ARM Cortex-M3
Stellaris Shuru Development Kit
Manual

Powered by

Centre for Electronic Design and Technology
Netaji Subhas Institute of Technoloy

Table of Contents

1. Embedded Systems

1.1 What is an Embedded System?

The first question that needs to be asked is **"What exactly is an embedded system?"** To be fair, however, it is much easier to answer the question of what an embedded system is not, than to try and describe all the many things that an embedded system can be. **An embedded system is usually a computer that is implemented for a particular purpose.** In contrast, an average PC computer usually serves a number of purposes: checking email, surfing the Internet, listening to music, word processing, etc. However, embedded systems usually only have a single task, or a very small number of related tasks that they are programmed to perform.

Every home has several examples of embedded systems. Any appliance that has a digital clock, for instance, has a small-embedded microcontroller that performs no other task than to display the clock. Modern cars have embedded systems onboard that control such things as ignition timing and anti-lock brakes using input from a number of different sensors.

Embedded systems rarely have a generic interface, however. Even if embedded systems have a keypad and an LCD display, they are rarely capable of using many different types of input or output. An example of an embedded system with I/O capability is a security alarm with an LCD status display, and a keypad for entering a password.

Due to strong features resemblance with PC Computers but in a smaller footprint, an embedded system is called an **"Embedded Computer"**. In the rest part of the manual, these two terms are used interchangeably.

In general, an Embedded System:

- Is a system built to perform its duty, completely or partially independent of human intervention.
- Is specially designed to perform a few tasks in the most efficient way.
- Interacts with physical elements in our environment, viz. controlling and driving a motor, sensing temperature, etc.

1.2 What are Embedded Systems Used For?

The uses of embedded systems are virtually limitless, because every day new products are introduced to the market that utilize embedded computers in novel ways. In recent years, hardware such as microprocessors, microcontrollers, and FPGA chips have become much cheaper. So when implementing a new form of control, it's wiser to just buy the generic chip and write your own custom software for it. Producing a custom-made chip to handle a particular task or set of tasks costs far more time and money. Many embedded computers even come with extensive libraries, so that "writing your own software" becomes a very trivial task indeed.

In the present life scenario, embedded systems are invariably used in all aspects of modern life. Table 1.1 gives the examples of these systems in various fields.

Field	Examples
Telecommunications	Telephone Switches, Routers and Network Bridges
Consumer Electronics	Personal Digital Assistants, Digital Cameras, GPS Receivers
Transport Systems	Motor Controllers, Navigators, Centralized Locking, Traffic Lights, ABS
Medical Systems	Medical Imaging, Electronic Stethoscopes
Education Systems	Projectors, Attendance Systems
Entertainment	Portable Electronic Games, simulation of real life objects and their behavior

Table 1.1 – Examples of Embedded Systems in various fields

1.3 Embedded Systems - Caveats

Embedded systems may be economical, but they are often prone to some very specific problems. A PC computer may ship with a glitch in the software, and once discovered, a software patch can often be shipped out to fix the problem. An embedded system, however, is frequently programmed once, and the software cannot be patched. Even if it is possible to patch faulty software on an embedded system, the process is frequently far too complicated for the user.

Another problem with embedded computers is that they are often installed in systems for which unreliability is not an option. For instance, the computer controlling the brakes in your car cannot be allowed to fail under any condition. The targeting computer in a missile is not allowed to fail and accidentally target friendly units. As such, many of the programming techniques used when throwing together production software cannot be used in embedded systems. Reliability must be guaranteed before the chip leaves the factory. This means that every embedded system needs to be tested and analyzed extensively.

2. General Purpose v/s Embedded Computers

A general-purpose computer like your PC in contrast to the Embedded Systems is covered by layer of Operating System, which does many different tasks, runs many processes simultaneously, divides each process into threads and so on. The user support and interface is of the prime importance while designing the general purpose PC. An embedded system is generally required to be Real Time where as no such prerequisite is required for a general purpose PC. **96% of the computers used in the world today are embedded systems.**

Differences between general purpose and embedded computers

1. **Re-programmability and dedicated function** – The feature of re-programmability is unheard in embedded computers whereas it is must for general-purpose computers, which must switch between user programs and system programs number of times. Embedded computers on the other hand are designed for dedicated function and once programmed their firmware is generally not changed unless there is need for upgrade or improvement.
2. **Features, RAM/ROM/Speed/Processor** – In embedded systems, capabilities are only provided if they make sense but that's not the case in general purpose computers where all the optimum hardware features should be provided to ensure compatibility with constantly changing bulky software.
3. **Program Execution** – A general-purpose computer generally takes time to boot the operating system before it starts executing the user-defined programs. On the other hand, an embedded computer should immediately start performing its dedicated task after the power is switched on.
4. **Reliability** – Embedded systems are required to be highly reliable because they are used in critical applications like medical imaging, aircraft navigation, banking facilities etc. Their breakdown can cause loss of life and property. General-purpose computers though preferred to be reliable, are generally not involved in critical sections as such.
5. **Real Time Performance** – Embedded systems are generally fast responding to changes because they are always executing the single task. Hence, they are able to response within a certain time interval of change and come in the category of real time systems.
6. **Environment Proof** – Embedded systems may be installed at various locations like roads, power plants, and aircrafts and hence must be protected from various environmental factors. On the other hand, general-purpose computers kept under the roof in offices or in homes do not have this requirement.

So now you have a clear idea of what embedded and general-purpose systems are and what they are required to do. The subsequent sections of this document will solely concentrate on embedded systems, architecture specifications and design using "**Texas Instruments® Stellaris® ARM Cortex-M3 32-bit microcontrollers"**.

3. Microcontrollers v/s Microprocessors

Microcontrollers of microprocessors are the heart of any embedded system. Perhaps, it is required to use one of the two in a system for it to be called embedded. The above two devices have computing power and can change their functionality based on the user program that is fed into their memories.

Microcontrollers or microprocessors based systems are easier and simpler to design than a digital state machine based system. It is cheaper to upgrade or modify the functionality of a controller based system by just changing the firmware (software) rather than to modify a purely digital state machine which requires circuit modifications for even the slightest of change.

You might have noticed that the terms – microprocessor and microcontroller have been used interchangeably up to now. It is time to consider the differences between the two devices while paying more attention to ARM Cortex M3 series microcontroller.

A microprocessor includes a central processing unit integrated on a single chip. It includes ALU – Arithmetic Logic Unit, Registers and Control Unit put together. The data bys, address bus and system bus are brought out of the chip to integrate with memory (Flash/EEPROM/ SRAM) and I/O – input/output subsystem so that we can have a fully functional computer. This fully functional computer is called a microcomputer. 8085 and 8086 are microprocessors from Intel are quite popular and generally taught in various engineering courses across India. All the various PC processors from Intel, like the 80386, Pentium, etc. are successors of the 8086 processor.

When the microcomputer talked above is integrated on a single silicon chip it is called a microcontroller. A microcontroller family is defined as the series of controllers having the same CPU core and architecture but different memory, speed and I/O features. Today there are many microcontroller families: Atmel's AVR, Intel's 8048 and 8051, Motorola's 68HC11, Zilog's Z8, Microchip's PIC and now ARM's Cortex-M series microcontrollers which are manufactured by a number companies like Texas Instruments, STMicroelectronics, Philips NXP, Renesas etc.

