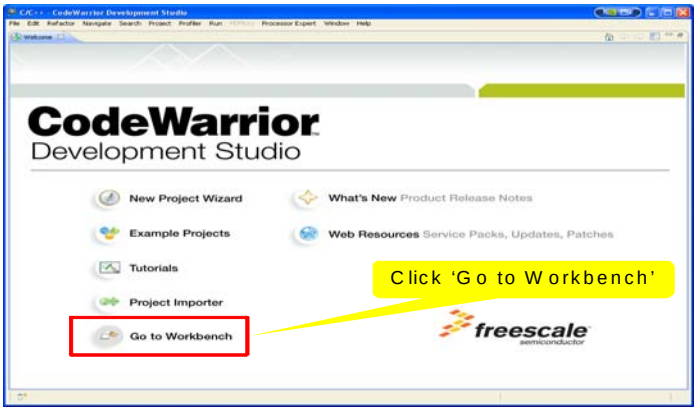

Figure 6. Welcome page link to "Go to Workbench"

3. Click on the **C/C++ button** on top right corner to view the project. The alternative to call C/C++ perspective is **Menu > Window > Open Perspective > Other… >C/C++**.

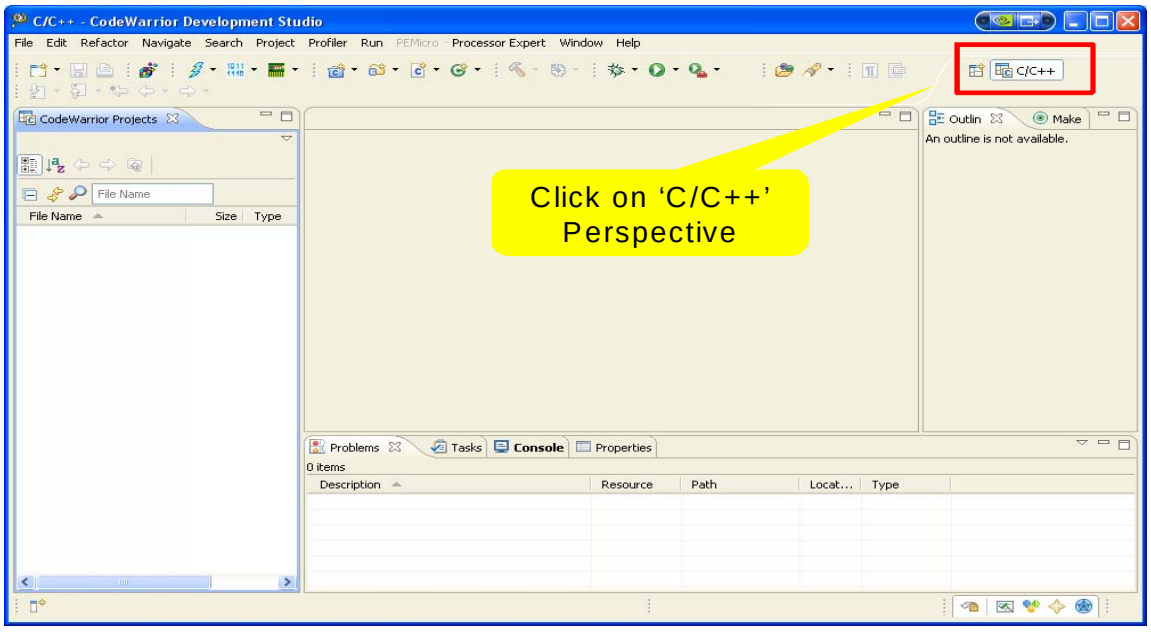

Figure 7. View the project using the C/C++ perspective

4. Import the MMA955xL template by choosing **Menu > Files > Import**.

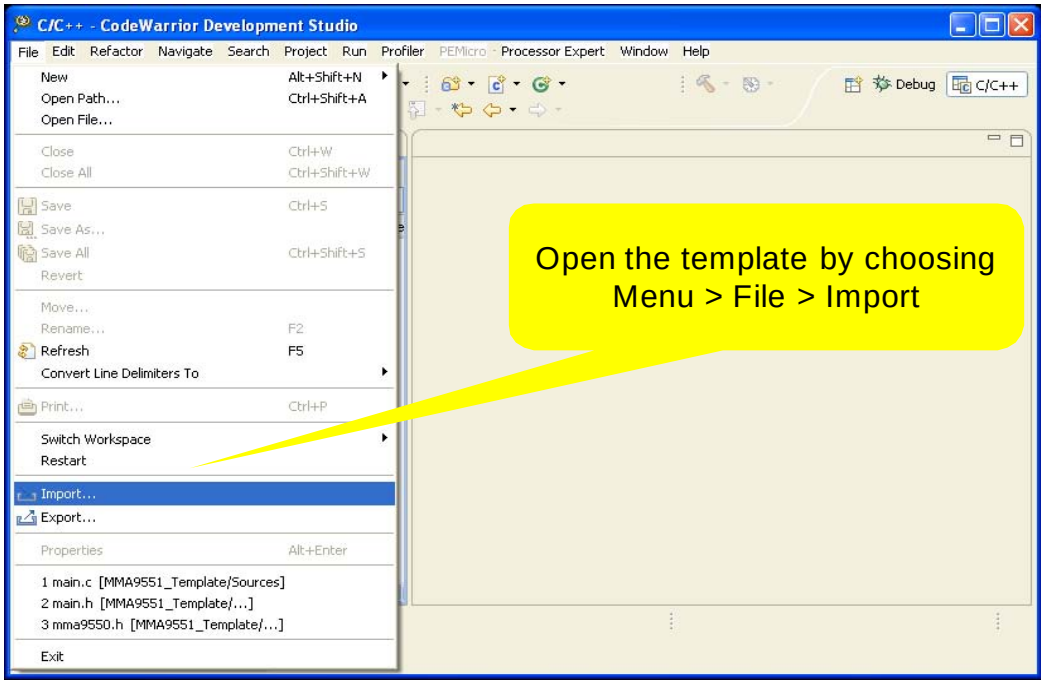

Figure 8. Import the MMA955xL template

5. From the Import/Select dialog, choose **General > Existing Projects into Workspace > Next**.
6. From the Import/Import Projects dialog, choose **Select archive file**.
7. Click **Browse** to location of the saved template.
8. From the Import/Select archive dialog, select **MMA9551_Template**.
9. Click **Open** to select the MMA9551_Template to be visible in Import Project.

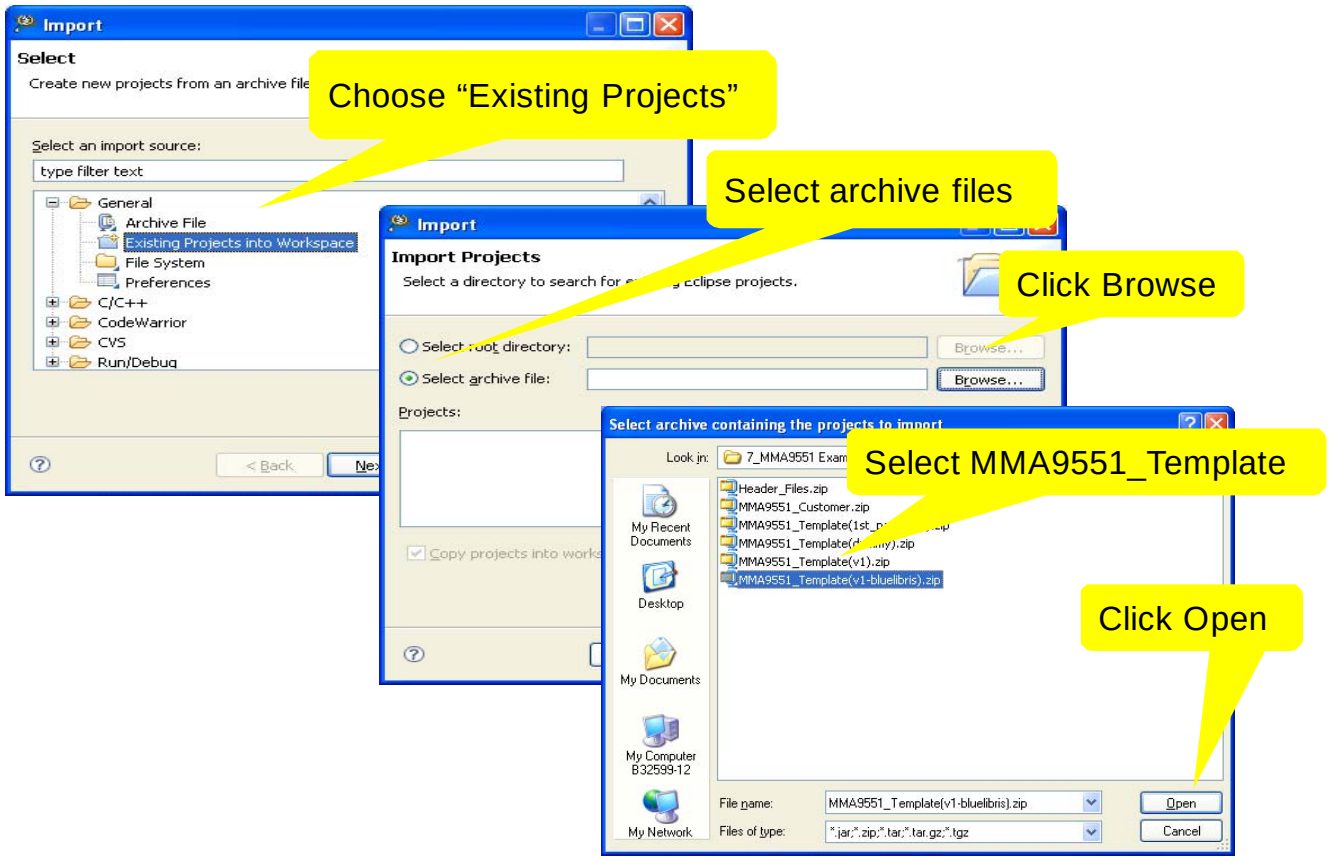

Figure 9. Choose the MMA955xL template

10. Click **Finish** to complete the import and open the MMA9551_Template project.

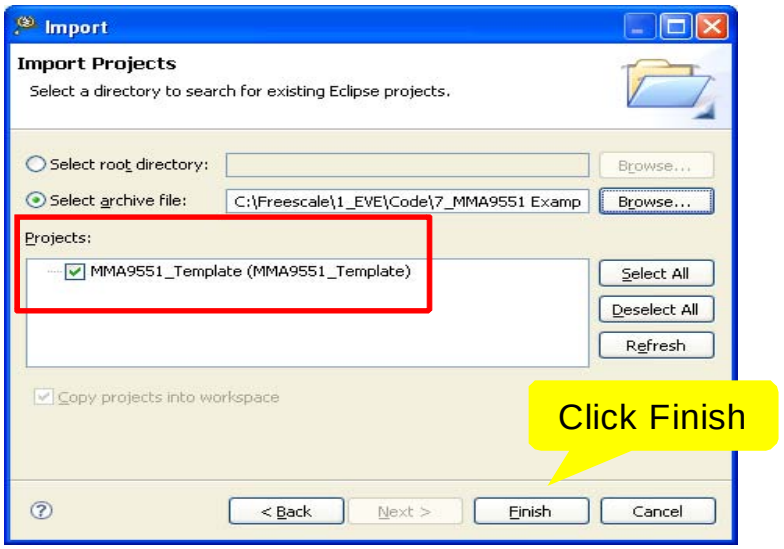

Figure 10. Complete the MMA955xL template import

11. Export the project, by choosing **Menu** > **Files** > **Export**. Choose **General** > **Achieve** to save the project in .zip or .tar format.

3.1.3 Code modifications based on the device

If building projects for MMA9551L devices, nothing needs to be changed. If MMA9550L is the target device, because of the built-in differences between MMA9550L and MM9551L, users need to modify the **Project.lcf** and application codes. See Table 1 for details.

Table 1. Differences between MMA9550L and MMA9551L

Item	MMA9550L	MMA9551L	Impact File
FLASH available space	6656 bytes	4608 bytes	Project.lcf
RAM available space	576 bytes	452 bytes	Project.lcf
Gesture functions with Freescale applications	No	Yes	application codes

3.1.3.1 Modify the application code

According to Table 1, MMA9550L devices provide more custom memory space compared to the MMA9551L. The MMA9551L gesture functions include portrait-landscape, tap, high-g / low-g, and tilt applications. Users need to comment out the code referring to these gesture applications. The code resides in **CustomApp1.c**.

3.1.3.2 Modify the Project.lcf file

Project.lcf is the linker file that tells the linker where to place the user memory allocation for both FLASH and RAM. It resides in the folder **Linker_Files** underneath the **Project_Settings** folder.

The beginning section of the linker file **Project.lcf** is shown in Example 1. The "#" symbol is interpreted as a comment marker by the linker. Note there is no "#" before the memory definitions for MMA9551L. Remove the "#" sign in front of the MMA9550L memory section to enable the compiler to successfully compile for the MMA9550L, and comment out the MMA9551L memory section to disable the compilation for MMA9551L. The results are shown in Example 2.

Example 1. Project.lcf for MMA9551L

```
# Memory ranges
MEMORY {
# code (RX) : ORIGIN = 0x00002600, LENGTH = 0x00001A00 # Use this line when using MMA9550L
code (RX) : ORIGIN = 0x00002E00, LENGTH = 0x00001200 # Use this line when using MMA9551L
ram (RWX) : ORIGIN = 0x00800440, LENGTH = 0x00000000 # 0 RAM space for Global or static variables
}
```

Example 2. Project.lcf after modifying for MMA9550L

```
# Memory ranges
MEMORY {
code (RX) : ORIGIN = 0x00002600, LENGTH = 0x00001A00 # Use this line when using MMA9550L
# code (RX) : ORIGIN = 0x00002E00, LENGTH = 0x00001200 # Use this line when using MMA9551L
ram (RWX) : ORIGIN = 0x00800440, LENGTH = 0x00000000 # 0 RAM space for Global or static variables
}
```

3.1.4 Build the project

To set up the build configuration and build the project, please use the following steps.

1. Select the project to be built from the CodeWarrior Project viewer.
2. From the main menu, select **Project** > **Properties** to open the build configuration.

