

MetaWatch Firmware Design Guide



1 Contents

| | | |
|-------|--|----|
| 1 | Contents | 2 |
| 2 | Introduction | 3 |
| 2.1 | Revision History | 4 |
| 3 | Hardware | 5 |
| 3.1 | Common Watch Features | 5 |
| 3.2 | Analog Watch | 5 |
| 3.3 | Digital Watch | 5 |
| 3.4 | Bluetooth Hardware Interface | 5 |
| 4 | Software Architecture | 5 |
| 4.1 | Bluetooth Software Interface | 5 |
| 4.2 | FreeRTOS | 6 |
| 4.3 | Tasks | 6 |
| 4.3.1 | Idle Task | 6 |
| 4.3.2 | Serial Port Profile | 6 |
| 4.3.3 | Display Task (LcdDisplay.c or OledDisplay.c) | 6 |
| 4.3.4 | LCD Driver Task (LCD only) | 6 |
| 4.3.5 | Serial RAM Task | 6 |
| 4.3.6 | Background Task | 6 |
| 4.3.7 | Pedometer Task | 6 |
| 4.4 | Messages | 6 |
| 5 | Development Environment | 6 |
| 5.1 | Folder Structure | 6 |
| 5.2 | First Steps | 7 |
| 5.2.1 | Linker Options | 10 |
| 5.2.2 | Programming Options | 11 |
| 5.3 | Debugging using IAR Embedded Workbench | 12 |
| 6 | The Pedometer Task – An Example | 13 |



2 Introduction

The MetaWatch Bluetooth enabled Analog and Digital watches allow anyone to develop custom applications for their phones and watches. The development of an application for the phone is not described in this document. The watch uses the Bluetooth Serial Port Profile (SPP).

| | |
|---------------------------------|-----------------|
| Revision 1.0 | October 3, 2011 |
| MetaWatch Firmware Design Guide | Page 3 of 14 |



2.1 Revision History

| Revision | Details | Date |
|----------|------------------|-------------|
| 1.0 | Original Release | 29 SEP 2011 |
| | | |

3 Hardware

3.1 Common Watch Features

Each watch can be charged, programmed, and debugged using the 4 pin Spy-Bi-Wire interface and clip. Each watch contains the logic required to recharge the 3.7V 75mAH lithium ion coin cell. The charging circuit also prevents the cell from being discharged too much. The MSP430's analog to digital conversion circuitry allows the battery voltage to be monitored.

The watch contains a vibrator. Its default use is to notify the user of a new call or message, signal a low battery, and to indicate that the Bluetooth link is lost.

The watch contains a KIONIX KXTF9-4100 or KXTF9-1026 accelerometer. It can be used to turn the watch into a pedometer or accept tap commands.

The Bluetooth radio is the Texas Instruments CC2560. The radio's current in sniff mode when non-discoverable and non-connectable is <100 μ A.

The watch also contains a light sensor. The light sensor can be used on the Analog Watch to control the contrast of the OLED displays.

3.2 Analog Watch

The analog watch has an analog face and two OLED displays. Each display can be controlled independently. The analog watch has 3 buttons. The middle button is also a pull switch.

3.3 Digital Watch

The digital watch contains a 96x96 pixel LCD display. This low power display consumes less than 5 μ A when the display is static. The digital watch has 6 buttons.

In addition, the digital watch contains a Microchip 23A640 serial ram. The 8K bytes of ram are used as a display buffer.

3.4 Bluetooth Hardware Interface

The Bluetooth radio interface consists of a serial port running at 115.2 kbits/s. The serial port speed limits the Bluetooth throughput to 80 kbits/s. With the current buffer configuration the receive rate is limited to 30 kbits/s. Clear to Send (CTS) is used to wake the microcontroller when the radio is in sniff mode. This allows the MSP430 to remain in a low power mode while waiting for messages from the phone.

4 Software Architecture

This design uses RTOS tasks to separate functions. Inter-task communication uses messages and queues.