This documents focuses on Texas Instruments Stellaris ARM Cortex-M3 based microcontroller.

4. Microcontroller Architectures: CISC and RISC

The proper understanding of this topic requires a lot of prior knowledge about instruction format, addressing modes, length of instruction, phases of execution of an instruction. As you may not be having the prerequisites at this time, the topic is explained in a slightly easy manner and some technical intricacies are skipped. Prior to going into the details of the architecture it is required to grasp a few basic fundamentals.

Instruction Set – The accumulation of all the assembly language instructions or mnemonics that the processor understands and can execute them is known as the instruction set of that processor. The manufacturer of the particular architecture provides the instruction set. The mnemonics can be directly translated to machine language (binary).

Instruction Code – An instruction code is a group of bits that instruct the computer to perform a specific operation. It generally includes opcode (what to do) and address of the operand (where to do). The total number of bits in the instruction code is called its length.

CPU Registers – They are 1 word (8 bits in 8-bit CPU, 16 bits in 16-bit CPU and 32 bits in 32-bit CPU) memory block present inside the CPU and contain operands to be provided to the ALU section and also store the result of operation by the ALU.

Table 4.1 compares the two types of microprocessor architectures, which are used, RISC being the latest one.

CISC – Complex Instruction Set Computer	RISC – Reduced Instruction Set Computer
• Large number of instructions, 100-250. • Instructions that manipulate operands directly in memory. • Variable length instruction formats. • Small number of registers in the CPU	• Relatively few instructions. • Memory access limited to store and load instructions. • Fixed length instructions. • Single cycle instruction execution. • Large number of CPU registers. • All operations done within the registers of the CPU.

Table 4.1 – Comparison of CISC and RISC Architectures

The ARM Cortex M3 uses RISC microprocessor architecture and is discussed in detail in the remaining part of the manual.

5. Background of ARM and ARM Architecture

5.1 A Brief History

ARM was founded in 1990 as Advanced RISC Machines Ltd., a joint venture of Apple Computer, Acorn Computer Group, and VLSI Technology. In 1991, ARM introduced the ARM6 processor family, and VLSI became the initial licensee. Subsequently, additional companies, including Texas Instruments, NEC, Sharp and ST Microelectronics, licensed the ARM processor designs.

Nowadays ARM partners ship in excess of 2 billion ARM processors each year. Unlike many semiconductor companies, ARM doesn't not manufacture processors or sell the chips directly. Instead it licenses the processor designs to business partners. This business model is commonly called intellectual property (IP) licensing.

5.2 Architecture Versions

Over the years, ARM has continued to develop new processors and system blocks. These include the popular ARM7TDMI processors, more recently the ARM11 processor family and latest being the ARM Cortex Processor family. Table 5.1 summarizes the various architecture versions and processor families.

Architecture	Processor Family
ARMv1	ARM1
ARMv2	ARM2, ARM3
ARMv3	ARM6, ARM7
ARMv4	StrongARM, ARM7TDMI, ARM9TDMI
ARMv5	ARM7EJ, ARM9E, ARM10E, XScale
ARMv6	ARM11, ARM Cortex –M
ARMv7	ARM Cortex-A, ARM Cortex-R, ARM Cortex-M

Table 5.1 – ARM architecture versions and processor families

In the manual we focus on the ARMv7 architecture and the ARM Cortex-M processor family. But before moving onto the Cortex-M, it is worthwhile to mention about the Cortex-A and Cortex-M and cite the basic differences.

5.3 ARM Cortex Processor Profiles

In the ARMv7 architecture the design is divided into three profiles –

1. **A Profile (ARMv7-A)** - is designed for high performance open application platforms. They can handle complex applications such as high end embedded operating systems. Example products include high-end smartphones, tablets PCs and PDAs.
2. **R Profile (ARMv7-R)** - is designed for high end embedded systems in which real time performance is needed. Example applications include high-end car braking systems.
3. **M-Profile (ARMv7-M)** – is designed for deeply embedded microcontroller type systems. Processors belonging to this profile are the subject for this manual and are studied in greater detail.

5.4 Instruction Set Development

Enhancement and extension of instruction sets used by the ARM processors has been one of the key driving forces of the architecture's evolution.

Historically, two different instruction sets were supported on the ARM processor: the **ARM instructions**, which are 32 bits and **Thumb instructions**, which are 16 bits. During program execution, the processor can be dynamically switched between the ARM state and the Thumb state. The Thumb instruction set provides only a subset of the ARM instructions, but it can provide higher code density. It is useful for products with tight memory requirements.

The **Thumb-2** technology extended the Thumb Instruction Set Architecture (ISA) into a highly efficient and powerful instruction set that delivers significant benefits in terms of ease of use, code size and performance. The extended instruction set in Thumb-2 is a superset of the previous 16-bit Thumb instruction set, with additional 16-bit instructions alongside 32-bit instructions. It allows more complex operations to be carried out in the Thumb state, thus allowing higher efficiency by reducing the number of states switching between ARM state and Thumb state.

The Cortex-M3 processor supports only the Thumb-2 (and traditional Thumb) instruction set. Instead of using ARM instructions, as in traditional ARM processors, it uses Thumb-2 instruction set for all operations. As a result, the Cortex-M3 processor is not backward compatible with traditional ARM processors, which use the ARM as well as Thumb instruction set.

The Thumb-2 instruction set is a very important feature of the ARMv7 architecture. For the first time, hardware divide instruction is available on an ARM processor, and a number of multiply instructions are also available.

6. Overview of the Cortex-M3

6.1 Salient Features of the Cortex-M3

- 32-bit microprocessor.
- 32-bit data path, 32-bit register bank and 32-bit memory interfaces.
- Harvard Architecture – separate instruction bus and data bus.
- 3-stage pipeline with branch speculation.
- Thumb-2 instruction set.
- No switching between ARM state and thumb state.
- Instruction fetches are 32 bits. Up to two instructions can be fetched in one cycle. As a result, there's more available bandwidth for data transfer.
- ALU with hardware divide and single cycle multiply.
- Configurable Nested Vector Interrupt Controller (NVIC).
- Maximum of 240 external interrupts can be configured.
- Low gate count, suitable for low power designs.
- Memory Protection Unit (MPU).
- Operation Mode Selection – user and privilege modes.
- Advanced debug components.