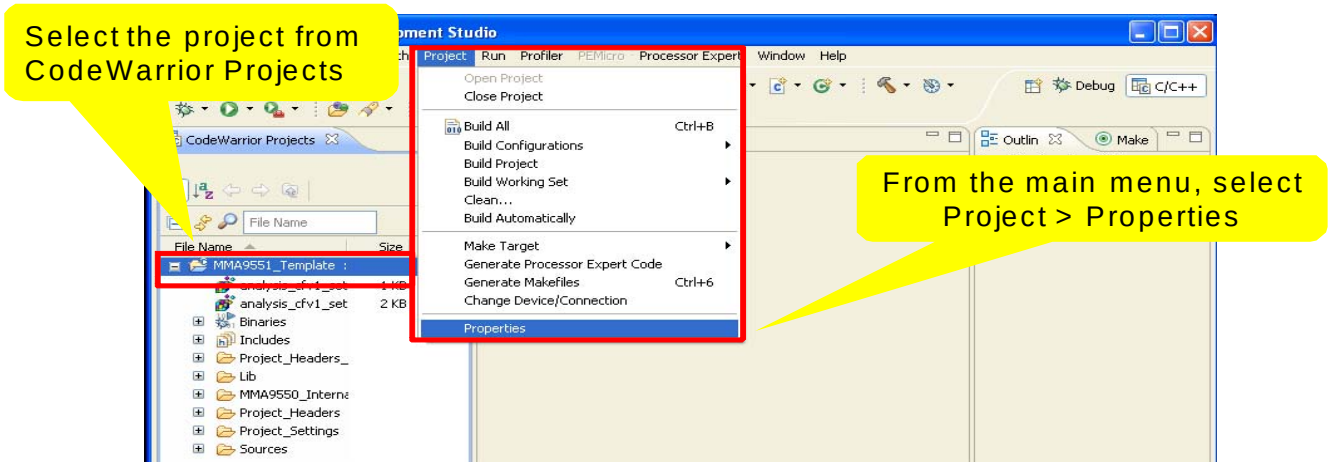

Figure 11. Open build properties

3. Choose **C/C++ Build > Settings** in the left menu, to open the settings configurations.
4. Under Tool Settings tab, select **ColdFire Linker > Input** and enter the entry function name prepended with an "_". Confirm that the program entry point is sent to the first function you want to run at runtime. In the MMA955xL template, this function is `user_app_init()`.
5. Click **Apply** to activate the changes.

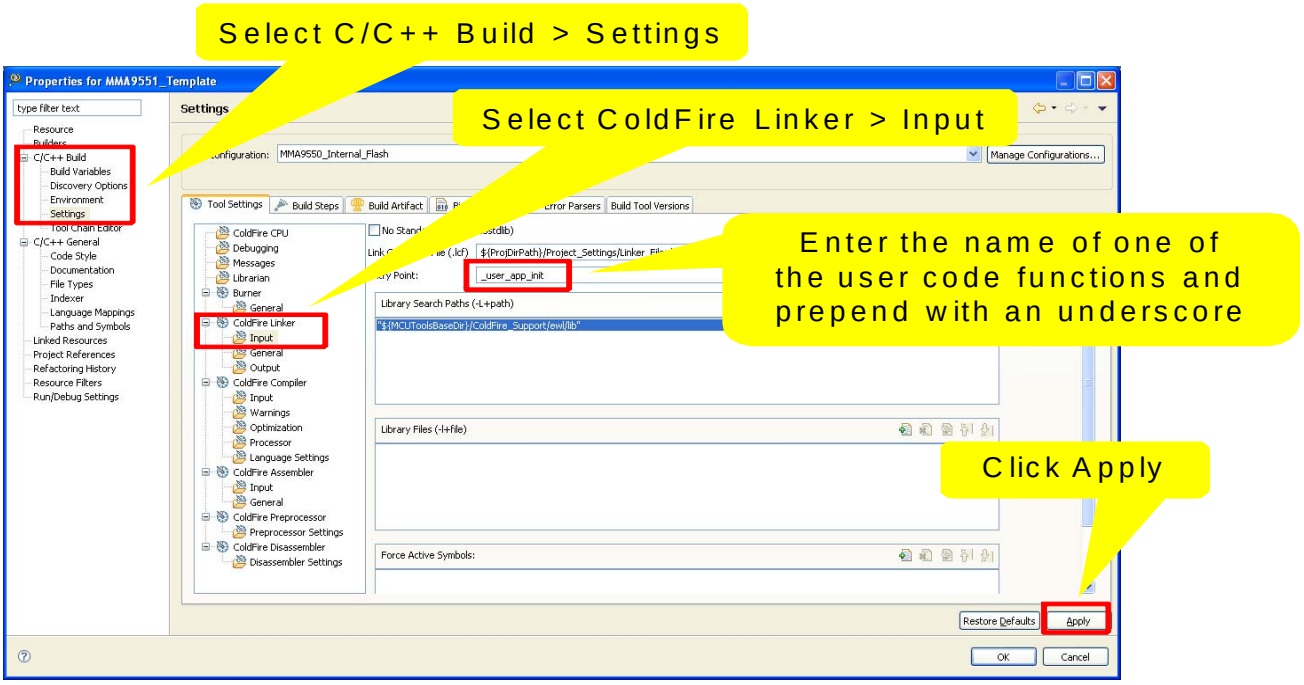

Figure 12. Set entry point function

6. Set compiler options. From the Tools Settings tab, select **ColdFire Compiler** > **Processor**.
7. In the fields on the right, confirm the following settings:
   - Struct Align(-align) is set to byte
   - Code Model is set to Near Relative (pc16)
   - Data Model is set to Far (32 bit)
   - Floating Point is set to Software
8. Disable the .sdata by entering “0” in the text box next to Use .sdata/.sbiss for (byte in integer between -1...32K).
9. Click **Apply**.
10. Click **OK** to exit the build configuration window.

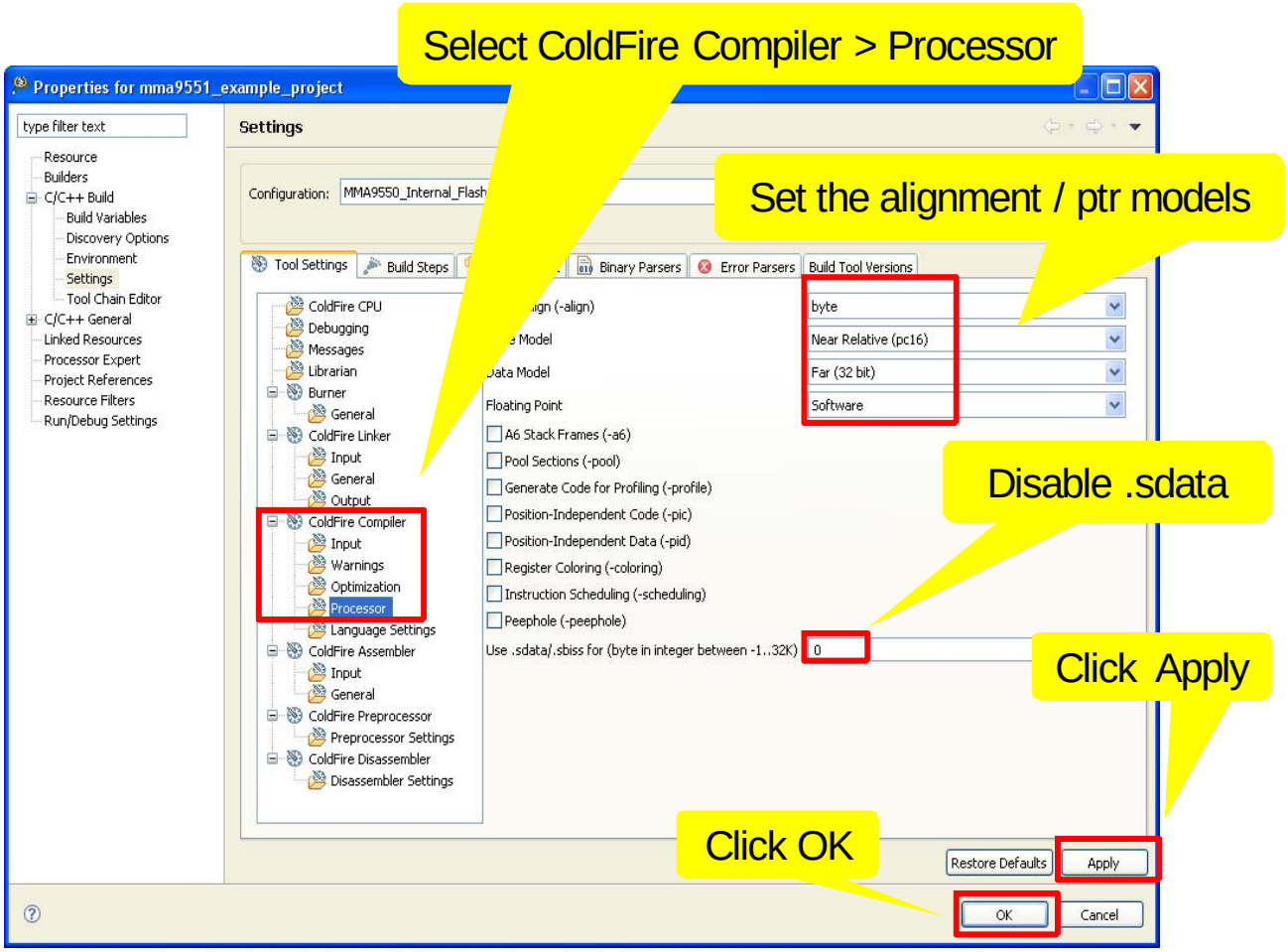

Figure 13. Set compiler options

11. Under the CodeWarrior Projects tab, right click on the project, and select **Clean Projec**t on the menu.

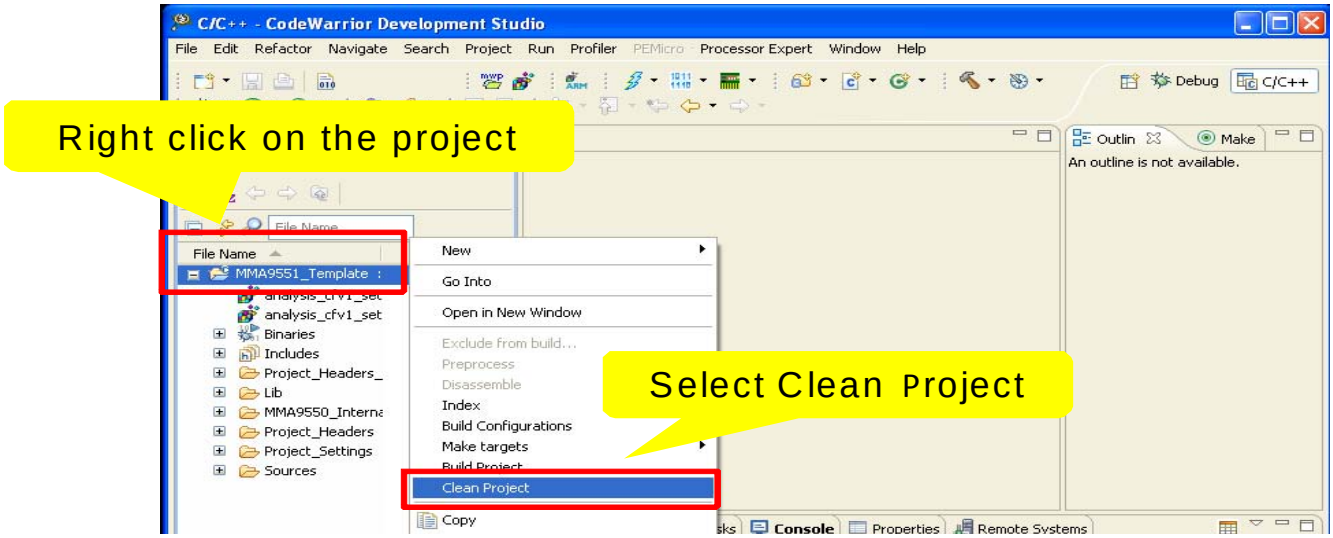

Figure 14. Clean project

12. Right click on the project again and select **Build Project** on the menu. The Build Project progress bar will display until the project build is complete.

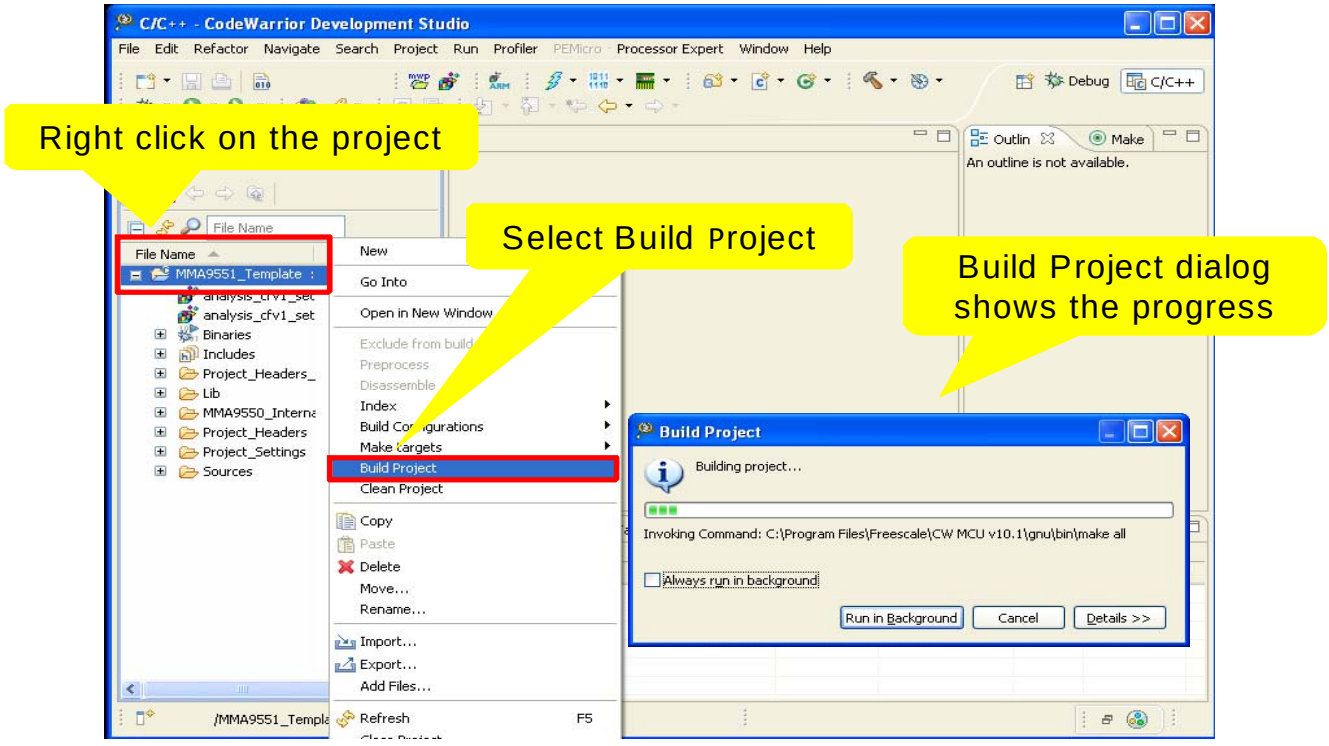

Figure 15. Build project

3.1.5 Download and Attach

CodeWarrior 10.1 IDE uses two configurations to connect to the physical device: Download and Attach. For those who have used CodeWarrior 6.3, these are equivalent to Connect and Hot Sync.

To be more specific, the Download configuration is used to program MMA955xL. It enables users to connect to the device, erase, and reprogram the user FLASH region. Users need to make sure that they don't proceed to debug the program directly after the download. MMA955xL devices need to first power down and reset to make the downloaded code effective.

To ensure they are getting the expected behavior from MMA955xL devices, users should power down the devices, then use Attach to connect to the devices to start their debug.

The Attach configuration is used to debug the MMA955xL. It enables users to connect to an already running device without having to reprogram the FLASH or reset the device.