4.1 Bluetooth Software Interface

This project contains a Bluetooth stack specifically tailored to the abilities of the Texas Instruments MSP430F5438A. The serial port profile (SPP) was chosen for this project. The application interface is through SerialProfile.h. After calling InitializeSppTask() the SPP task will open the stack, enabled pairing, and allow a connection to be made. It will also automatically go into sniff mode. The user does not have to worry about the details of the Bluetooth connection.

The user application can control turning the radio on/off (which includes opening and closing the stack), setting sniff parameters, disabling sniff completely, and the sniff mode entry delay.

For the full API see the Doxygen output (index.html).

4.2 FreeRTOS

FreeRTOS version V6.0.5 was used for this project. FreeRTOS provides task, queues, semaphores, and the OS tick. The tick occurs at a 1 millisecond rate. At each tick the scheduler switches between tasks that are ready to run. Some deficiencies of the RTOS become apparent when trying to create a low power system, but those problems are outweighed by benefits. The RTOS provides some assurance that each task will get a fair share of the processor. With an RTOS, the software is simpler to write.

4.3 Tasks

Tasks were used to separate each of the main functions.

4.3.1 Idle Task

The idle task is used to control entry into low power mode (LPM). When the queues for all tasks are empty and the Bluetooth Serial Port is ready to sleep the micro will enter LPM3. A timer interrupt or message from the radio can wake the microcontroller.

4.3.2 Serial Port Profile

The serial port profile and underlying stack require 4 tasks. This cannot be changed by the user. The Serial Port Profile handles creating the Bluetooth connection and processes incoming and outgoing messages.

4.3.3 Display Task (LcdDisplay.c or OledDisplay.c)

The display task keeps track of the current mode of the watch. It draws the clock and other system screens. The display task handles the button presses and generates menus.

4.3.4 LCD Driver Task (LCD only)

The LCD driver task uses direct memory access (DMA) to write screen data to the LCD.

4.3.5 Serial RAM Task

This task uses DMA to read and write data from the serial ram. The serial ram task handles addressing the different buffers. When the phone draws a screen into memory it is handled by the serial ram task.

4.3.6 Background Task

The command task handles buttons, vibration, timers, and other functions. For example it handles the Get Device Type message, the Set Vibrate Mode message, and the Set Real Time Clock message.

4.3.7 Pedometer Task

The pedometer task is an example task. It sets up the accelerometer to detect motion.

4.4 Messages

Messages are used for communication with the phone and for inter-process communication. These are defined in Messages.h. User created messages must be in the range of 0x60-0x9f to insure compatibility with future revisions. The RouteMsg() and RouteMsgFromIsr() functions are used to send messages to the correct queue (task).

5 Development Environment

IAR Embedded Workbench for the MSP430 version 5.20 was used for this project.