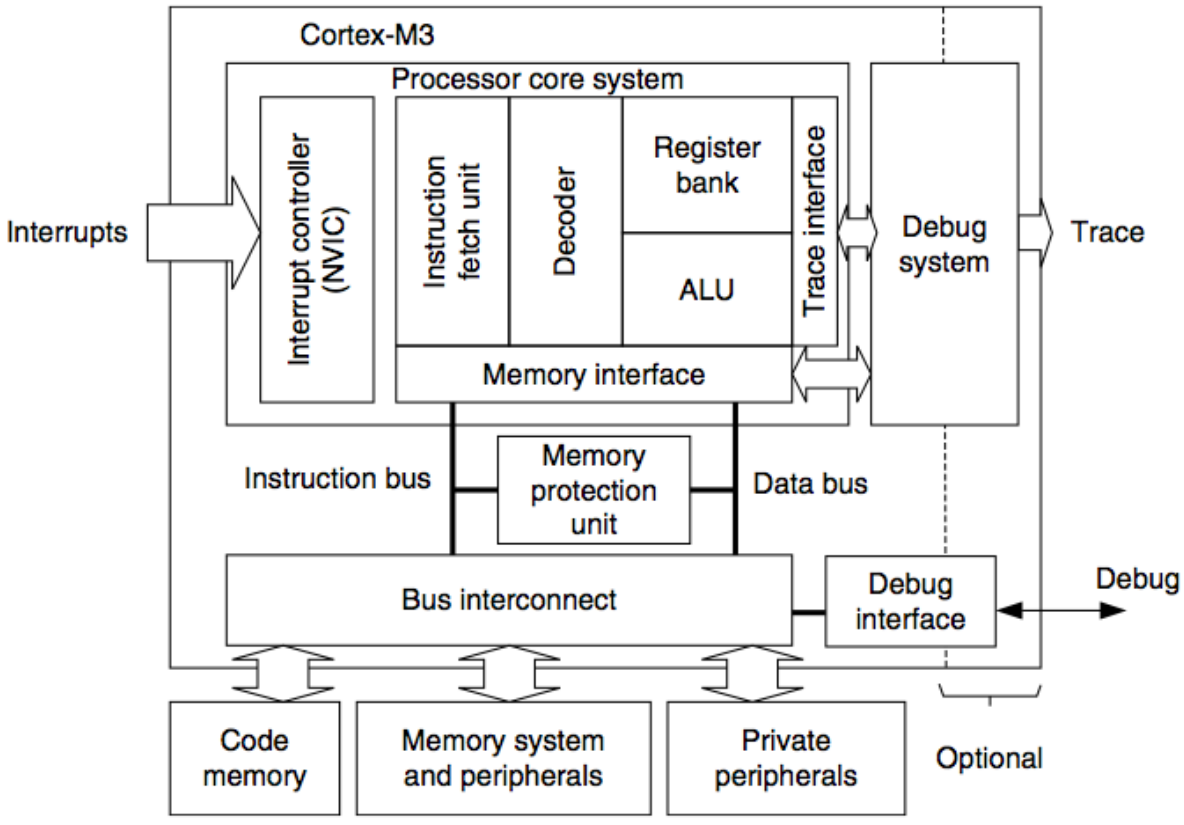


Figure 6.1 ARM Cortex-M3 Architecture View

6.2 Registers

- The Cortex-M3 has registers R0 through R15. R13 (the stack pointer) is banked, with only one copy of the R13 visible at a time. Figure 6.2
  - R0 – R12: General Purpose Registers. These are 32 bit registers for data operations. Some 16-bit Thumb instructions can only access a subset of these registers (low registers R0-R7).
  - R13: Stack Pointers. Contains two stack pointers. They are banked so that only one is visible at a time.
    Main Stack Pointer (MSP) – The main stack pointer used by the Operating system and exception handlers.
    Process Stack Pointer (PSP) – used by the application code.
  - R14: Link Register. When a subroutine is called, the return address is stored in the link register.
  - R15: The Program Counter. The program counter is the current program address.
- The Cortex-M3 also has a number of special registers. Figure 6.3. They are -
  - Program Status registers (PSR)
  - Interrupt Mask registers (PRIMASK, FAULTMASK and BASEPRI).
  - Control register (CONTROL)
- The Cortex-M3 has 18 types of registers in total compared to 37 registers for traditional ARM.

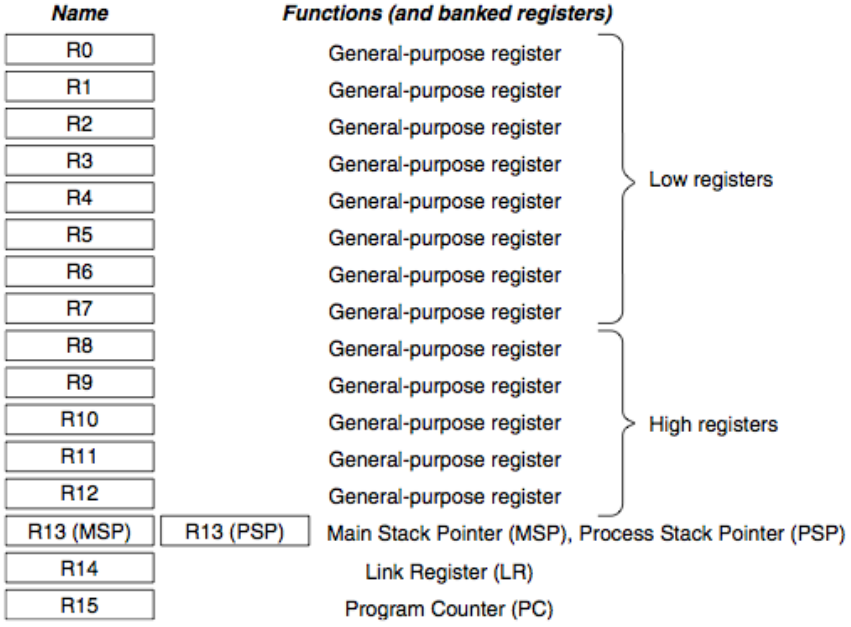


Figure 6.2 Registers in Cortex-M3

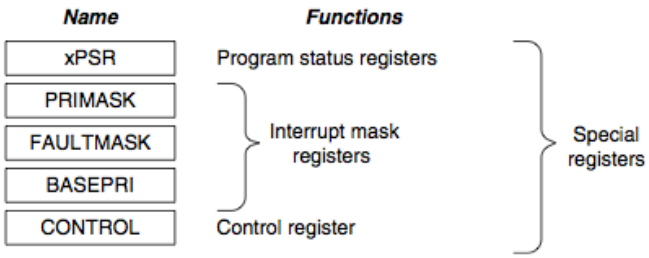


Figure 6.3 Special Registers in Cortex-M3

6.3 Memory Map

The Cortex-M3 has predefined memory maps, which allows built in peripherals, such as the interrupt controller and debug components, to be accessed by simple memory access instructions. The predefined memory map also allows the Cortex-M3 processor to be highly optimized for speed and ease of integration in system-on-a-chip (SoC) designs.