3.1.5.1 Setup the download configuration

1. Open Debug Configuration from the menu and click on the arrow sign next to the Debug symbol.
2. Click **Debug Configurations**.

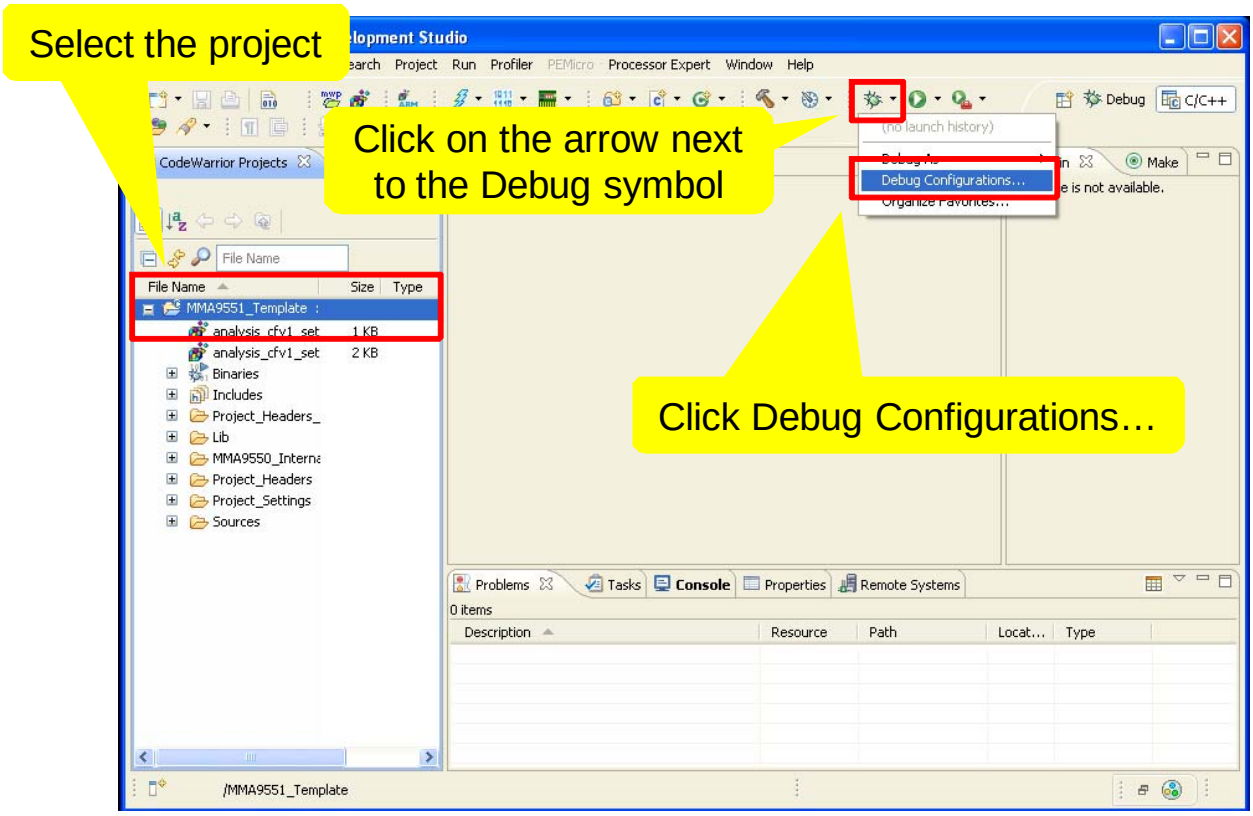

Figure 16. Open debug configuration

3. In the menu options on the left, double click **CodeWarrior Download** to create a new download section in the left window. If the download section already exists, click it to select.
4. In the top right Name field, add **_Download** to the end of the configuration name.
5. On the Main tab, to select the project name from the list, click **Browse**.
6. On the Main tab, to select the application to download to the device, click **Browse**. This file will have the "***.elf**" suffix.
7. After confirming your debugger is plugged in, on the Main tab under Remote system, click **Edit** to set System.
8. Select your debugger model from the pulldown menu and click **Apply** for the settings to take effect.

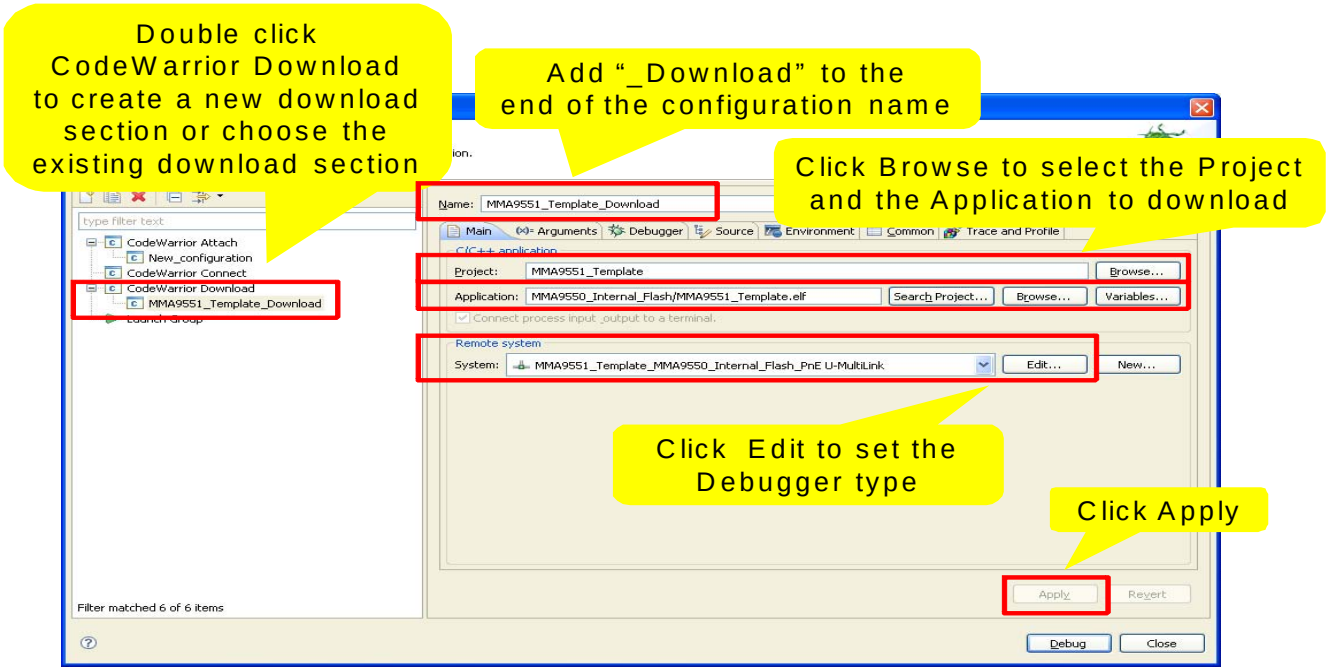

Figure 17. Set download configuration

3.1.5.2 Set the Download details for the debugger

1. Set System Type to the device the code will be downloaded to. Select **MMA9550** in the pulldown menu. The device is listed under coldfire, MCF51MMA family.

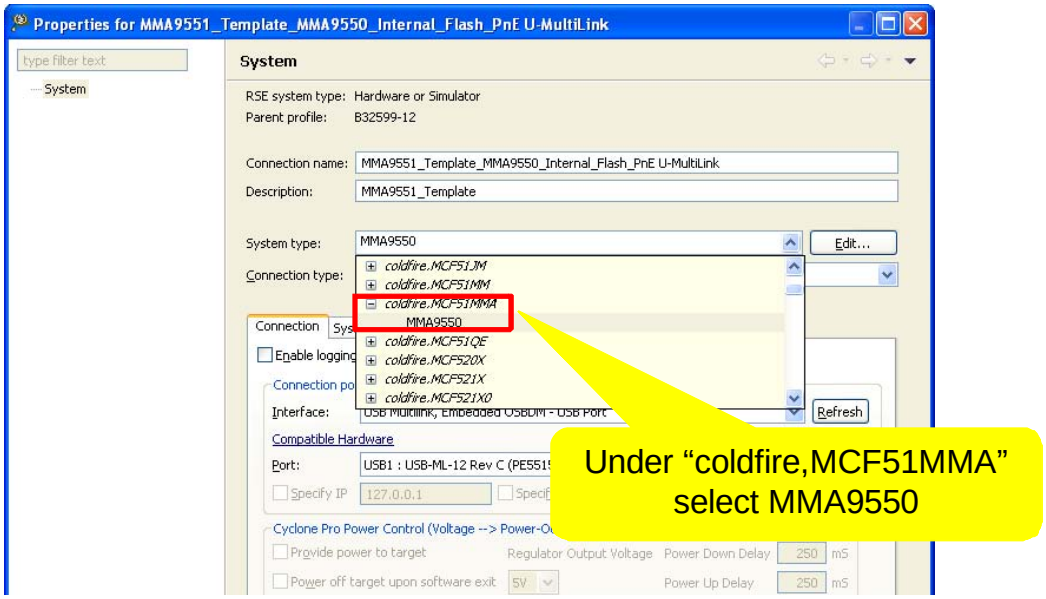

Figure 18. Set system type / device to download code to

2. Select the connection type from the pulldown menu. If your debugger is connected, CodeWarrior will suggest the connection type.
3. In the Connection tab, Interface is updated automatically. If not, click **Refresh** to manually start a debugger tool search.
4. Confirm it is the correct debugger you are using and click **OK** to continue.

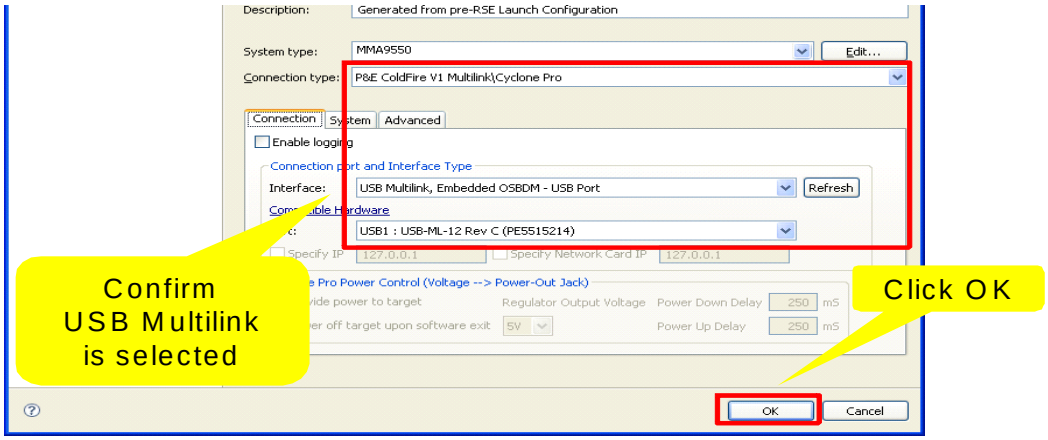

Figure 19. Set download tool

5. Select the **Debugger** tab.
6. On the Debug menu, check the box to Stop on startup at Program entry point and click **Apply**.

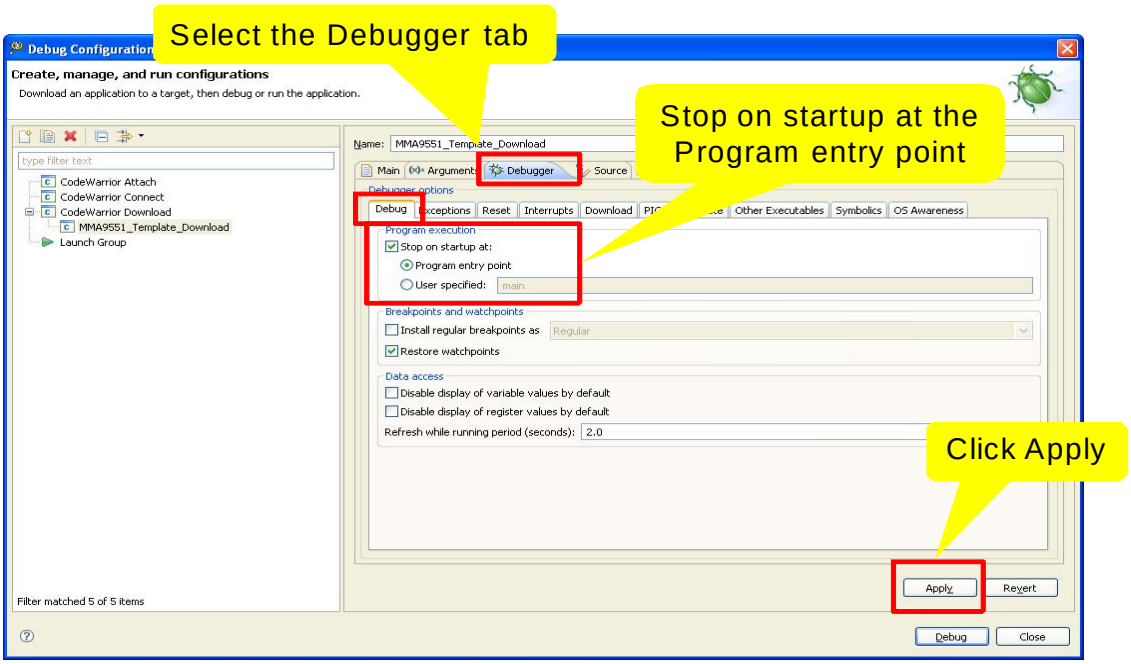

Figure 20. Set entry point function

3.1.5.3 Begin the Download

1. Connect the debugger to the MMA955xL Sensor Toolbox evaluation board using the BDM cable.
2. Connect the kit, via USB cable, into the host PC.
3. Confirm that the green LED D3 is lit. If not, make sure both ends of the USB cable are plugged in, and the switch SW1 is turned on.

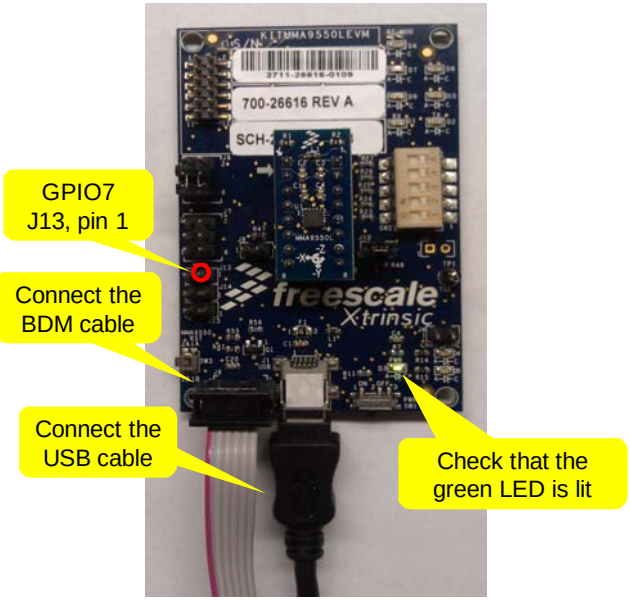

Figure 21. Connect MMA955xL Evaluation Kit and BDM to PC

4. In the CodeWarrior C/C++ perspective, choose the project to download to the device.
5. Click on the arrow next to the Debug symbol.
6. Click **Debug As > CodeWarrior Download** to start the downloading.