5.1 Folder Structure

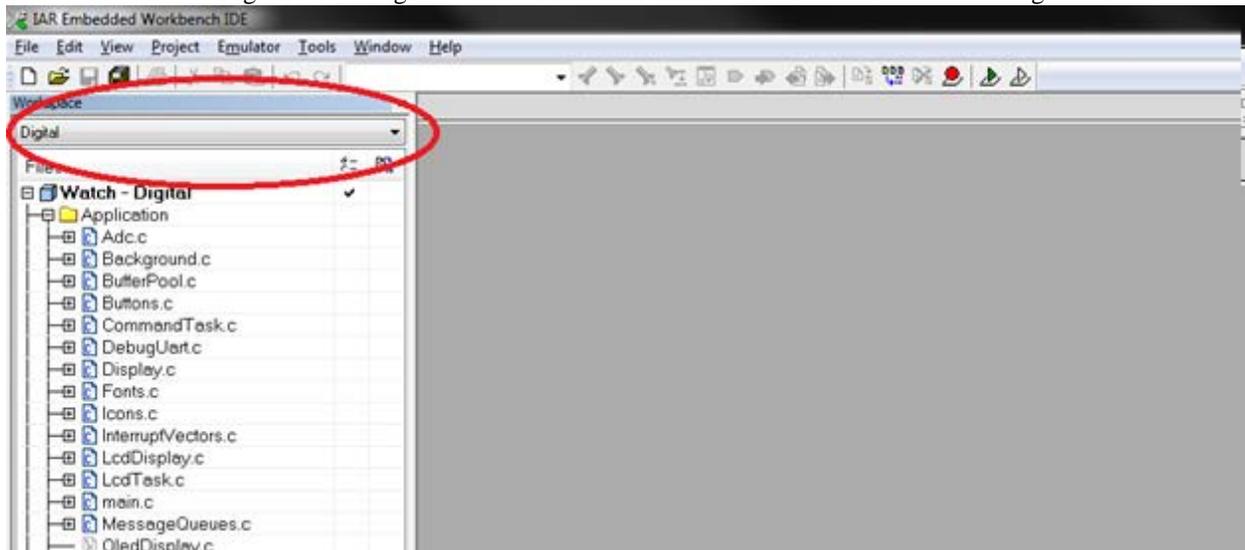
- Common – Files that are common to the Serial Port Profile and user application

| | |
|---------------------------------|-----------------|
| Revision 1.0 | October 3, 2011 |
| MetaWatch Firmware Design Guide | Page 6 of 14 |

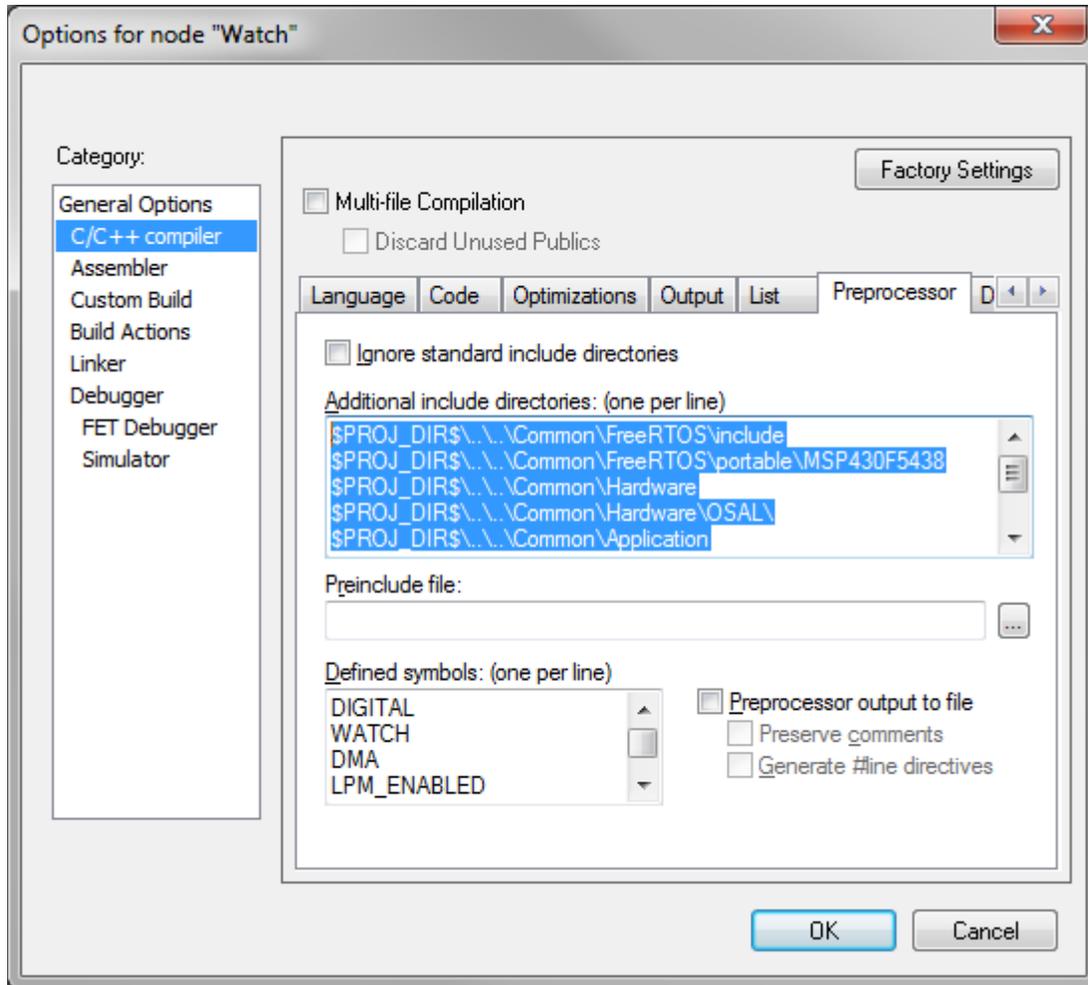
- Application
- FreeRTOS
- Hardware
 - OSAL – Non-volatile storage functions provided by Texas Instruments
- Library – The two library files for the Serial Port Profile (stack).
- Settings – IAR workspace settings
- Watch – The user application
 - Application
 - Hardware - Hardware abstraction layer
 - F5xx_F6xx_Core_Lib – Low level files provided by Texas Instruments
 - Project – IAR project settings