The Cortex-M3 design has an internal bus infrastructure optimized for this memory usage. In addition, the design allows these regions to be used differently. For example, data memory can still be put into the CODE region, and program code can be executed from an external Random Access Memory (RAM) region. The Cortex-M3 memory map is outlined in Figure 6.4

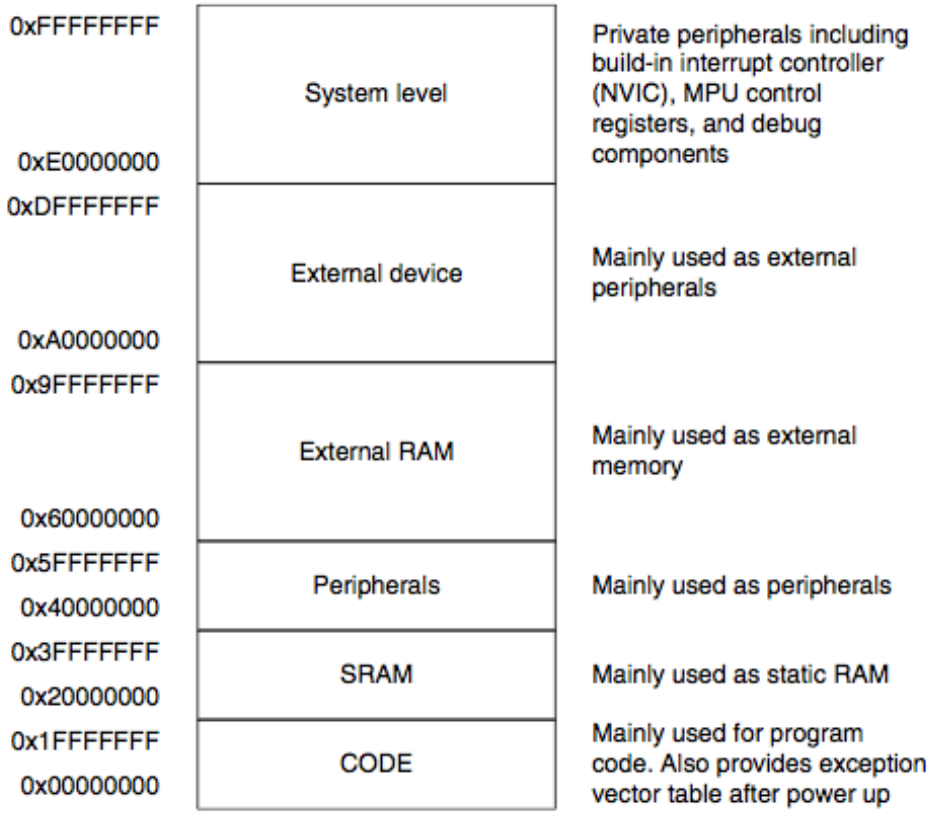


Figure 6.4 Cortex-M3 Memory Map

6.4 Operation Modes

The Cortex-M3 processor has two modes and two privilege levels. The operation modes (thread mode and handler mode) determine whether the processor is running a normal program or running an exception handler like an interrupt handler or system exception handler. The privilege levels provide a mechanism for safeguarding memory accesses to critical regions as well as providing a basic security model.

When the processor is running a main program (thread mode), it can be either in a privileged state or a user state, but exception handlers can only be in a privileged state. When the processor exit reset, it is in thread mode with privileged access right. In this state,

a program has access to all memory ranges and can use all supported instructions.

Software in the privileged access level can switch the program into the user access level using the control register. When an exception takes place, the processor will always switch back to the privileged state and return to the previous state when exiting the exception handler. A user program cannot change back to the privileged state by writing to the control register. It has to go through an exception handler that programs the control register to switch the processor back into the privileged access level when returning to thread mode. Figure 6.5 outlines the available operation and privilege levels.

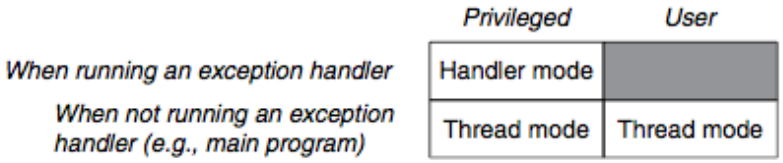


Figure 6.5 Operation modes and Privilege Level in Cortex-M3

6.5 Nested Vector Interrupt Controller (NVIC)

The Cortex-M3 processor includes an interrupt controller called the Nested Vectored Interrupt Controller (NVIC). It is closely coupled to the processor core and provides a number of features as follows:

- •Nested interrupt support
- •Vectored interrupt support
- •Dynamic priority changes support
- •Reduction of interrupt latency
- •Interrupt masking

Nested Interrupt support - All the external interrupts and most of the system exceptions can be programmed to different priority levels. When an interrupt occurs, the NVIC compares the priority of this interrupt to the current running priority level. If the priority of the new interrupt handler is higher than the current level, the interrupt handler of the new interrupt will override the current running task.

Vectored Interrupt Support - When an interrupt is accepted, the starting address of the interrupt service routine (ISR) is located from a vector table in memory.

Dynamic Priority Changes Support - Priority levels of interrupts can be changed by software during run time. Interrupts that are being serviced are blocked from further activation until the ISR is completed, so their priority can be changed without risk of accidental reentry.

Reduction of Interrupt Latency - The Cortex-M3 processor also includes a number of advanced features to lower the interrupt latency. These include automatic saving and restoring some register contents and reducing delay in switching from one ISR to another.

Interrupt Masking - Interrupts and system exceptions can be masked based on their priority level or masked completely using the interrupt masking registers BASEPRI, PRIMASK, and FAULTMASK.

7. Texas Instruments® Stellaris® 32-bit ARM Cortex-M Family

In the Stellaris Shuru Development board we use an ARM Cortex-M3 based microcontroller from the Stellaris family manufactured by Texas Instruments. Before getting into the details of the controller used the specifics of the development kit, we would like to highlight some of the prime features available on the Stellaris family of microcontroller by Texas Instruments.

- **ARM® Cortex™ -M3 v7-M Processor Core**
  - Up to 80 MHz
  - Up to 100MIPS (at 80 MHz).
- **On-Chip Memory**
  - 256KB Flash, 96KB SRAM.
  - ROM Loaded with Stellaris Driver-Lib, Boot-loader, AES Tables and CRC.
- **External Peripheral Interface (EPI)**
  - 32-bit dedicated parallel bus for external peripherals.
  - Supports SDRAM, SRAM/Flash, M2M.
- **Advanced Serial Integration**
  - 10/100 Ethernet MAC and PHY.
  - 3 CAN 2.0 A/B Controllers.
  - USB Full Speed, OTG/Host/Device.
  - 3 UARTs with IrDA and ISO 7816 support.
  - 2 I^2Cs.
  - 2 Synchronous Serial Interfaces (SSI).
  - Integrated Interchip Sound (I^2S).
- **System Integration**
  - 32 Channel DMA Controller.
  - Internal precision 16MHz oscillator.
  - 2 watchdog times with separate clock domains.
  - ARM Cortex SysTick timer.
  - 4 32-bit timers with RTC capability.
  - Lower power battery backed hibernation module.
  - Flexible pin-muxing capability.
- **Advanced Motion Control**
  - 8 advanced PWM outputs for motion and energy applications.
  - 2 Quadrature Encoder Inputs (QEI).
- **Analog**
  - 2 x 8 Channel 10-bit ADC (for a total of 16 channels).
  - 3 analog comparators.
  - On-chip voltage regulator (1.2V internal operation).

The controller used on the Stellaris Shuru board may not have all the features outlined above. We discuss the features and capabilities of the Stellaris LM3S608 ARM Cortex-M3 based microcontroller, which is the heart of the Shuru development kit.