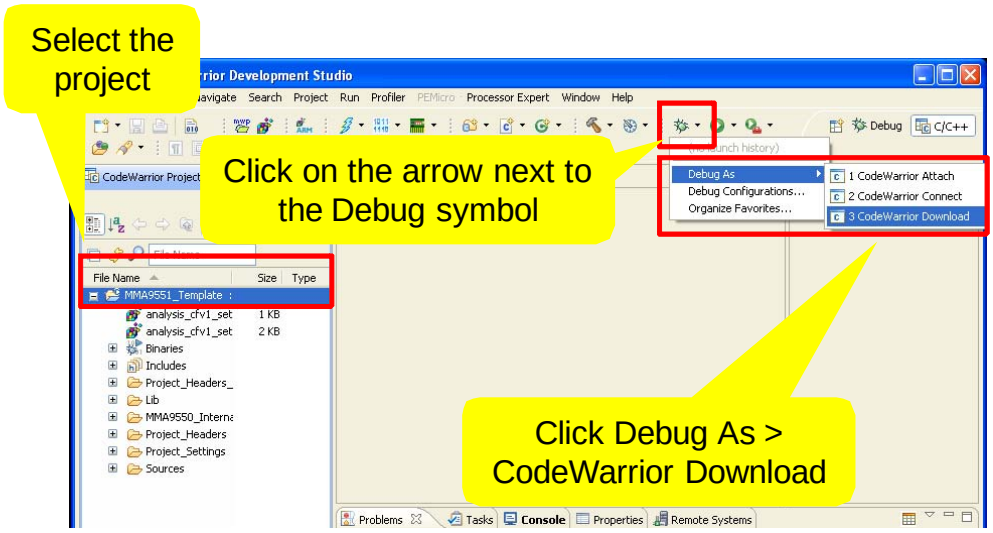

Figure 22. Start code download

7. After the code is downloaded, the default Debug perspective opens, showing multiple debug windows. Click on the red box to finish the download.

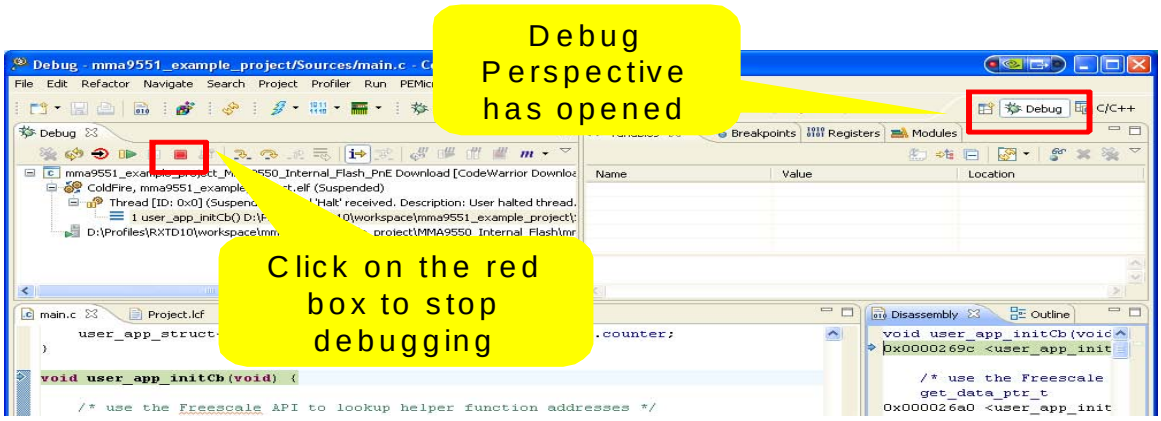

Figure 23. Download finished and stop debugging

8. Place the board flat on a table. Connect an oscilloscope on GPIO7. A square wave of 24.4 Hz signal should be observed. A digital analyzer screen capture is shown in Figure 24.

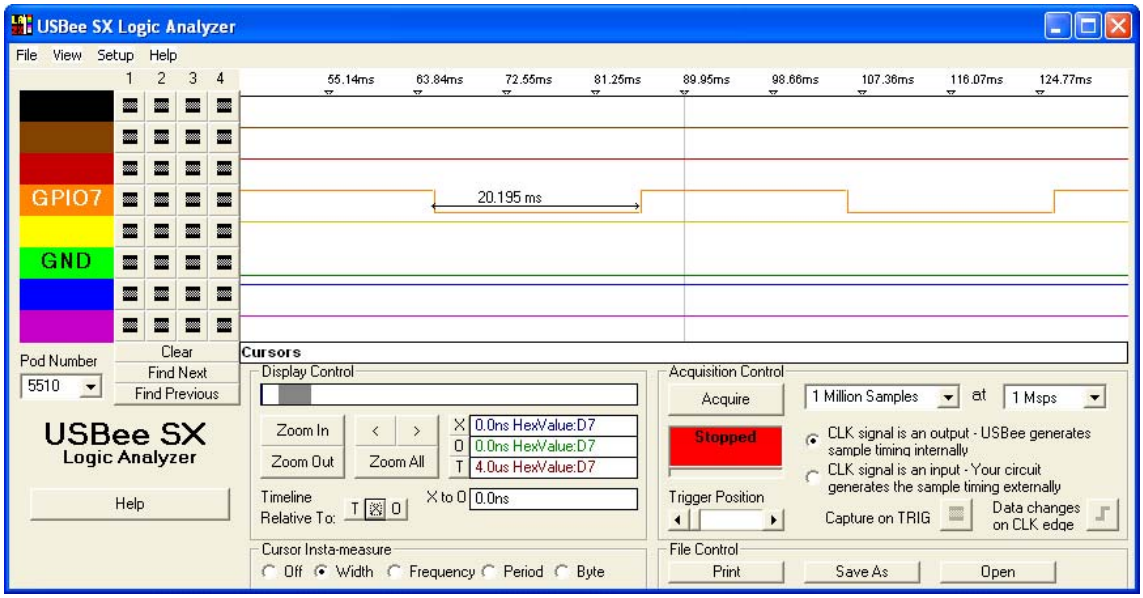

Figure 24.

3.1.5.4 Setup the Attach configuration

1. Open Debug Configuration from the menu.
2. Click on the arrow next to the Debug symbol.
3. Click **Debug Configurations**.
4. In the menu options on the left, double click **CodeWarrior Attach**, to create a new Attach section.
5. If an Attach section already exists, click on the section name to change the setting.
6. Click **Apply** for the settings to take effect.

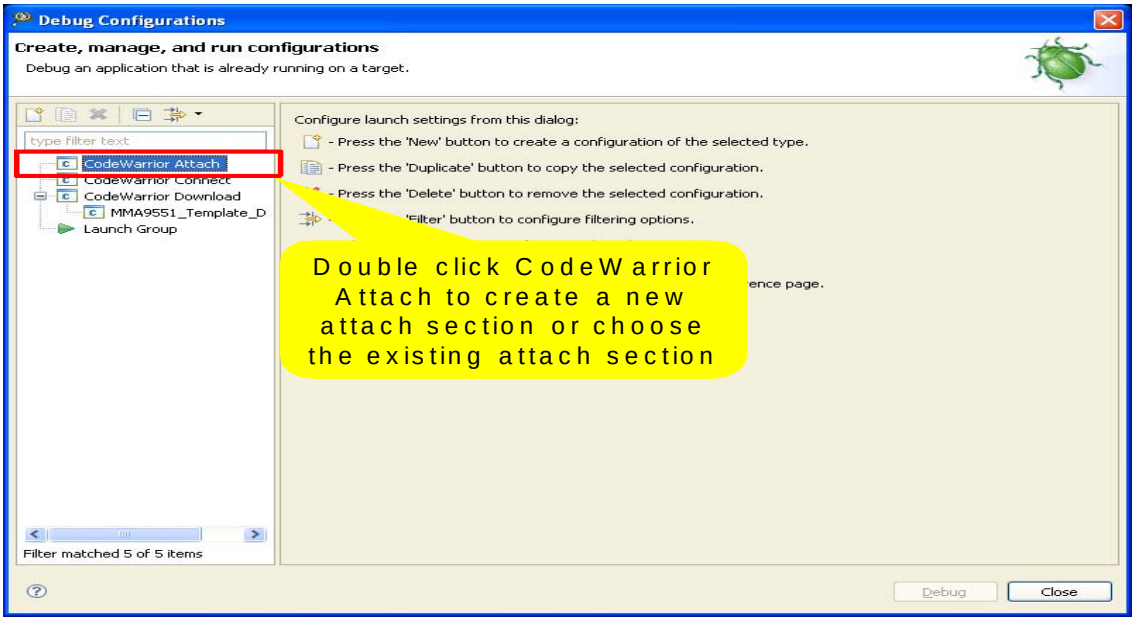

Figure 25. Create a new Attach section

7. Change the configuration name in the top right Name field.
8. Select the project according to the project name. Click **Browse** in the Main tab to access the project list.
9. Select the application to debug from the device. Click **Browse** to locate the default project. This file will have the "***.elf**" suffix.
10. Set System based on your debugger. After confirming your debugger is plugged in, on the Main tab under Remote system, click **Edit** to select the model from the pulldown menu.

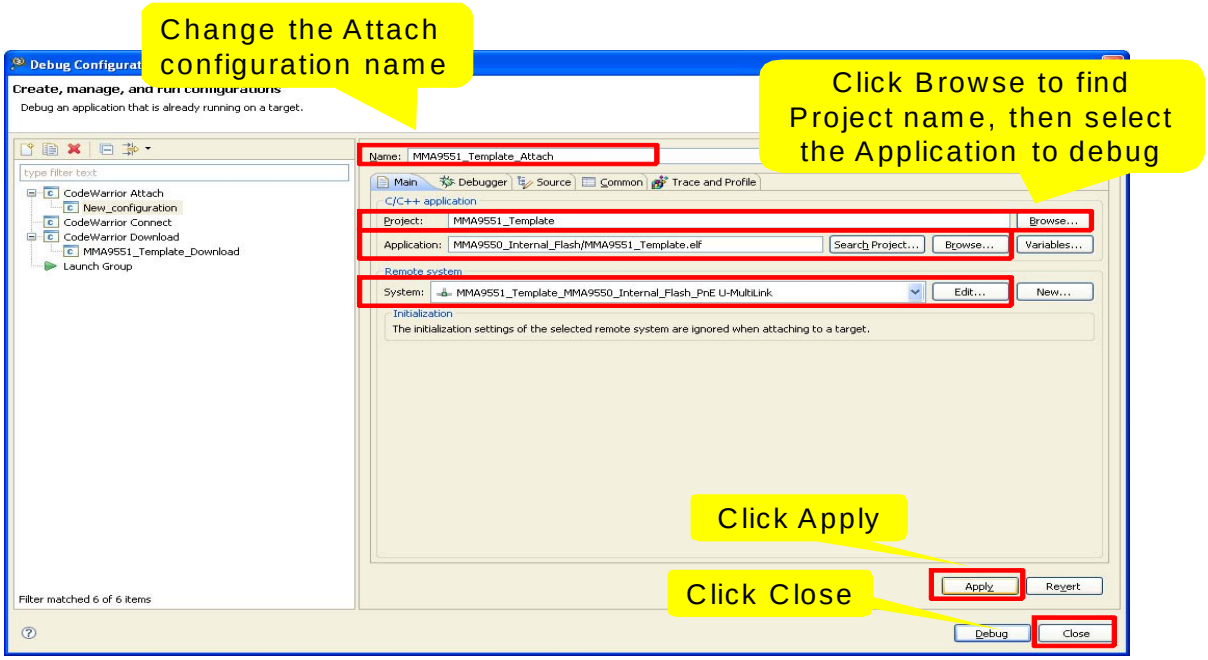

Figure 26. Set debug configuration

3.1.5.5 Set the Attach details for debugger

1. Click **Edit** next to the system pulldown menu. A window will open.
2. Set the System Type to the device that the code will be downloaded to.
3. Select **MMA9550** in the pulldown menu. The device is listed under “coldfire,MCF51MMA” family

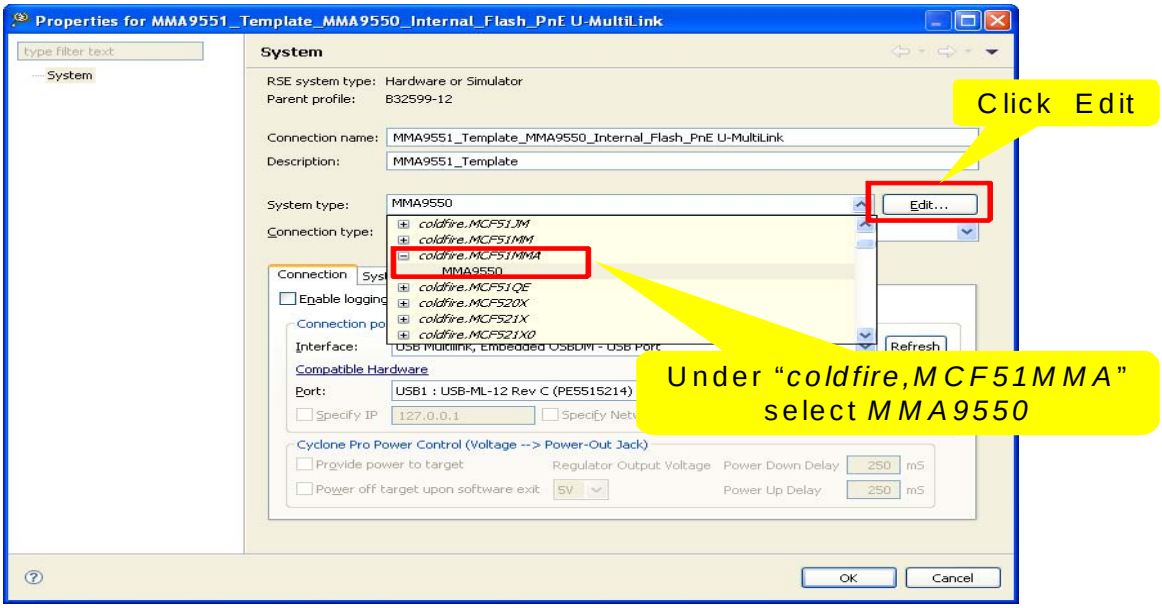

Figure 27. Set system type / device to download code to

4. Select the Connection type from the pulldown menu. If your debugger is connected, CodeWarrior will suggest the connection type.
5. In the Connection tab, Interface is updated automatically. If not, click **Refresh** to manually start a debugger tool search. Confirm it is the correct debugger you are using.
6. Click **OK** to continue.