5.2 First Steps

Open the workspace file Theta.eww. One of four projects can be selected Devboard, Digital, Analog, or Analog Devboard. Choose Digital for the digital watch. These instructions will also work for the Analog watch.



Right click on Watch-Digital and select options. Go to the C/C++ compiler category and select the preprocessor tab.



From this page we can see the directories used for the project

They are:

```

$PROJ_DIR$..\..\Common\FreeRTOS\include
$PROJ_DIR$..\..\Common\FreeRTOS\portable\MSP430F5438
$PROJ_DIR$..\..\Common\Hardware
$PROJ_DIR$..\..\Common\Hardware\OSAL\
$PROJ_DIR$..\..\Common\Application
$PROJ_DIR$..\Hardware
$PROJ_DIR$..\Hardware\F5xx_F6xx_Core_Lib\
$PROJ_DIR$..\Application

```

The defined symbols that make this project build as a digital watch are also visible. They are:

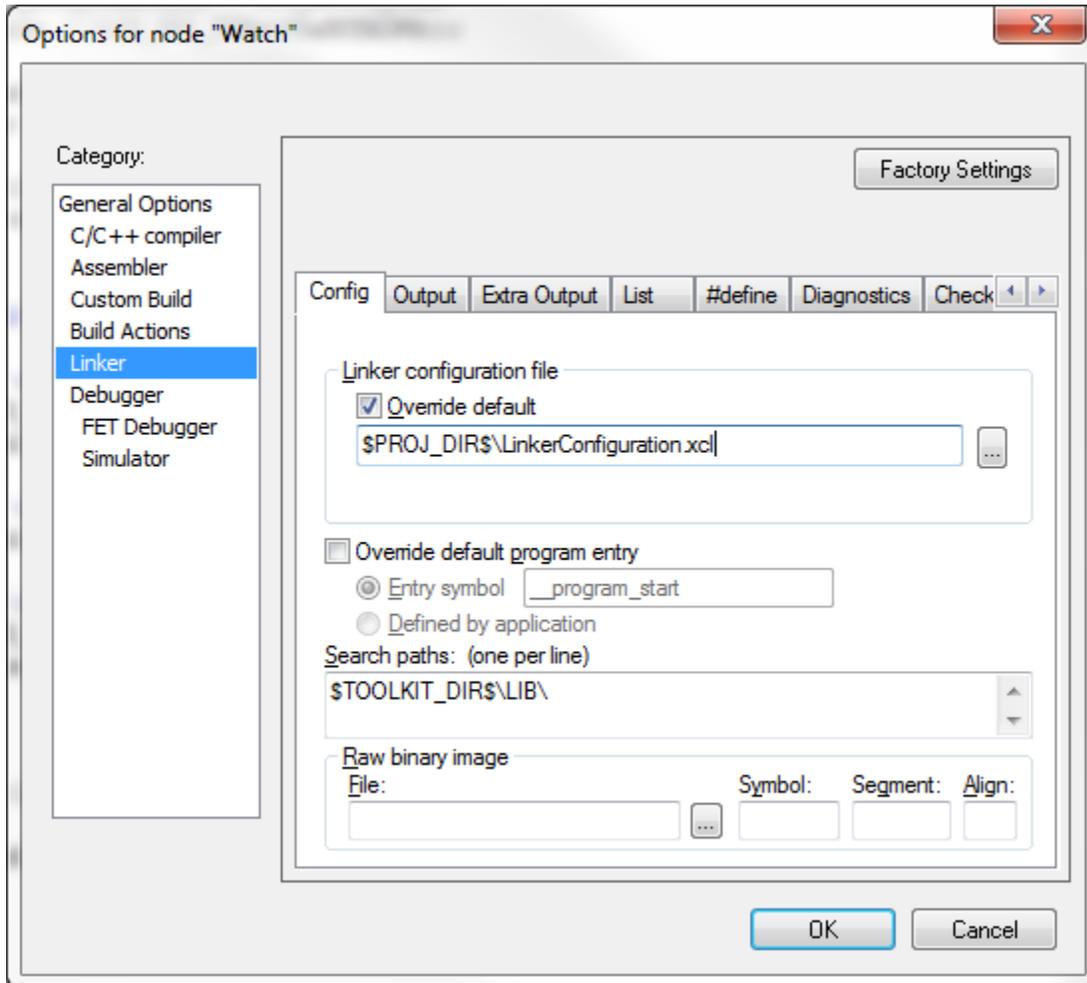
```

DIGITAL
WATCH
DMA
LPM_ENABLED
xTASK_DEBUG

```

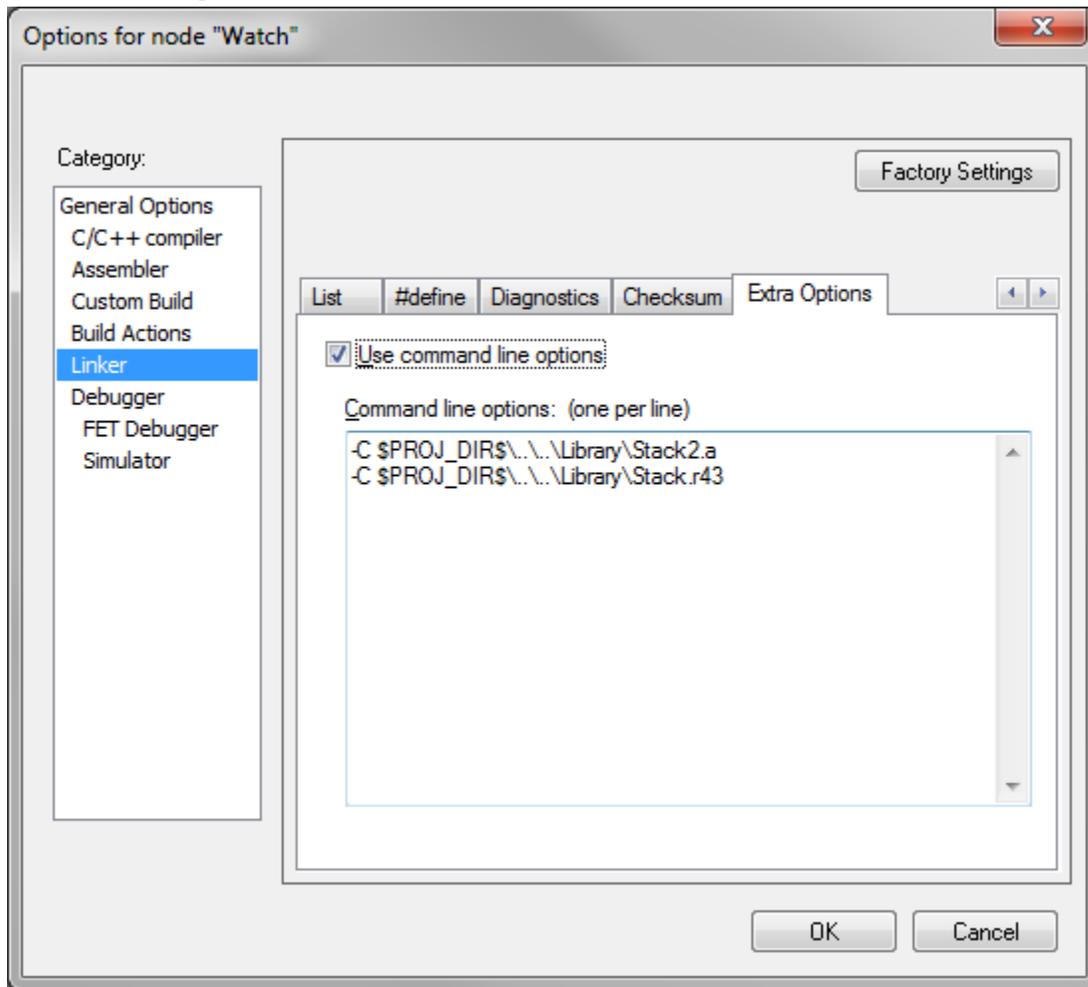
Removing the “x” from TASK_DEBUG will cause the compilation to change. When built with this option the memory usage of each task is printed at a regular interval. The format is “Free Used Total” and is in words (16 bits) not bytes.