8. Texas Instruments® Stellaris® LM3S608 Microcontroller - Features

A Texas Instruments® Stellaris® LM3S608 microcontroller forms the heart of the Stellaris Shuru board. The underlying sections describe the various features available on the controller and its pin out.

- 32-bit RISC Performance
  - 32-bit ARM® Cortex™-M3 v7M architecture.
  - SysTick timer, providing a simple 24-bit clear-on-write, decrementing, wrap-on-zero counter.
  - Thumb® compatible Thumb-2 only instruction set.
  - 50-MHz Operation
  - Integrated Nested Vector Interrupt Controller.
  - 23 interrupt with eight priority levels.
- Internal Memory
  - 32KB single cycle flash.
  - 8KB single cycle SRAM.
- GPIOs
  - 5-28 GPIOs, depending on configuration.
  - 5V tolerant input configuration.
  - Fast toggle capable of a change every two-clock cycle.
  - Programmable control for GPIO interrupts.
  - Programmable control for GPIO pad configuration.
- General Purpose Timers
  - Three general-purpose timers modules each of which provides two 16-bit timers/ counters. Each can be configured independently:
    - As a single 32-bit timer.
    - As one 32-bit Real-time Clock.
    - For Pulse Width Modulation.
- ARM FiRM-compliant Watchdog Timer.
- ADC
  - Eight analog input channels.
  - Single ended and differential input configurations.
  - On-chip internal temperature sensor.
  - Sample rate of 500 thousand samples/second.
- UART
  - Two fully programmable 16C550-type UARTs.
  - Separate 16x8 transmit (TX) and receive (RX) FIFOs to reduce CPU interrupt service loading.
  - Fully programmable serial interface characterisitics
    - 5,6,7 or 8 data bits.
    - Even/Odd/Stick or No-Parity generation.
    - 1 or 2 stop bit generation.
- Synchronous Serial Interface
  - Master of Slave Operation.
  - Programmable clock bit rate and operation.
  - Programmable data frame size from 4 to 16 bits.

- I^2C
  - Devices on the I2C bus can be designated as either master or a slave.
  - Four I2C modes
    - Master transmit
    - Mater receive
    - Salve transmit
    - Slave receive
  - Two transmission speeds, standard (100kbps) and Fast (400kbps).
  - Master and slave interrupt generation.
- Analog Comparators
  - One integrated analog comparator.
  - Configurable for output to drive an output pin, generate an interrupt or initiate an ADC sample sequence.
- Power
  - On-chip Low Drop (LDO) voltage regulator with programmable output user adjustable from 2.25V to 2.75V.
  - Low power options on controller: Sleep and Deep Sleep Modes.
  - Low power options for peripherals: software controls shutdown of individual peripherals.
  - 3.3V supply brownout detection and reporting via interrupt or reset.
- Flexible Reset Sources
  - Power-on Reset (POR)
  - Reset pin assertion.
  - Brownout (BOR) detector alerts to system power drops.
  - Software reset.
  - Watchdog timer reset.
- JTAG
  - IEEE 1149.1-1990 compatible Test Access Port (TAP) controller.
  - Four bits Instruction Register (IR) chain for storing JTAG instructions.
  - IEEE standard instructions: BYPASS, IDCODE, SAMPLE/PRELOAD, EXTEST and INTEST.
  - Integrated ARM Serial Wire Debug (SWD).

9. Texas Instruments® Stellaris® LM3S608 Microcontroller – Pin out

The soul of the Stellaris Shuru is the LM3S608 microcontroller. It follows the ARM Cortex-M 32-bit RISC architecture. The LM3S608 used on the kit is in the 48-pin Thin Quad Flat Package (TQFP). The pin assignment is shown is Figure 9.1.

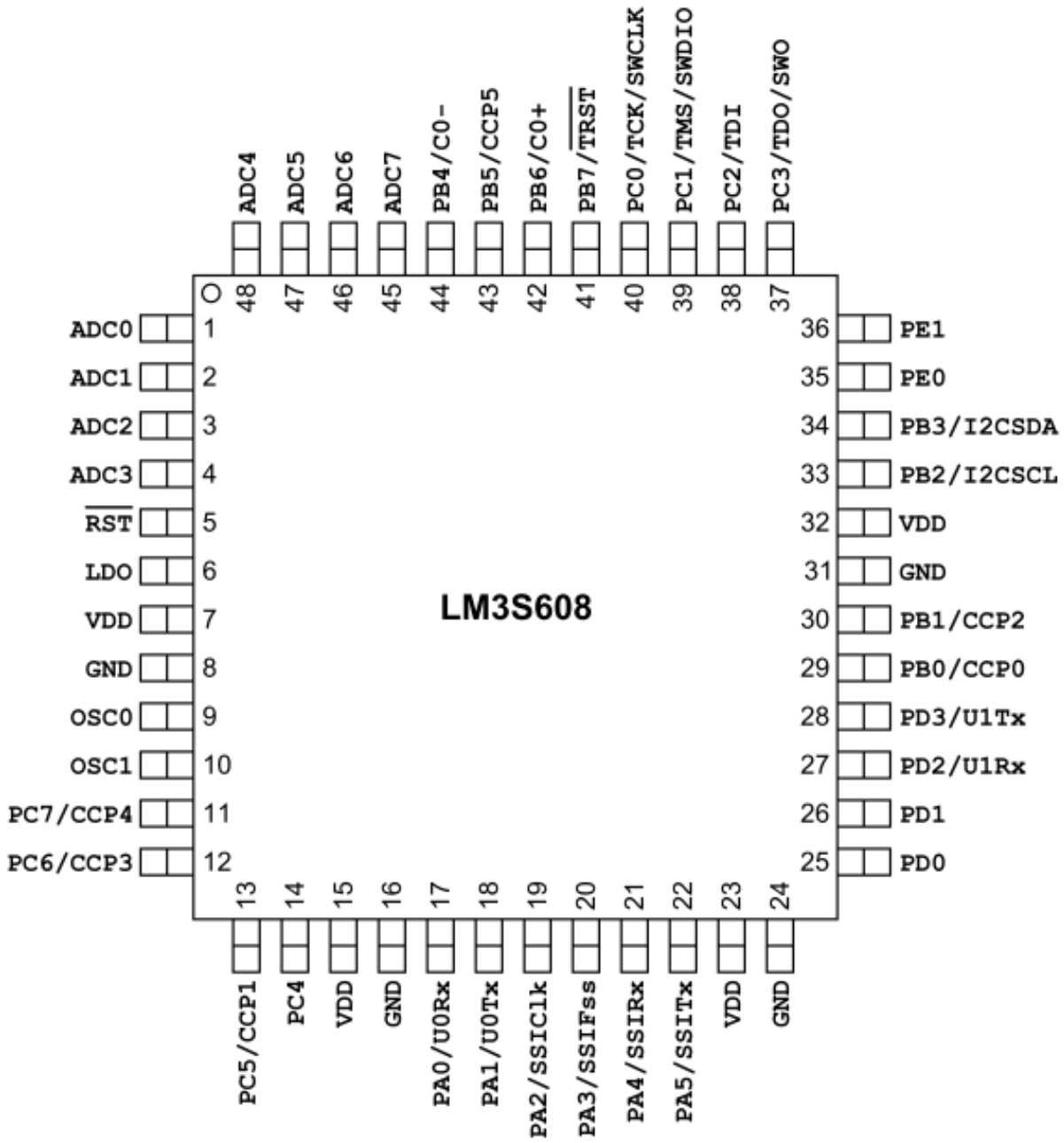


Figure 9.1 LM3S608 Pin Configuration (TQFP Package)

10. Introduction to the Stellaris Shuru Kit

The Stellaris Shuru kit has unique and substantial features, which can be tested and evaluated, right out of the box.