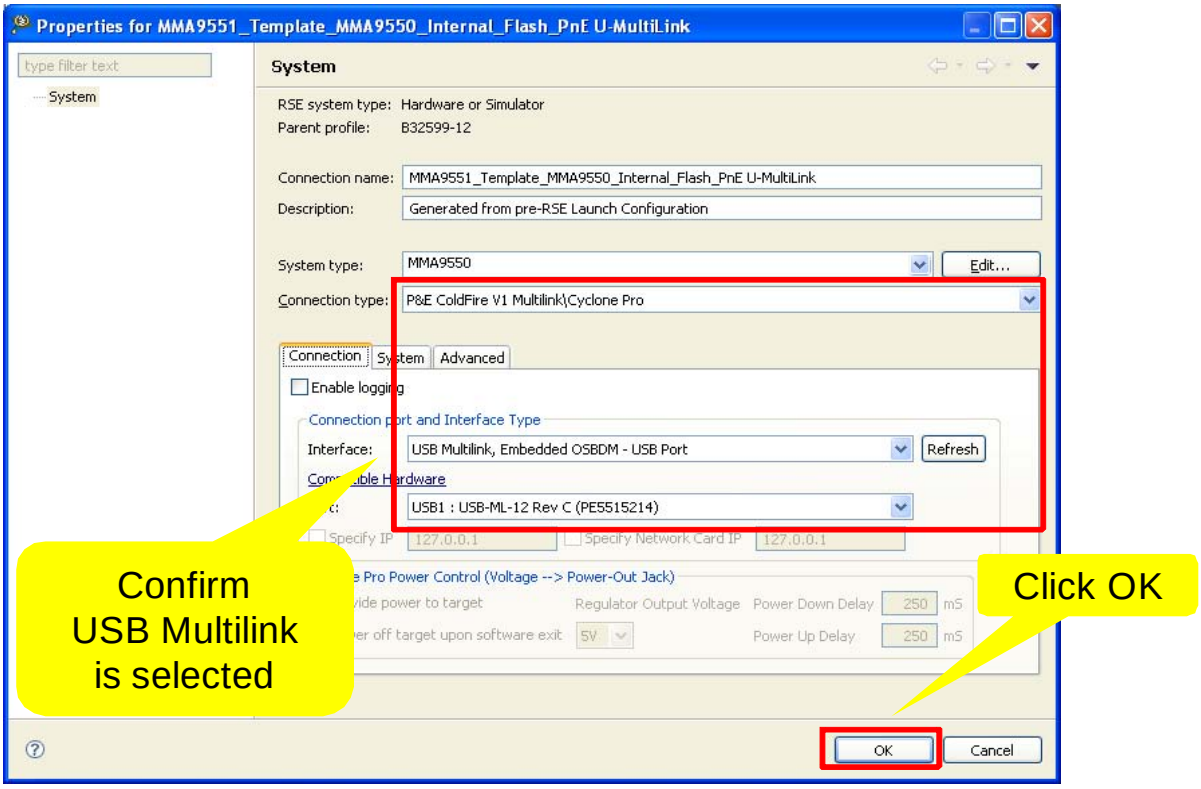

Figure 28. Set system type / device to Attach to

3.1.5.6 Begin the Attach to debug

Perform a power cycle or reset as shown in Figure 29, to allow the new MMA955xL custom firmware to link to the Freescale platform.

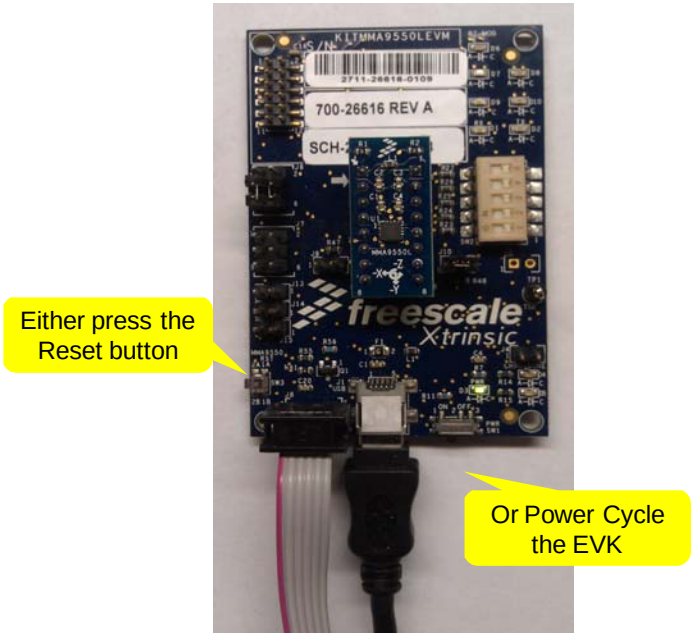

Figure 29. Perform a power cycle or reset

1. In the CodeWarrior C/C++ perspective, choose the project.
2. Click on the arrow next to the Debug symbol.
3. Click **Debug As > CodeWarrior Attach** to start the Attach.

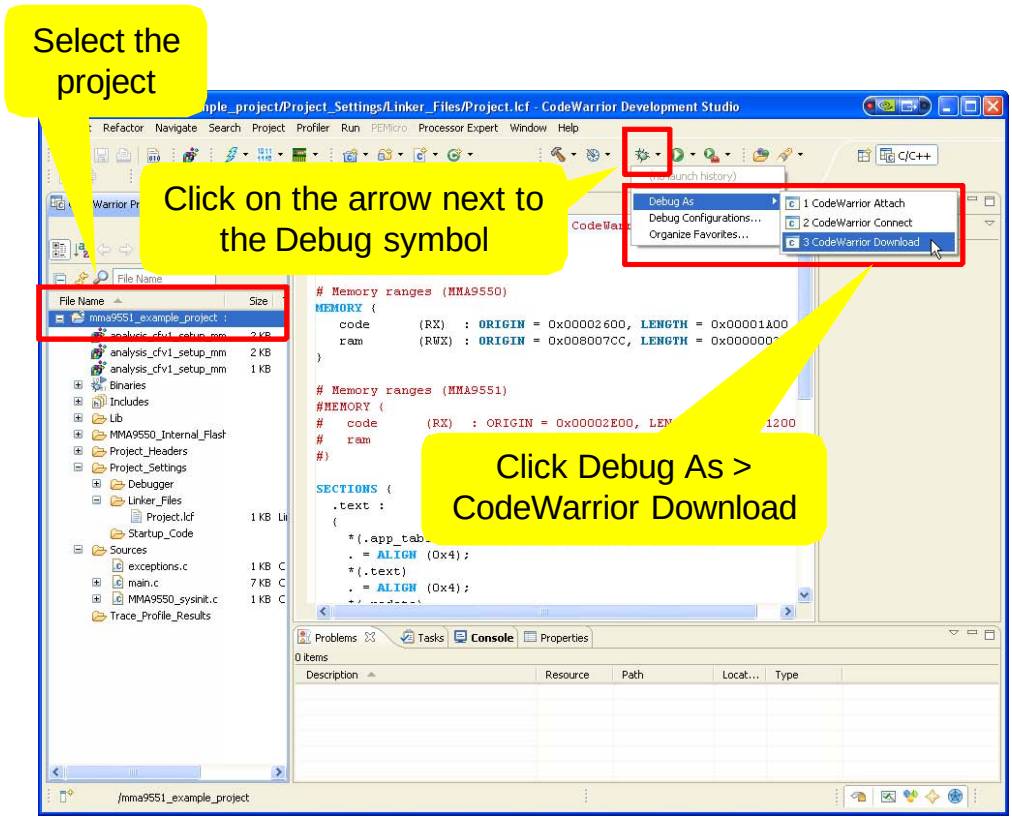

Figure 30. Start CodeWarrior Attach

4. After a successful Attach to the device, the default Debug perspective opens, showing multiple debug windows. The debug controls are outlined in red in Figure 31.

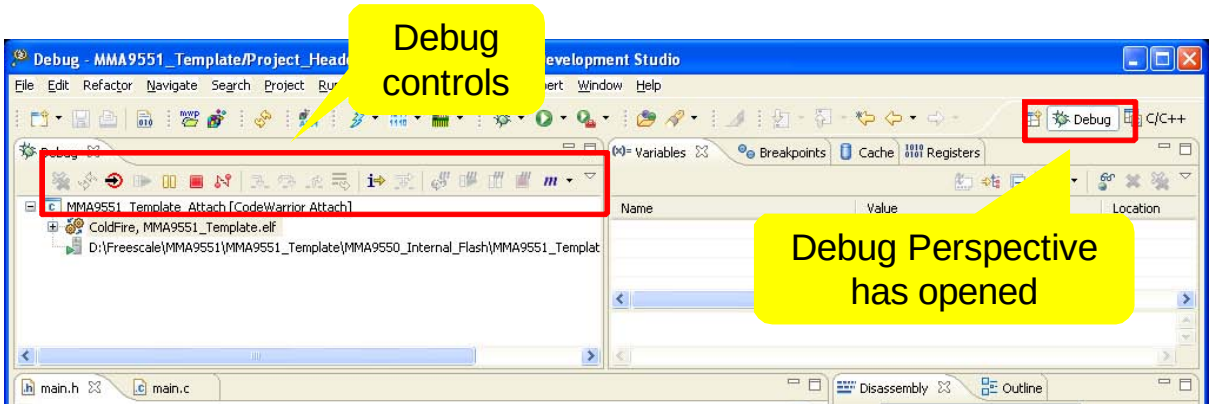

Figure 31. Attach is successful

5. In the Debug perspective, the default window displays are Debug, Variables, Code, and Disassembly.

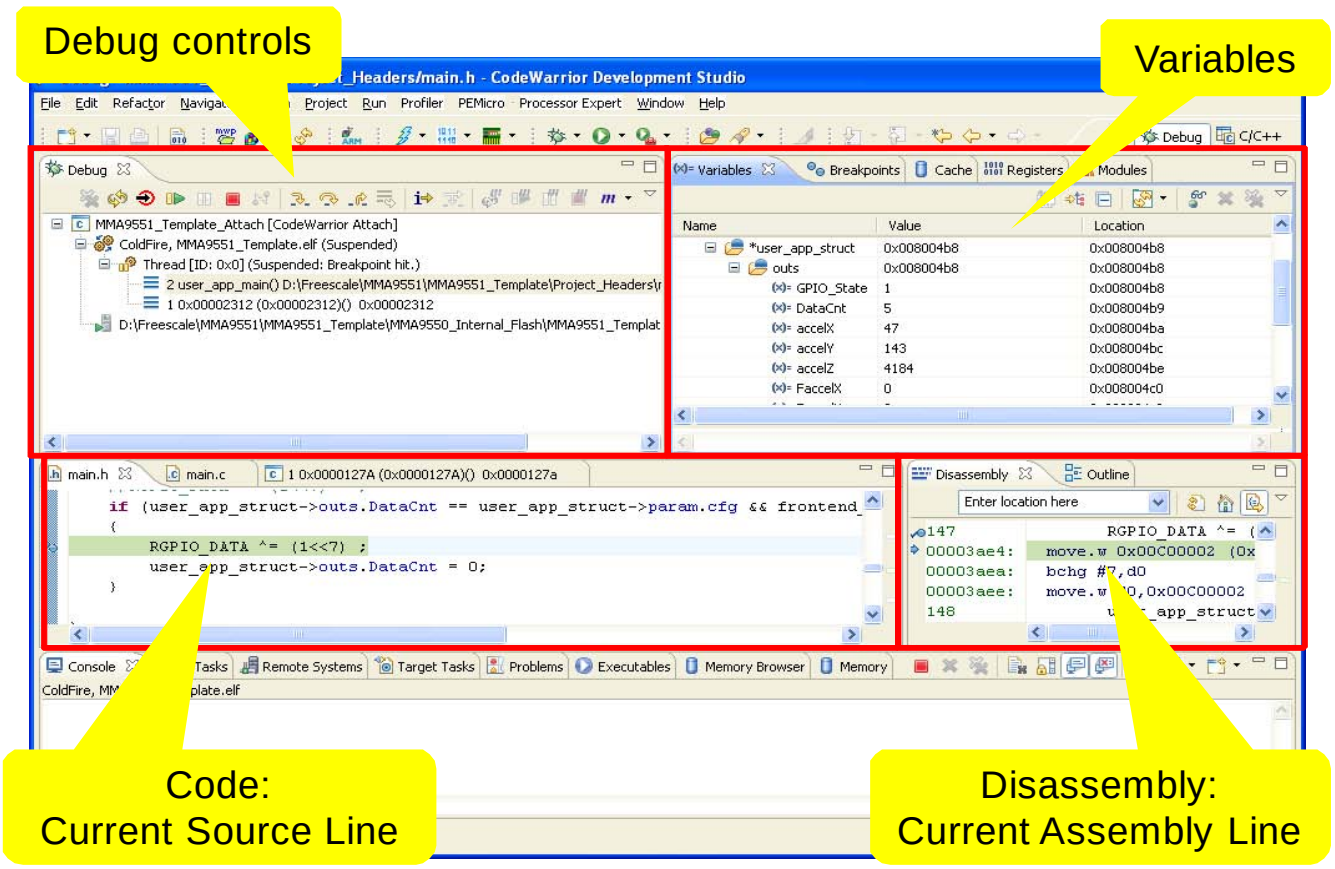

Figure 32. Debug perspective

6. There are two ways to set/revoke breakpoint in the line of code:
   a) Double click in the border next to the line of code where you want to set breakpoint. Double click again to revoke the break point.
   b) Point your mouse to the border next to the line of code where you want to set breakpoint. Right click to bring up the debug menu and select **Toggle Breakpoint** to set/revoke breakpoint.