Go to the linker configuration page. The special linker file determines where the non-volatile (OSAL) items are stored.

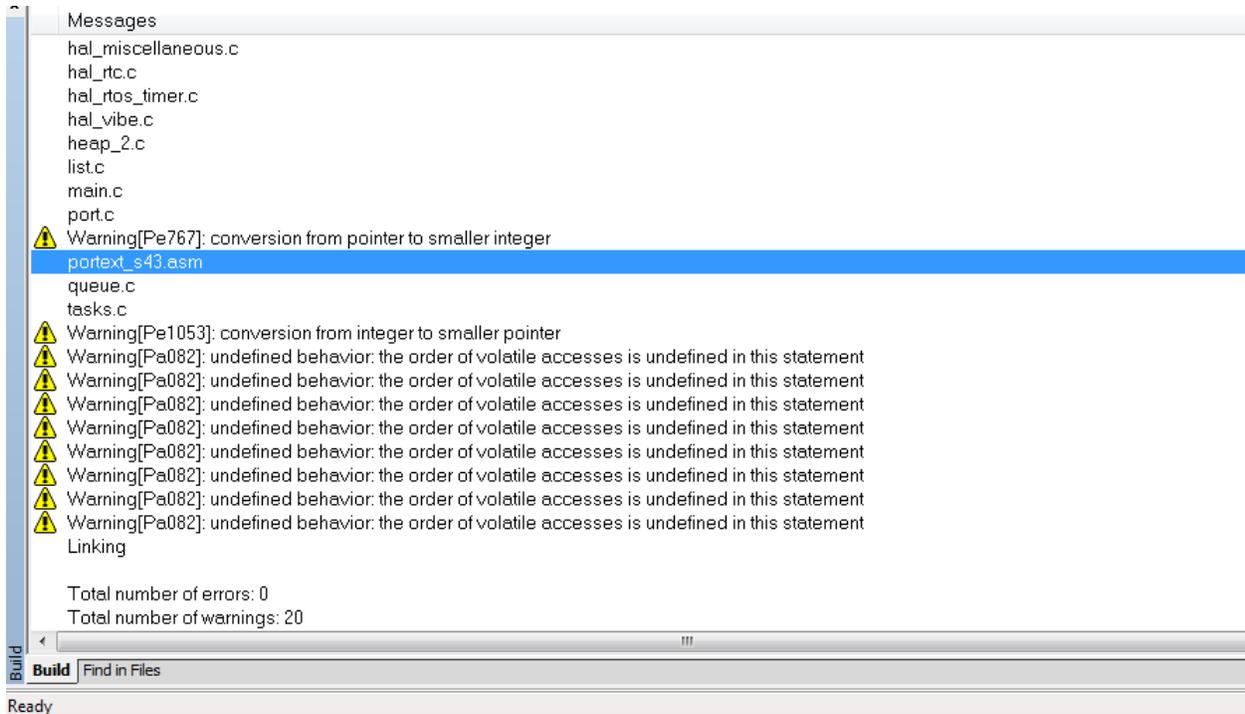


Another important page, which is also in the linker category, is the Extra Options page. This is where the Serial Port Profile (stack) libraries are linked in. Without this there will be undefined functions after the linker has run.

5.2.1 Linker Options



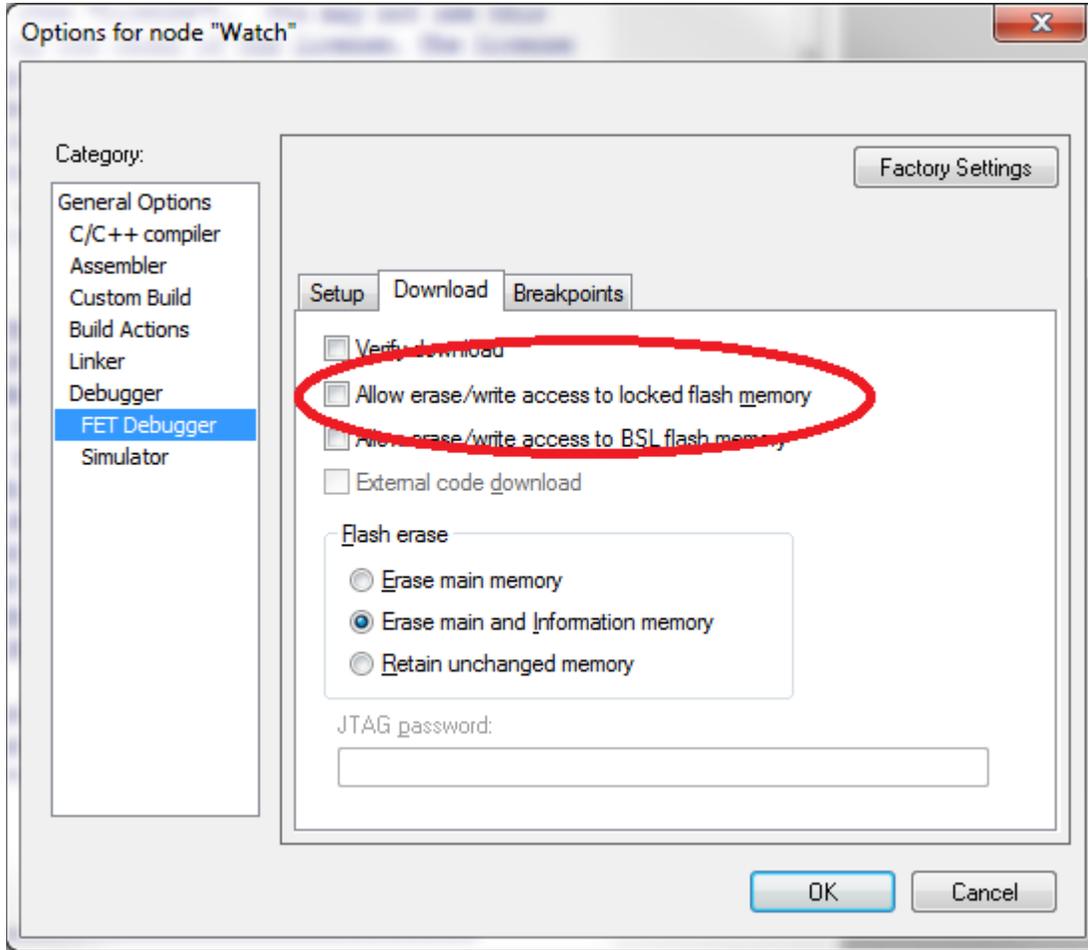
To build the project select Project->Rebuild All. The following output will be generated:



These warnings can be safely ignored.