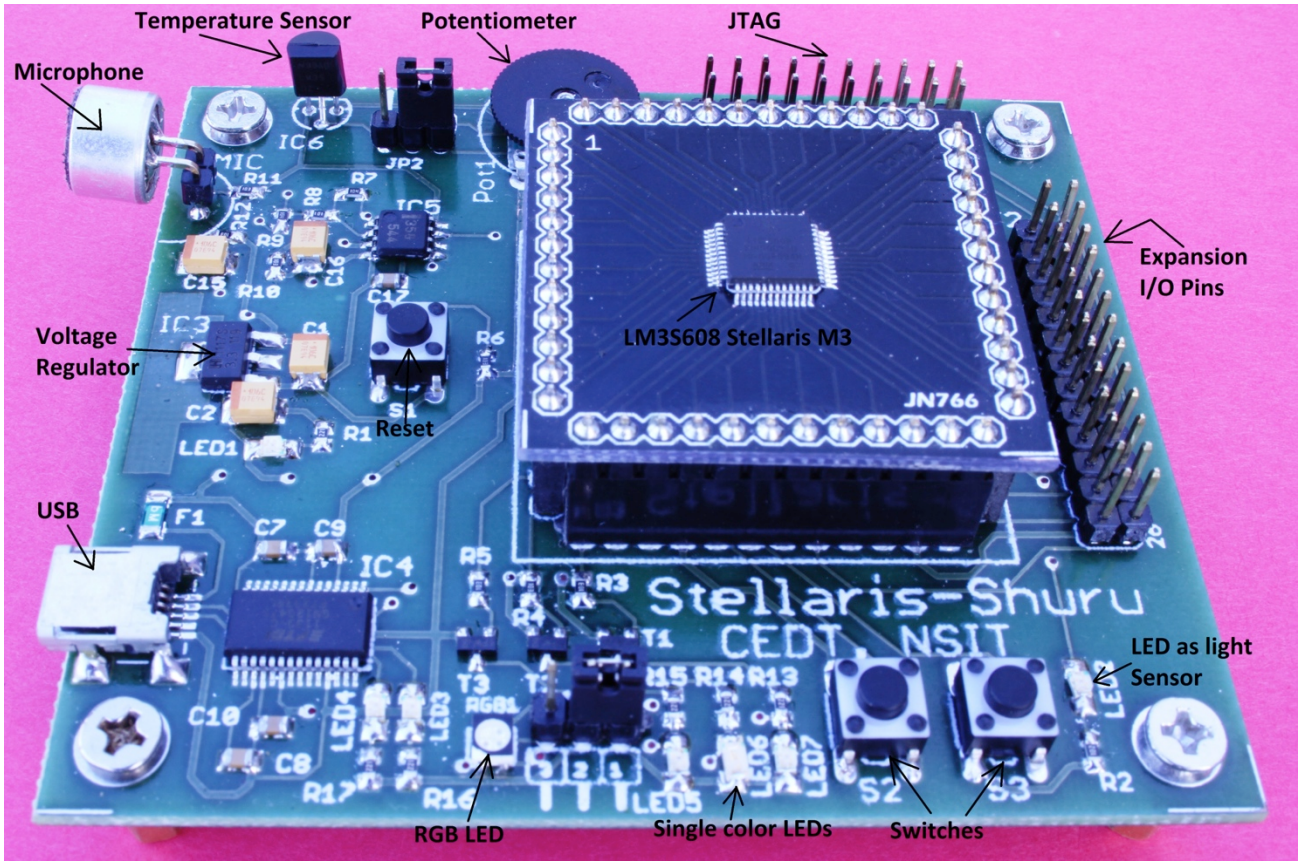


Figure 10.1 Stellaris Shuru Kit

10.1 Features Overview

- Stellaris LM3S608 microcontroller placed on break out board, which can be removed and replaced with any other Stellaris 48 pin microcontroller.
- User programmable push buttons and ultra-bright LEDs, both unicolor and RGB.
- Reset pushbutton and power indicator LED.
- A LM35 temperature sensor for taking temperature readings using the Analog to Digital Convertor.
- Thumbwheel potentiometer for driving an Analog to Digital Convertor.
- Microphone circuit with high gain and sensitivity.
- Ambient light sensor using a LED in reverse bias.
- Standard ARM 20-pin JTAG debug connector.
- I/O signal breakout pads for hardware prototyping.
- UART0 accessible through a USB virtual COM port (VCP).
- Programmable through UART using preinstalled boot loader.
- USB interface for all communication and power.

10.2 Board Layout

Figure 10.2 shows the board layout for the kit from the component side.