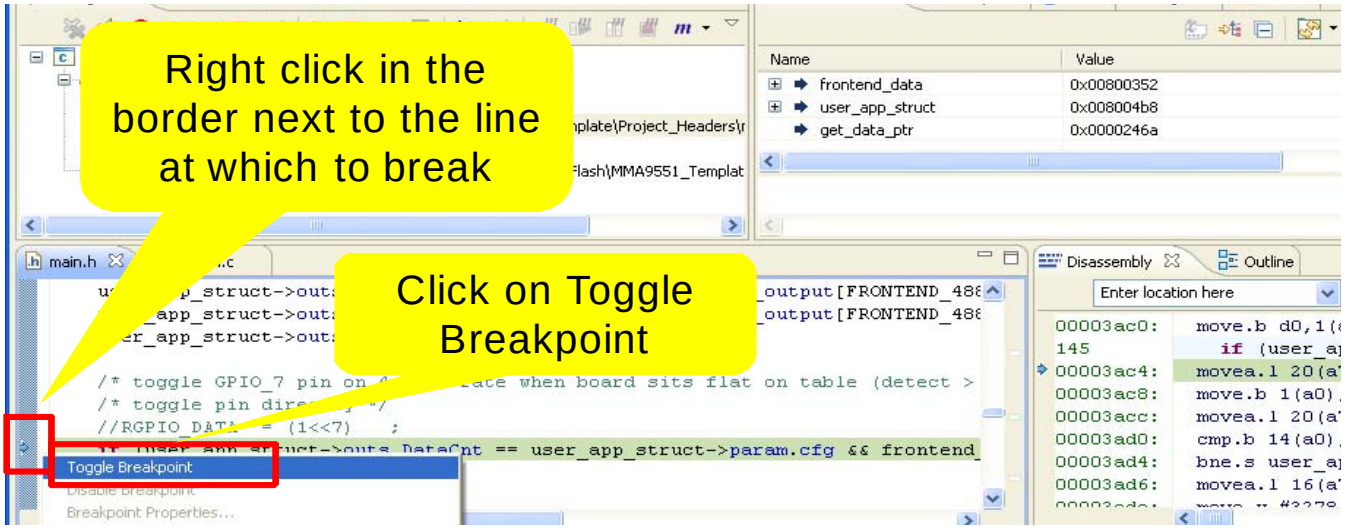

Figure 33. Set Breakpoints

7. Press **Run** to begin the program. Changed variables are highlighted in the Variables tab.

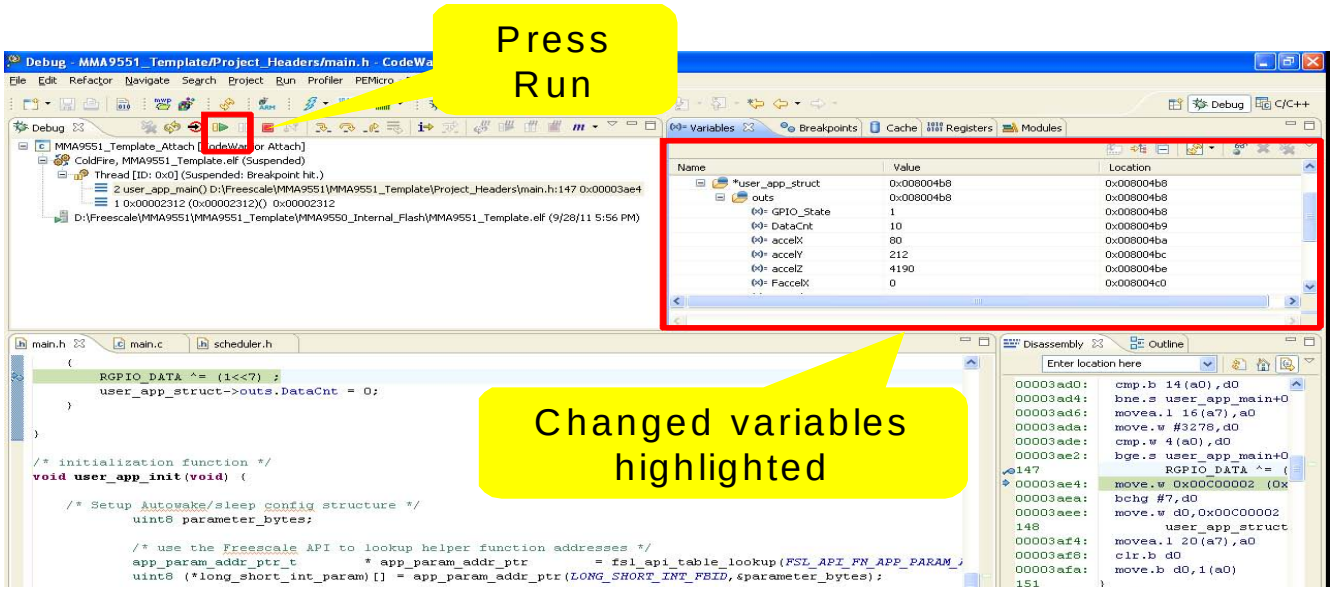

Figure 34. Run program

3.2 Sensor Toolbox Kit

Applications can be accessed using MMA955xL Sensor Toolbox Evaluation kits. This is a software tool that runs on a Windows PC, and communicates with the MMA955xL evaluation board through a USB cable. Users can access an application's configuration registers and status registers via the Slave Port Mailbox access tab. This tool enables users to interact with the applications in the MMA955xL family devices with or without the CodeWarrior IDE.

For this application note, it is assumed that users understand the basics of sensor toolbox. For details, please refer to the MMA955xL Users' Guide on the Freescale MMA955xL website.

3.2.1 Read /Write custom application data using sensor toolbox:

1. Connect the MMA955xL Sensor Toolbox Evaluation Kit to the computer via the USB port and confirm the green LED light D3 is lit. If not, make sure both ends of the USB cable are plugged in, and the switch SW1 is turned on.
2. Run the sensor toolbox software from the PC. The sensor toolbox software will recognize the device and display the Communication Interface (Comm.Interface) window on the monitor.
3. Click **Open Com** to establish the connection between the PC and MMA955xL devices.

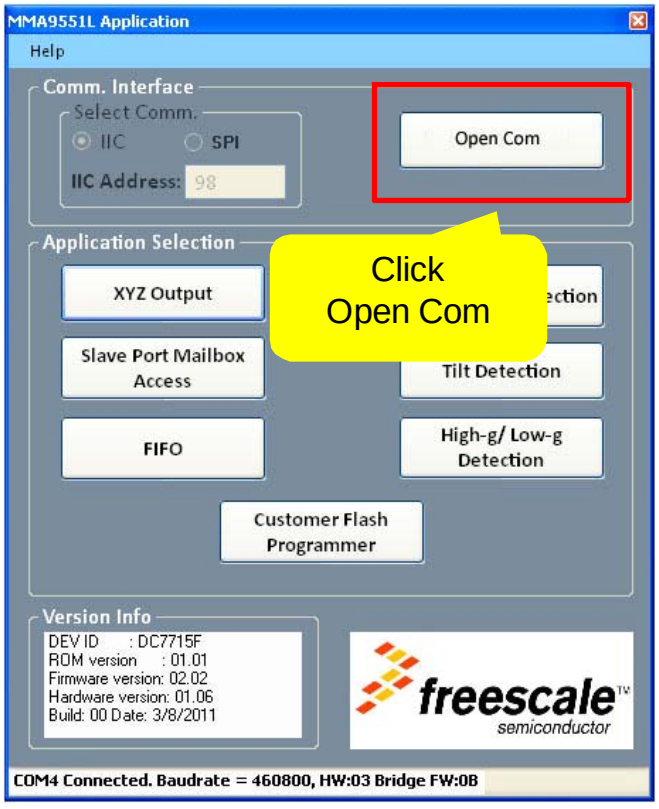

Figure 35. Establish connection between PC and MMA955xL

4. Sensor toolbox recognizes and displays specific device information in the Version Info text box. Click **Slave Port Mailbox Access** to open the Mailbox window.

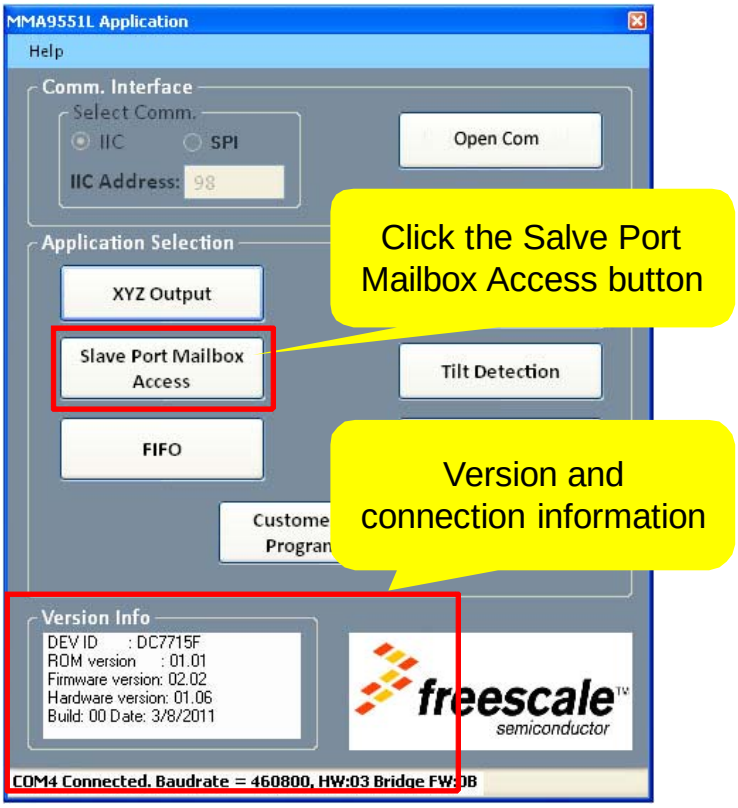

Figure 36. Establish slave port Mailbox access

Here is an example of interacting with the MMA955xL template application which has been assigned the APP_ID of 0x19. This particular application has some configuration registers and some status or output registers. The next few sections describe how to read the configuration registers, read the output registers, and write the configuration registers.

5. Reading application configuration registers:
   a) Enter the custom application **APP_ID** (FBID) in the Func ID box in hexadecimal format. In this example, the APP_ID 0x19(FBID) is decimal 25, type in 19.
   b) Enter the desired number of bytes to read in the **No. (number) of bytes to Read** in hexadecimal format. In our example we would like to read one byte of status data.
   c) Change the **Register Offset** value if needed. In our example we will start reading at the start, or zeroth, byte.
   d) Start the read command by pressing the Read Configure button.
   e) The data will show in the LOG window. The first four bytes are the response header, and the remaining bytes are data. In our example the one byte of data that we read back is 0x00.

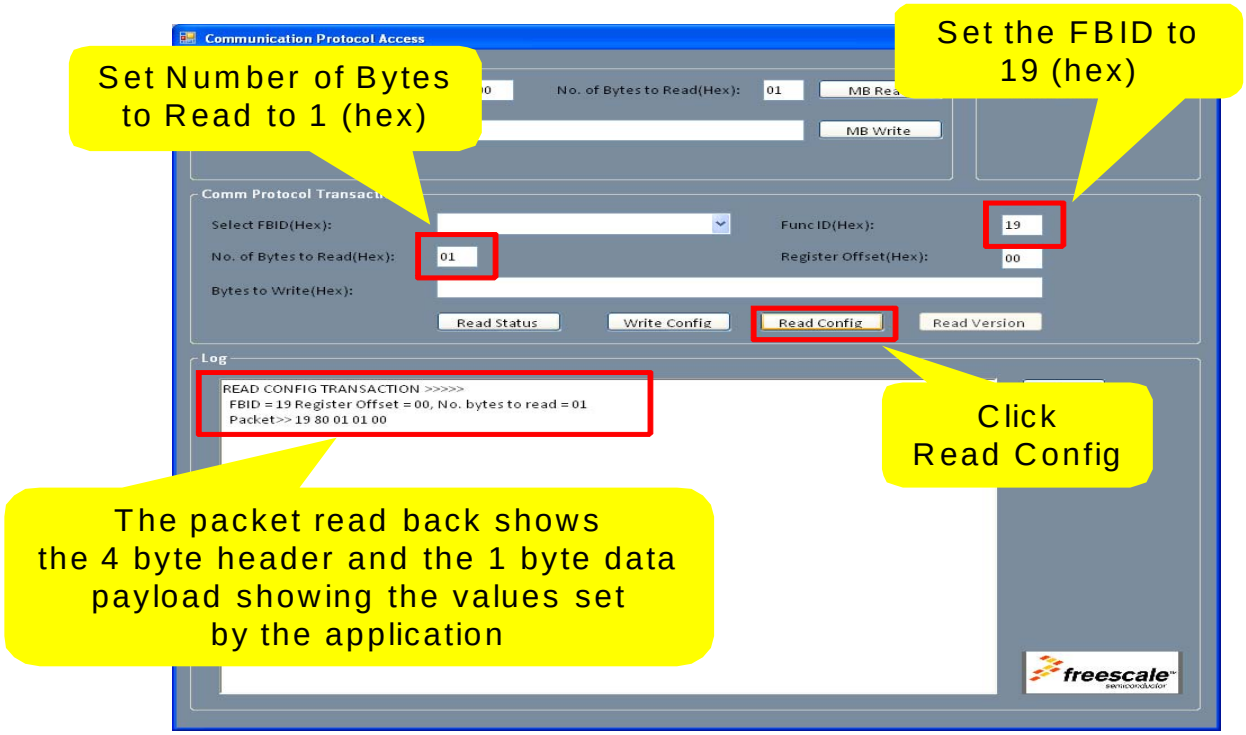

Figure 37. Read configuration from custom application

As mentioned, the response header consists four bytes;

- The APP_ID of the responding application
- The requested number of bytes
- The actual number of bytes
- The error status (0x80 means command complete, any other value indicates an error condition)

6. To read custom application output data:
   a) Enter the custom application **APP_ID (FBID)** in the Func ID box in hexadecimal format. In our example, the application APP_ID (FBID) is decimal 25, type in 19.
   b) Set the **No. (number) of bytes to Read** in hexadecimal format. For our example, we will use 14 byte, type in 0E.
   c) Change the **Register Offset** value if needed. In our example we will start reading at the start, or zeroth, byte.
   d) Start the status byte reading by clicking the **Read Status** button. The data will show in the LOG window. Each read shows the 4 byte command responses and the 14-byte status. The status is being updated by the application, so it updates on each read.

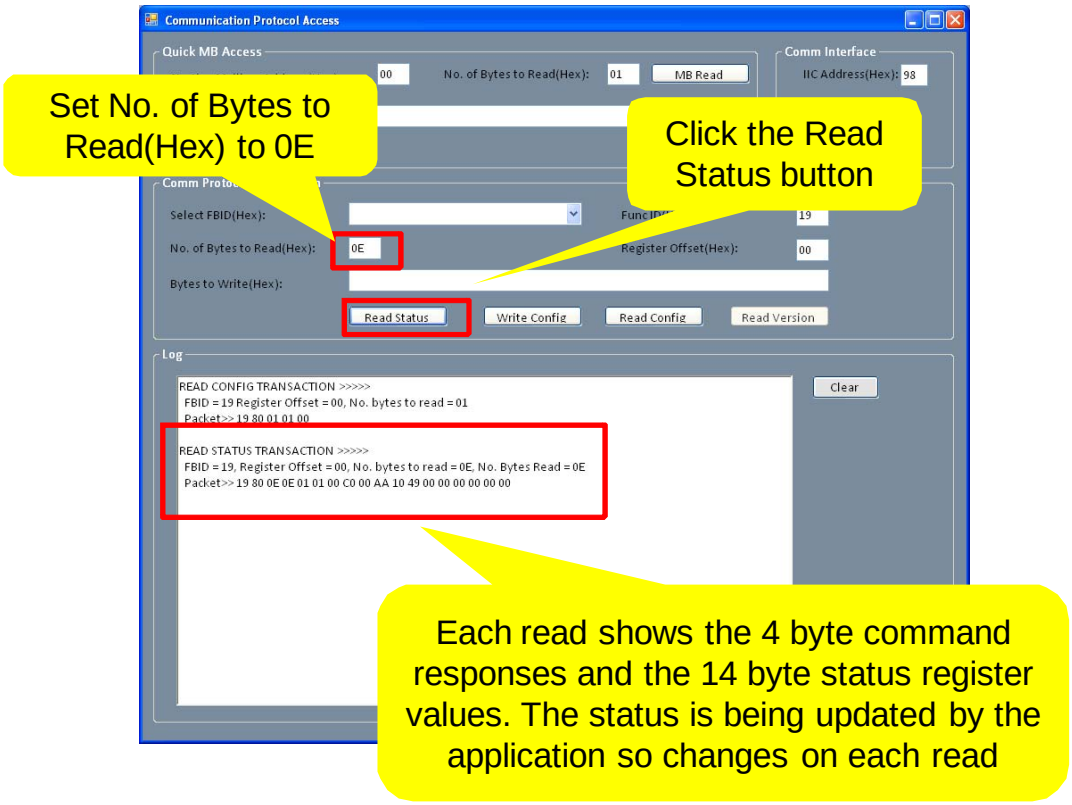

Figure 38. Read output data from custom application

For example, users can observe the application's XYZ accelerometers outputs on bytes 6-11.