5.2.2 Programming Options

It is important to make sure the checkmark to allow erase/write access to locked flash memory is not selected. Important calibration information is stored in Information Memory Segment A. If it is erased the accuracy of the timekeeping will be affected.



5.3 Debugging using IAR Embedded Workbench

To debug the watch using IAR (or to program it) connect the clip to the watch and the USB cable to the PC. It is preferable to connect the USB directly to the USB and not through a USB port replicator. Press the play button.



A message will pop up indicating the download is occurring. Now press the go button.



The debugger allows one to set breakpoints, examine memory, and single step through the code.

6 The Pedometer Task – An Example

The pedometer task is an example task. It could contain the code for a pedometer application.

6.1 Queue and Task Creation

Each task has a queue. Queues are used for communication between tasks. Messages from the phone can also be routed to a task's queue. The controlling parameter for a queue is its length. The size should be kept as small as possible. All queues in this system operate on the same type of object; the tMessage object. When objects are placed into a queue they are copied.

```
#define PEDOMETER_TASK_MSG_QUEUE_LEN ( 8 )
#define PEDOMETER_TASK_STACK_DEPTH (configMINIMAL_STACK_DEPTH + 20)
#define PEDOMETER_TASK_PRIORITY (tskIDLE_PRIORITY + 1)

void InitializePedometerTask(void)
{
    QueueHandles[PEDOMETER_QINDEX] =
        xQueueCreate( PEDOMETER_TASK_MSG_QUEUE_LEN, MESSAGE_QUEUE_ITEM_SIZE );

    xTaskCreate(PedometerTask,
                "PEDOMETER",
                PEDOMETER_TASK_STACK_DEPTH,
                NULL,
                PEDOMETER_TASK_PRIORITY,
                &PedometerTaskHandle);
}
```

The stack depth determines how much memory is allocated to the task at run time. Each task requires a minimum amount of memory to handle interrupts. The additional memory is required to store local variables. The InitializePedometerTask() creates a queue and a task.

6.2 Message Handling

The PedometerTask starts by configuring the accelerometer for operation. Once this is complete all future activity occurs due to an interrupt from the accelerometer. The PedometerTask waits for a message in its queue. When it receives a message it is processed by the handler. After a message is processed it is sent to the free queue.

```
/* from PedometerTask() */
for(;;)
{
    if( pdTRUE == xQueueReceive(QueueHandles[PEDOMETER_QINDEX],
                                &PedometerTaskMsg, portMAX_DELAY ) )
    {
        PedometerTaskMessageHandler(&PedometerTaskMsg);

        SendToFreeQueue(&PedometerTaskMsg);

        CheckStackUsage(PedometerTaskHandle, "Pedometer Task");
    }
}
```

When an interrupt occurs a message is sent. It is not shown here, but when this interrupt occurs the processor is



brought out of LPM3 (if required).

```
void AccelerometerWakeOnThresholdIsr(void)
{
    tMessage Msg;
    SetupMessage(&Msg,AccelerometerWakeMsg,NO_MSG_OPTIONS);
    RouteMsgFromIsr(&Msg);
}
```

The RouteMsgFromIsr() function in MessageQueues.c is responsible for sending the message to the correct queue. For each message that is added, the corresponding lines in MessageQueues.c must be added.

```
case AccelerometerWakeMsg:    SendMsgToQ(PEDOMETER_QINDEX,pMsg); break;
```