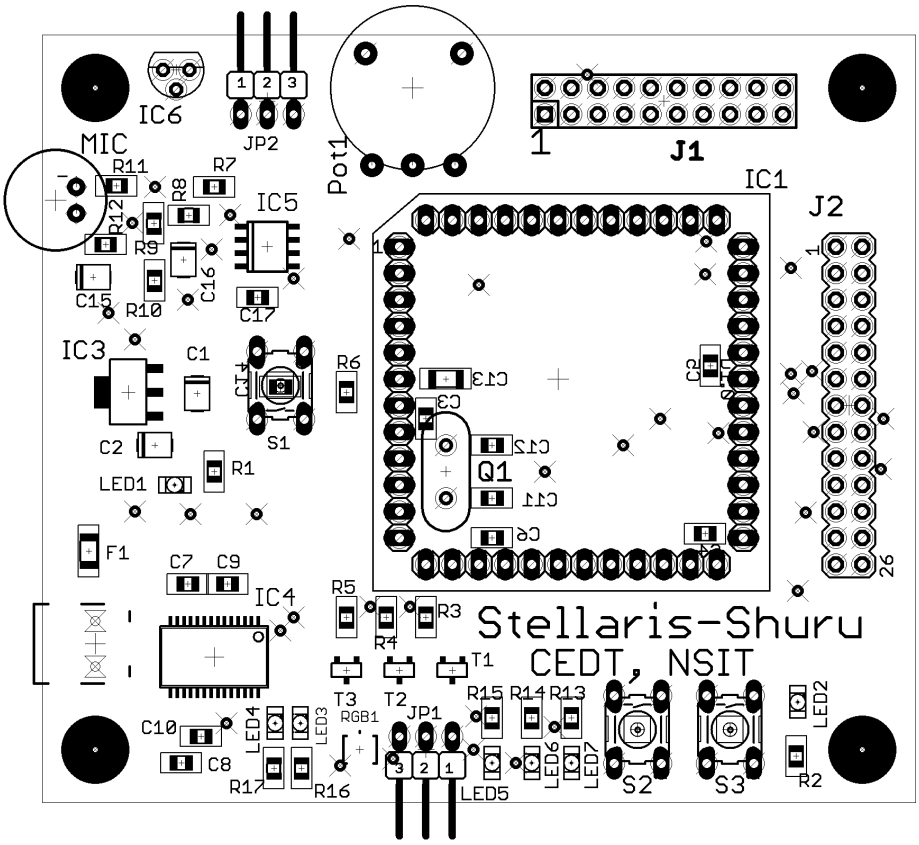
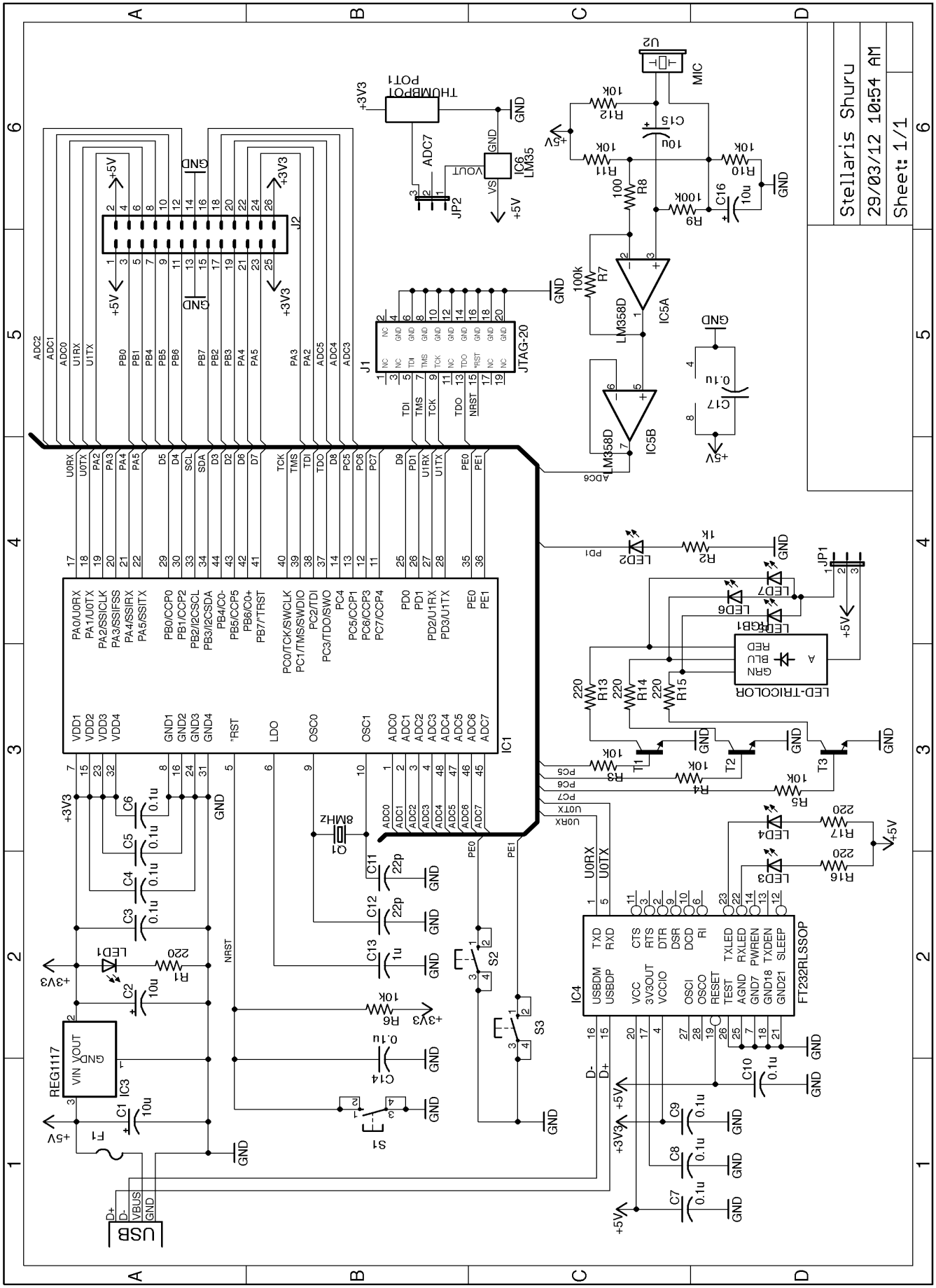


Figure 10.2 Board layout of the Stellaris Shuru Kit (component side)

10.3 Schematic



11. Components of the Kit

This section divides the schematic from the last section into modules and explains the purpose and operation of each module and the hardware design.

11.1 Power Supply

The power supply section of the kit is shown below in figure 11.1. Power is drawn directly from the USB bus of a computer, which is at +5V potential. This voltage is then fed to a Low Drop Out voltage regulator LM1117-3.3 (IC3) to generate +3.3V required by the microcontroller and other kit components. Tantalum capacitors C1 and C2 provide input and output filtering respectively. LED1 connected in series with the resistor R1 provides visual indication for power being supplied to the kit.

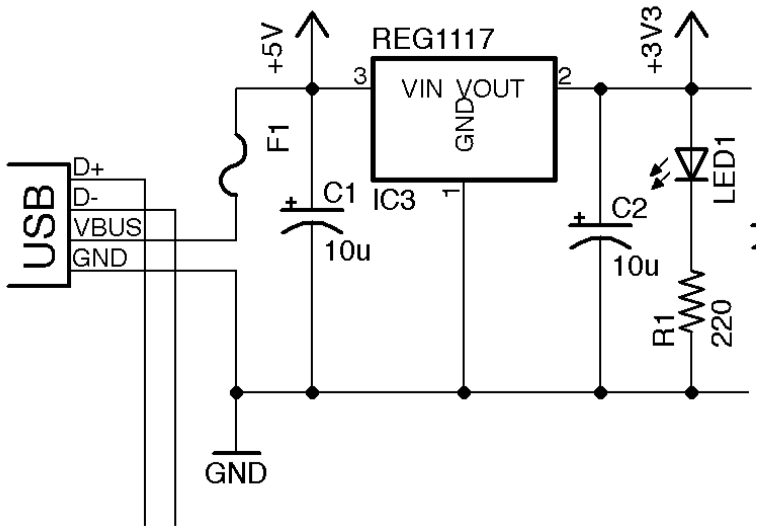


Figure 11.1 Power Supply

11.2 Microcontroller Connections

The connection related to the LM3S608 microcontroller are shown in figure 11.2. The microcontroller (IC1) is powered by the +3.3V output of the LM1117-3.3V LDO regulator connected to the several VCC pins on the chip. Capacitors C3, C4, C5 and C6 act as bypass capacitors for power supply filtering. They are also used as tank capacitors in high speed switching. A crystal (Q1) of 8MHz is used for clocking of the micro controller. Capacitors C11 and C12 of 22pF are used to bias the crystal. A capacitor (C13) of value 1uF is used to bypass the internal LDO of the microcontroller to GND. An active low pushbutton (S1) is connected to the Reset pin of the microcontroller. The required JTAG pins on the microcontroller have been brought out on a 20-pin expansion header (J1) for In-Circuit Debugging. Pins, which are not tied to function on the board, have been brought out on a 26-pin expansion header (J2).