7. To write custom application configuration data, repeat the previous steps.
   a) Type in the text box with the number of bytes given in hexidecimal format. In this example, the application's first configuration register byte is set to 5, to change the GPIO7 pin output to toggle at 48.8 Hz from the default 24.4 Hz. Type in 5 in the Bytes to Write(Hex) text box.
   b) Click the **Write Config** button.
   c) Click the **Read Config** button to verify the write. The data will show in the LOG window. Each read shows the 4 bytes command response and the 1 byte configuration register. The configuration remains the same on each read, unless written. In this example, the read value is 5.

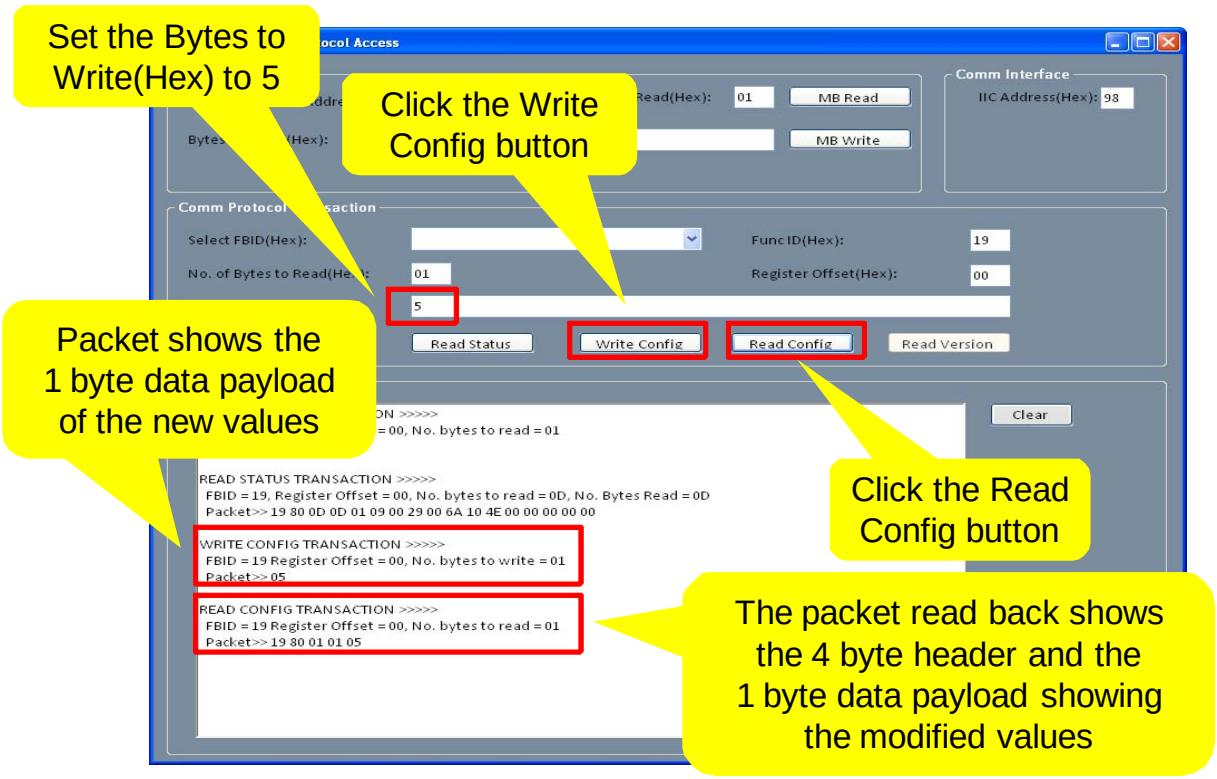

Figure 39. Write configuration from custom application

3.3 MMA955xL reference manuals

The MMA955xL Software Reference Manual is a useful tool. It outlines, in topic format, all the Freescale platform functions that can be called by the user and is available on the Freescale website within the MMA955xL family.

The MMA955xL Hardware Reference Manual is another tool. Users can find the hardware descriptions of the MMA955xL family, and gain more insight of the devices capabilities.

4 Template contents

4.1 Custom applications on the Freescale platform

As mentioned in Section 2, “Architecture”, the code for MMA955xL family devices follows a hierarchical structure. The layers, listed from bottom to top are Freescale platform, application tables, and custom applications. All custom applications adhere to this structure. The following information can be used to propagate it into multiple applications.

4.1.1 Application Table

The application table is a table composed of the table identifier (ID), the total number of applications and a combination of application maps, as show in Example 3.

The Freescale platform searches for applications tables at runtime, and will recognize them only if the application table uses ID: 0x9550C0DE. The Freescale platform considers itself one application table, and allows three other custom application tables to be bound in. For the same reason, users can stack up to three custom images in the custom FLASH.

Each CodeWarrior project should contain only one application table. For information on how to program multiple custom images into FLASH, please refer to the application notes for MMA955xL on the Freescale website.

Example 3. main.c Define application table with one application

```
__declspec(app_table) app_table_t app_table = {
    TABLE_IDENTIFIER,                          /* table identifier */
    1,                                         /* num_of_apps */
    {
        {(cbFunction)&user_app_initCb,         /* init function address */
         (cbFunction)(NULL),                   /* reset function address */
         (cbFunction)(NULL),                   /* clear function address */
         (cbFunction)&user_app_main,           /* main function address */
         (uint8_t)(USER_APP_FBID),             /* application id */
         sizeof(struct user_app_param_tag),    /* number of parameter bytes */
         sizeof(struct user_app_outs_tag),     /* number of output bytes */
        }
    }
};
```

The MMA955xL template application table resides in **main.c**. The code built around it, in the **main.c** file, tells the compiler to set the table ID 0x9550C0DE as the first four bytes of this custom image.

This table must be aligned with memory 512 byte page boundary. This is implied since the linker file sets the start FLASH location on the page boundary.

4.1.2 Application Map

The application map is the table that outlines the external interface of a custom application. It is composed of four custom application function pointers, one unique APP_ID (FBID) and two bytes, logging the size of the application parameter and outputs.

The Freescale platform uses this information to run the application, assign memory space for the application and access its variables.

Users need to pass down four function pointers to the application map:

- One function when the MMA955xL initializes
- One function to be called at the MMA955xL Reset condition
- One function to be called at the MMA955xL Clear condition
- Application main function

4.1.2.1 Application map definitions

Reset: happens when the parts are powered up, woken up from sleep and when the reset bits of the Reset/Suspend/Clear Control Application are set.

Clear: happens when the parts are powered up, woken up from sleep and when the clear bits are set where users write the Mailbox via I^2C communication.

A unique ID number per application is required. APP_IDs (FBIDs) between 0x19 and 0x1F are reserved for application ID and users can choose any number in this range.

Users can find the application map definition in **customer_apps_binding.h**, under the Project_Headers_Firmware folder. The implementation of application map is marked in the application table code shown in Example 4.

Example 4. Customer_apps_binding.h, application map definition

```
typedef struct Data_APMap_tag
{
  void (*initCbFn)(void);   /* init  callback function pointer */
  void (*rstCbFn)(void);    /* reset callback function pointer */
  void (*clrCbFn)(void);    /* clear callback function pointer */
  void (*function)(void);   /* function pointer */
  uint8_t  AP_ID;
  uint8_t  parameter_bytes;
  uint8_t  output_bytes;
}Data_APMap_t;
```

4.1.3 Program the last four bytes of FLASH

The last four bytes of the FLASH must be **0xFFFF0337** which tells the MMA955xL how to power up.

CAUTION

A common mistake engineers make is erasing the custom FLASH region, programmed in custom applications, and forgetting to program the last region which results in the devices no longer communicating with sensor toolbox.

To program these four bytes, define the FOPT register as shown in Example 5. The NVOPT-3 refers to the FOPT register location.

Example 5. main.c, Set the last four bytes to 0xFFFF0337

```
uint32 fopt @(NVOPT-3) = 0xFFFF0337;
```

4.2 Define RAM memory for custom applications

Users can request RAM space using a Freescale API. The memory allocation function has the following description:

Example 6. RAM memory allocation function

```
uint8_t * request_ram_ptr (uint16 size, uint8 u8app_id);
```

4.2.1 Data structure

Users need to provide the size of the requested memory to fix variables. These variables must be organized like the data structure code in Example 7. It is made of three parts: *custom* application output type, *custom* parameter type and *private* variable type.

The Freescale platform makes the variables in **outs_t** and **params_t** structure public, and leaves the private structure internal to the custom application. Users can set up the flags and other intermediate variables in the private section if no visibility is needed.

Users need to define the variables and organize them into the following three categories. The order of these structures must remain the same, as listed below, if using the sensor toolbox.

1. Output variables
2. Parameter variables
3. Private variables

Example 7. CustomApp1.h, data structure

```
/* user data structure. note that the outputs must be first, followed
 * immediately by the parameters without any padding in between */
typedef struct user_var_struct_tag {
    struct outs_tag {
        …
    } outs_t;
    struct param_tag {
        …
    } params_t;
    struct user_app_private_tag {
    …
    } private_t;
} user_var_struct_t;
```

4.2.2 Variable space setup

To request custom memory location, use API `request_ram_ptr ()`.

Users follow four steps to request and receive variable spaces.

a) Define a custom application data structure. Make sure that the structure is like the code below.
b) Request the API point using the fsl_api_table_lookup (`FSL_API_FN_REQ_DATA_RAM`);
c) Request memory space using this API by providing the size of RAM heap space, and the requesting memory application APP_ID (FBID).
d) Receive a pointer, pointing to this memory space.

The code in the MMA955xL template is shown in Example 8 using `user_app_struct_t` as the custom application data structure type.

Example 8. CustomApp1.c, Define memory for custom application

```
/* initialization for user application, assign RAM data space */
   /* use the Freescale API to lookup helper function addresses */
request_ram_ptr_t* request_ram_ptr = fsl_api_table_lookup(FSL_API_FN_REQ_DATA_RAM);

   /* assign RAM space */
user_app_struct_t* user_app_struct = (user_app_struct_t*) request_ram_ptr
(sizeof(user_app_struct_t),USER_APP_FBID);
```

4.3 Set the custom application run rate

The Freescale firmware platform has the schedule structure to determine the way custom applications run. This schedule structure logs the run rate (frequency), the starting point, and the function priority and its activity level threshold.

The code in Example 9 shows how users can set up this parameter in the initialization routine.

Example 9. schedule.h, Scheduler structure

```
typedef union sched_parms_tag {
    struct {
        priority_t priority     :6; // which task to execute by the scheduler
        activity_t activity     :2; // execute during high and/or low activity
    } bits;
    uint8_t Byte;
} sched_parms_t;


typedef struct scheduler_struct_tag {
    struct scheduler_outs_tag {
        uint32 timeout_status;
      //  uint8  task_count/*[MAX_TASKS]*/;
    }outs;
    struct scheduler_param_tag {
        task_status_t    waiting_to_start;
        task_status_t    user_interrupt[10];
        sched_parms_t    sched_parms[MAX_FBID];
    } param;
} scheduler_struct_t;
```

4.3.1 Run applications based on accelerometer activity

Users can set the application activity parameter to set the condition when this task is run, based on the accelerometer.

NEVER:	The custom application never runs. This is a setting to temporary disable a task.
ACTIVE:	The custom application runs when accelerometer movement is high.
INACTIVE:	The custom application runs only when accelerometer movement is low or undetectable.
ALWAYS:	The custom application runs regardless of accelerometer movement.

4.3.2 Applications run repeatedly

If the custom application is running on a regular schedule, users can set the priority parameter to run the application at a preset rate.

4.3.3 Applications run when interrupt is triggered

If the custom application is running based on interrupt trigger, users can set the interrupt table to identify the trigger resource.

4.3.4 Run schedule setup

Users can follow the steps below to set up the custom application schedule properties.

Steps a-d are implemented in the code shown in Example 10.

a) Request the API point using the fsl_api_table_lookup (`FSL_API_FN_GET_DATA_PTR`)
b) Request the data pointer using this API by providing the requesting memory application APP_ID (FBID).
c) Receive a pointer, pointing to this memory space.
d) Set the priority and activity level for the custom application.
e) Or set the interrupt source for the custom application.

Example 10. CustomApp1.c, Set the custom application run rate

```
/* Define task run frequency */

/* use the Freescale API to lookup helper function addresses */
   get_data_ptr_t * get_data_ptr = fsl_api_table_lookup(FSL_API_FN_GET_DATA_PTR);

   /* look up the address of scheduler data structures */
   scheduler_struct_t    * scheduler_struct    = get_data_ptr(SCHEDULER_FBID);

   /* configure the scheduler for priority and activity */
   scheduler_struct->param.sched_parms[USER_APP_FBID].bits.priority = TASK488HZ;
   scheduler_struct->param.sched_parms[USER_APP_FBID].bits.activity = ALWAYS;
```

4.4 Access accelerometer data

Accelerometer data acquisition is managed internally by the frontend module. The data structure of this module can be found in the header file **frontend.h**.

4.4.1 Read schedule setup

Users can follow the steps below to set up the custom application schedule properties.

Step a-d are implemented in the code shown in Example 11.

a) Request the API point using the fsl_api_table_lookup (`FSL_API_FN_GET_DATA_PTR`)
b) Request the data pointer using this API by providing the Frontend module ID (`XYZ_DATA_FBID`).
c) Receive a pointer, pointing to application memory space.
d) Read the 3-axis data from output structure by using the pointer reference.
e) Configure the accelerometer by writing to the prams variables.

Example 11. CustomApp1.c, access accelerometer data

```
/* use the Freescale API to lookup helper function addresses */
get_data_ptr_t* get_data_ptr = fsl_api_table_lookup(FSL_API_FN_GET_DATA_PTR);

/* Look up the address of RAM data structures */
user_app_struct_t  * user_app_struct = get_data_ptr(USER_APP_FBID);
frontend_data_t * frontend_data = get_data_ptr(XYZ_DATA_FBID);

/* acquire accelerometer data and copy to user memory location */
user_app_struct->outs.accelX = frontend_data>outs.accel_output[FRONTEND_488_100][FRONTEND_X];
user_app_struct->outs.accelY = frontend_data>outs.accel_output[FRONTEND_488_100][FRONTEND_Y];
user_app_struct->outs.accelZ = frontend_data>outs.accel_output[FRONTEND_488_100][FRONTEND_Z];
```

4.5 Gesture functions

Gesture functions are available for MMA9551L and include:

- portrait-landscape
- high-g/low-g
- tap
- tilt

The header file, **gestures.h**, includes the data structure for these functions. The basic steps to call user gesture functions are the same as those used to call other functions.

4.5.1 Gesture function setup

Users can follow the steps below to set up custom application schedule properties.

Steps a-d are implemented in the code examples that follow.

a) Request the API pointer using the fsl_api_table_lookup (`FSL_API_FN_GET_DATA_PTR`)
b) Request the data pointer using this API by providing the gesture application APP_ID (FBID).
c) Receive a pointer, pointing to application memory space.
d) Read output structure variables to get the function status.
e) Write to prams variable to configure the gesture function.

4.5.2 Combine gesture flag for more custom applications

Users have access to individual gesture function status bits, which makes it possible to mix different gesture status together and make decisions based on the combination. This broadens the user's options for use-case implementation. The following application examples show how the mixing of gestures can help.

4.5.2.1 Application 1

In the first application example, the Z-lock (tilt) status of the MMA9551L device, is monitored with an add-in debounce feature. This debounce feature is now available because of the MMA9551L's quick flag access and programming features. If the device is kept at less than a 30 degree tilt on Z-axis from horizon, the output of GPIO7 would toggle every 10 scheduling cycles. This application can be used to ensure that the object remains relatively flat to the table.

Example 12. CustomApp1.c, application 1 Code

```
/* use the Freescale API to lookup helper function addresses */
get_data_ptr_t* get_data_ptr = fsl_api_table_lookup(FSL_API_FN_GET_DATA_PTR);
/* Look up the address of RAM data structures */
pl_struct_t* pl_struct= get_data_ptr(PORTRAIT_LANDSCAPE_FBID);

/* acquire PL Z-lock status, and debouce on status Zlock, and Zlcok & LAPO */
if (pl_struct->outs.bits.ZTilt_Angle_Lockout == 1)
{
   // log event, if Zlock is detected, increase ZlockCnt
   user_app_struct->outs.ZlockCnt += 1;
}
else
{
   user_app_struct->outs.ZlockCnt = 0;
}

/* process GPIO_7 on based on debounce value */
if (user_app_struct->outs.ZlockCnt == user_app_struct->param.event_cnt)
{
    RGPIO_DATA ^= (1<<7);
    user_app_struct->outs.ZlockCnt = 0;
}
```

4.5.2.2 Application 2

This second application, shown in Example 13, is to provide portrait-landscape orientation when the board sits at a less tilted angle. Its output toggles when the device is standing at portrait-up position AND the tilt angle of the device (Z-axis) is less than 30 degree for more than 10 schedule cycles. This angle is set in Z-lock threshold.

Example 13. CustomApp1.c, application 2 Code

```
/* use the Freescale API to lookup helper function addresses */
get_data_ptr_t* get_data_ptr = fsl_api_table_lookup(FSL_API_FN_GET_DATA_PTR);
/* Look up the address of RAM data structures */
pl_struct_t* pl_struct= get_data_ptr(PORTRAIT_LANDSCAPE_FBID);

/* acquire PL Z-lock status, and debouce on status Zlock, and Zlcok & LAPO */
if (pl_struct->outs.bits.ZTilt_Angle_Lockout == 1)
{
    /* acquire PL Z-lock and orientation status, and process log event
     * if orientation = portrait up && Zlock is detected,
     * increase PLmixCnt
     * else clear PLmixCnt
```

```
    */
    if (pl_struct->outs.bits.LAPO == 0x01)
        user_app_struct->outs.PLmixCnt += 1;
    else
        user_app_struct->outs.PLmixCnt = 0;
}

/* process GPIO_7 based on debounce value*/
if (user_app_struct->outs.PLmixCnt == user_app_struct->param.event_cnt)
{
    RGPIO_DATA ^= (1<<7);
    user_app_struct->outs.PLmixCnt = 0;
}
```

4.5.2.3 Application 3

Users can also combine the two previous examples of code, as shown in Example 14, to reduce the code space and increase code efficiency.

Example 14. CustomApp1.c, access gesture functions

```
/* use the Freescale API to lookup helper function addresses */
get_data_ptr_t* get_data_ptr = fsl_api_table_lookup(FSL_API_FN_GET_DATA_PTR);

/* Look up the address of RAM data structures */
pl_struct_t* pl_struct= get_data_ptr(PORTRAIT_LANDSCAPE_FBID);
/* acquire PL Z-lock status, and debouce on status Zlock, and Zlcok & LAPO */
if (pl_struct->outs.bits.ZTilt_Angle_Lockout == 1)
{
   user_app_struct->outs.ZlockCnt += 1;
   /* acquire PL Z-lock and orientation status, and process
    *    log event, if orientation = portrait up && Zlock is not detected,
    *    increase PLmixCnt
    *    else clear PLmixCnt
    */
   if (pl_struct->outs.bits.LAPO == 0x01)
      user_app_struct->outs.PLmixCnt += 1;
   else
      user_app_struct->outs.PLmixCnt = 0;
}
else
{
   user_app_struct->outs.ZlockCnt = 0;
}

/* process GPIO_7 on cfg
* if cfg = Data, GPIO7 is toggled when boards sits on table > 0.8g and DataCnt reach event_cnt
* if cfg = Zlock, GPIO7 is toggled when board is tilted from horizon < 30 degree for 10 frame
*    cycle
* if cfg = PLCnt, GPIO is toggled when board does not sits on table < 30 degree for more then
*    event_cnt counts
*/
switch (user_app_struct->param.cfg)
{
   case Data:
```

```
        if (user_app_struct->outs.DataCnt == user_app_struct->param.event_cnt &&
(user_app_struct->outs.accelZ > 0x0CCE))
        {
           RGPIO_DATA ^= (1<<7);
           user_app_struct->outs.DataCnt = 0;
        }
        break;
   case Zlock:
        if (user_app_struct->outs.ZlockCnt == user_app_struct->param.event_cnt)
           {
           RGPIO_DATA ^= (1<<7);
           user_app_struct->outs.ZlockCnt = 0;
        }
        break;
   case PLmix:
        if (user_app_struct->outs.PLmixCnt == user_app_struct->param.event_cnt)
        {
           RGPIO_DATA ^= (1<<7);
           user_app_struct->outs.PLmixCnt = 0;
        }
        break;
   default:
        break;
}
```

4.6 Stream data to FIFO

Users can direct data from application functions and buffer in DATA FIFO. This is a great way to view the device historical data.

4.6.1 Configure the FIFO attributes

Users should follow the steps below to set up custom application DATA FIFO attributes. Refer to the MMA955xL Software Reference Manual to understand the parameter variables then configure the params variable values.

Steps a-f, excluding step e, are implemented in the code below.

a) Request the API point using the fsl_api_table_lookup (`FSL_API_FN_GET_DATA_PTR`)
b) Request the data pointer using this API by providing the DATA FIFO ID (`FIFO_FBID`).
c) Receive a pointer, pointing to application memory space.
d) Pick one function block as FIFO data source, byte number to capture for each record from this function block, decide the FIFO size (see below) and its run mode, and where to set the FIFO Watermark.
e) Read params to confirm the configuration.

Example 15. CustomApp1.c, Configure DATA FIFO

```
/* initialization for DATA FIFO functions */

   /* use the Freescale API to lookup helper function addresses */
```

```
//get_data_ptr_t * get_data_ptr = fsl_api_table_lookup(FSL_API_FN_GET_DATA_PTR);

/* look up the address of DATA FIFO structures */
dataFifo_struct_t*dataFifo_struct_ptr= get_data_ptr(FIFO_FBID);

/* set up configures for FIFO */
dataFifo_struct_ptr->param.config.freeRun.cfg.Byte= 0x0C;// FreeRun mode, 6 bytes
dataFifo_struct_ptr->param.sz = 0x0060;// FIFO size = 96 bytes
dataFifo_struct_ptr->param.ch1_app_id = XYZ_DATA_FBID;// Route in Frontend data
dataFifo_struct_ptr->param.wmrk = 0x0030;// Watermark at 48 bytes
```

4.6.1.1 Decide the FIFO size

DATA FIFO size can be decided using the formula shown in Equation 1:

DATA FIFO size = Overhead + Record (bytes) = 6 + (number of records) x (1 + data byte per record) (bytes) ***Eqn. 1***

The DATA FIFO resides in the RAM area. The maximum size of the FIFO is also dependent on the memory usage of the rest of the program.

CAUTION:

Because this memory is assigned in runtime, FIFO space might corrupt other RAM space if the total RAM allocation request is over the available memory size. It is recommended to double check if the FIFO size is realistic.

4.6.2 Steps to read DATA FIFO status

Follow the steps below to read status bytes.

a) Request the API point using the fsl_api_table_lookup (`FSL_API_FN_GET_DATA_PTR`)
b) Request the data pointer using this API by providing the DATA FIFO ID (`FIFO_FBID`).
c) Receive a pointer, pointing to application memory space.
d) Read the first four bytes of output registers.

4.6.3 Read the DATA FIFO record

The first byte of each record is the application ID, and the rest of the record is the data byte. The length of the record depends on the data payload size. Records saved in the FIFO start at offset 0x06. These first six bytes are comprised of four bytes status registers and two bytes time stamp.

In the example below, the payload of each record is six bytes. The DATA FIFO buffered records are read out in the Mailbox interaction with the following steps:

a) Send DATA FIFO APP_ID (FBID)
b) Send DATA FIFO Read status register byte 0x30
c) Send Offset byte
d) Send Read byte number

e) Read all returned output register values with fixed offset 0x00, extract record data by going to the 7th byte.

f) Read two bytes time stamp with fixed offset 0x00.

Example 16. Mailbox command line, Read DATA FIFO records

```
/* Read 20 bytes of DATA FIFO */
// 0F: FIFO_FBID
// 30: read output/status register
// 00: note here the offset should always be below 4
// 20: read 20 bytes from DATA FIFO
Send to Mailbox 0 write command: 0F 30 00 20
Send to Mailbox 0 Read command
```

NOTE

The offset must be set to 0, 1, 2, or 3, otherwise a command error will be asserted. The total byte to read needs to be the integer multiples of the record byte number plus the six bytes overhead to ensure the complete record is read. However, reading partial record will not cause command errors. Users need to reset the DATA FIFO module after programming to have read-record access.

The DATA FIFO is only designed to be read by the I^2C/SPI host via the Mailbox, and cannot be read by the applications in the code area.

4.7 Stream events to FIFO

Like using the DATA FIFO to store the data from applications, users can also use EVENT FIFO to log events occurring from applications.

4.7.1 Configure the FIFO attributes

Users should follow the steps below to set up the custom application's DATA FIFO attributes. Refer to the MMA955xL Software Reference Manual to understand the parameter variables. then configure the params variable values.

Steps a-f, excluding step e, are implemented in the code below.

a) Request the API point using the fsl_api_table_lookup (`FSL_API_FN_GET_DATA_PTR`)
b) Request the data pointer using this API by providing the EVENT FIFO ID (`EVENT_QUEUE_FBID`).
c) Receive a pointer, pointing to application memory space.
d) Decide the FIFO size (see below), where to set the FIFO Watermark at and the maximum time without READ before the timeout flag is raised.
e) Read params to confirm the configuration.

Example 17. CustomApp1.c, Configure EVENT FIFO

```
/* initialization for event FIFO functions */

/* use the Freescale API to lookup helper function addresses */
//get_data_ptr_t * get_data_ptr = fsl_api_table_lookup(FSL_API_FN_GET_DATA_PTR);

/* look up the address of data FIFO structures */
```

```
evntFifo_struct_t*eventFifo_struct= get_data_ptr(EVENT_QUEUE_FBID);

/* set up configures for FIFO */
eventFifo_struct->param.sz = 0x0060; // FIFO size = 96 bytes
eventFifo_struct->param.wmrk = 0x0040; // Watermark = 64 bytes
eventFifo_struct->param.time_out = 0x0100;// Timeout = 256 Sample cycles
```

4.7.1.1 Decide the EVENT FIFO size

EVENT FIFO size can be decided using the formula shown in Equation 2:

EVENT FIFO size = Overhead + Record (bytes) = 4 + ((number of records) x 6) (bytes) ***Eqn. 2***

Following this formula gives users highest code efficiency. Partial record is allowed in EVENT FIFO, but is not recommended because the memory space is not utilized efficiently.

The EVENT FIFO resides in the RAM area. The maximum size of the FIFO is also dependent on the memory usage of the rest of the program.

CAUTION

Because this memory is assigned in runtime, FIFO space might corrupt other RAM space if the total RAM allocation request is over the available memory size. It is recommended to double check if the FIFO size is realistic.

4.7.2 Read the EVENT FIFO status

Follow the steps below to read status bytes.

a) Request the API point using the fsl_api_table_lookup (`FSL_API_FN_GET_DATA_PTR`)
b) Request the data pointer using this API by providing the EVENT FIFO ID (`EVENT_QUEUE_FBID`).
c) Receive a pointer, pointing to application memory space.
d) Read the first four bytes of output registers.

4.7.3 Read the EVENT FIFO data

The EVENT FIFO contents start with four bytes of EVENT FIFO status, followed by the records contents. Each record is six bytes long. The first two bytes are the Frame Counter (indicator of time), followed by the application's ID, and the remaining three bytes are payload. The payload is the result of bit AND operation of the application's status register and the event queue mask.

The EVENT FIFO logs events are read out in the Mailbox interaction with the following steps.

a) Send EVENT FIFO APP_ID (FBID)
b) Send EVENT FIFO Read status register byte 0x30
c) Send Offset byte 0x00
d) Send Read byte number
e) Read all output registers with fixed offset 0x00, extract record data by going to the 5th byte.

Example 18. Mailbox command line, read EVENT FIFO records

```
/* Read 20 bytes of EVENT FIFO */
// 0F: FIFO_FBID
// 30: read output/status register
// 00: note here the offset should always be below 4
// 20: read 20 bytes from EVENT FIFO
Send to Mailbox 0 write command: 10 30 00 20
Send to Mailbox 0 Read command
```

The event FIFO is only intended to be read by the I^2C/SPI host via the Mailbox, and cannot be read by the applications in the code area.

NOTE

The offset must be set to 0, 1, 2, or 3, otherwise a command error will be asserted. To ensure the complete record is read, the total byte to read needs to be the integer multiples of the record byte number plus the four bytes overhead. However, reading partial record will not cause command errors. Users need to reset the EVENT FIFO module after programming, to have read-record access.

5 Summary

This document covered how to build a custom applications project using the MMA955xL template. The template and other tools and documents, are available on the Freescale website at http://www.freescale.com/webapp/sps/site/prod_summary.jsp?code=MMA9550L.

The topics covered include, the high-level firmware architecture, defining custom applications, requesting memory space for variables, setting up the run task schedule, accessing accelerometer data, using gesture features, logging data and events in the FIFO and utilizing available application development tools.

Referencing and applying this information can enhance the user experience when building custom applications.

Related Documentation

The MMA9550L/MMA9551L device features and operations are described in a variety of reference manuals, user guides, and application notes. To find the most-current versions of these documents:

1. Go to the Freescale homepage at:

   http://www.freescale.com/

2. In the Keyword search box at the top of the page, enter the device number MMA9550L/MMA9551L.
3. In the Refine Your Result pane on the left, click on the Documentation link.

How to Reach Us:

Home Page:
www.freescale.com

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

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

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

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

Asia/Pacific:
Freescale Semiconductor China Ltd.
Exchange Building 23F
No. 118 Jianguo Road
Chaoyang District
Beijing 100022
China
+86 10 5879 8000
support.asia@freescale.com

For Literature Requests Only:
Freescale Semiconductor Literature Distribution Center
1-800-441-2447 or +1-303-675-2140
Fax: +1-303-675-2150
LDCForFreescaleSemiconductor@hibbertgroup.com

AN4129
Rev. 0
01/2012