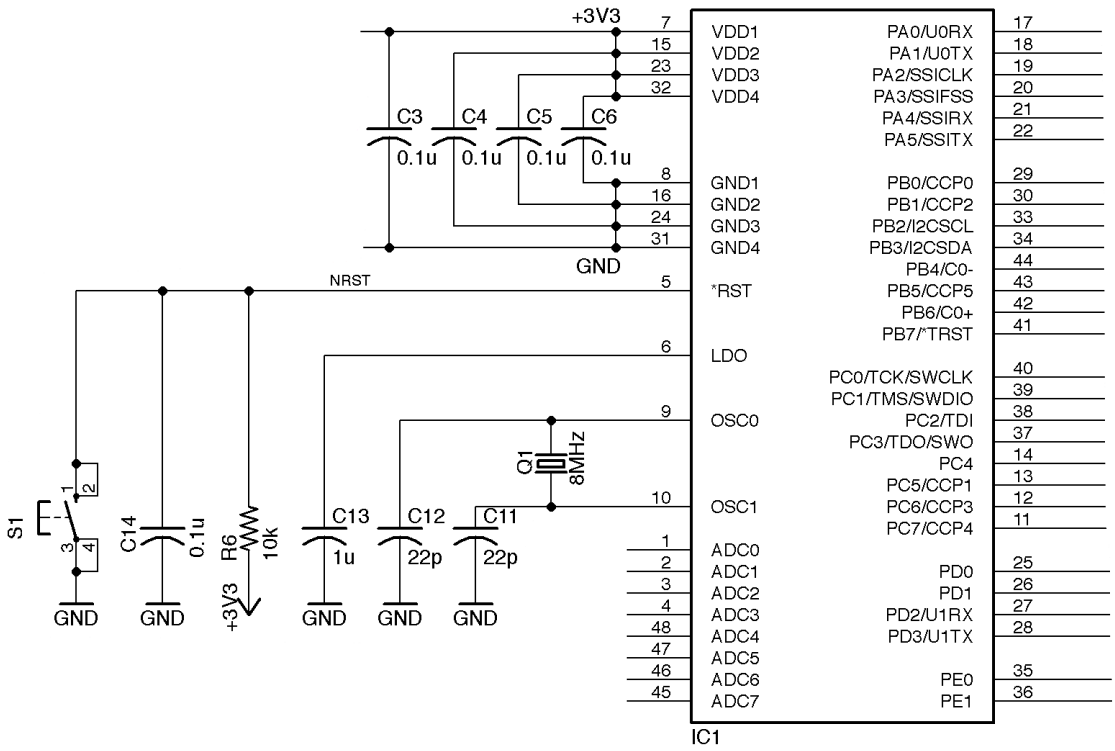


Figure 11.2 Microcontroller Connections

11.3 ARM 20-Pin JTAG Connector

The JTAG pins of the microcontroller have been brought out to a 20-pin ARM compliant JTAG connector. The connections for the JTAG connector are shown in Figure 11.3.

11.4 I/O Expansion Header

The unused I/O ports of the microcontroller, which aren't tied to any function on the board, are brought out on a 26-pin expansion header, which can be used for hardware prototyping. The connection for the 26-pin expansion header are shown in Figure 11.4

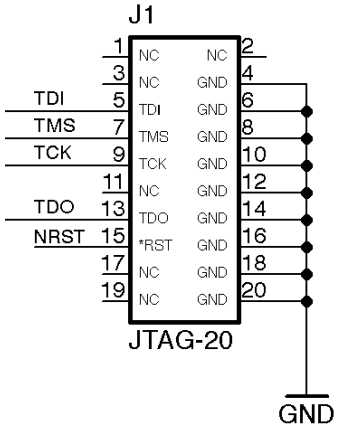


Figure 11.3

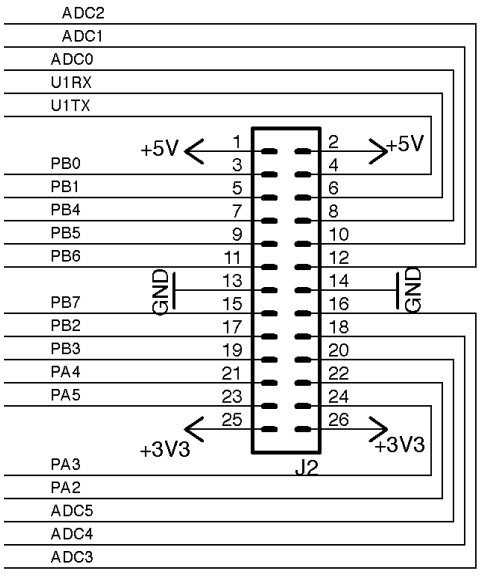


Figure 11.4

11.5 USB Virtual COM Port

The Stellaris Shuru kit uses a FT232RL (IC4) USB-to-Serial bridge connected between UART0 of the microcontroller and the USB. This helps in field programming of the microcontroller over UART using the supplied boot loader. The port can also be used to UART values to a PC application, viz. Hyper Terminal, Bray++, etc. The connections for the FT232RL are shown in Figure 11.5.

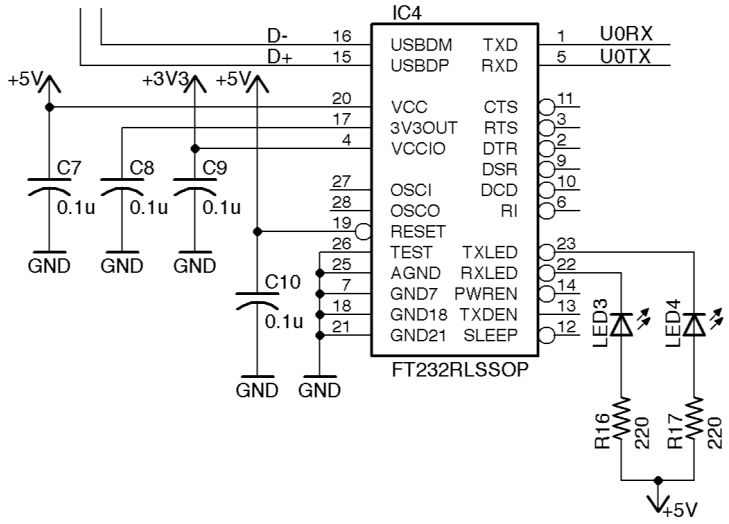


Figure 11.5 USB Virtual COM Port

11.6 Audio Input and Amplifier

The Audio amplifier is a two-stage, ac-coupled microphone amplifier. The microphone that we have used is a capacitive microphone and it requires a biasing voltage of +5V supplied through R12. The supply voltage to the LM358 is +5V and is taken directly from the USB bus. The LM358 is a two-stage amplifier, the first stage is in non-inverting configuration and the other is a voltage follower. Resistors R10 and R11 provide a bias voltage equal to half the supply voltage to input of the first op-amp. But the DC gain of the op-amp is very low, by virtue of high resistance R9 and the capacitor C15. For alternating voltage, the capacitive impedance is low and provides high AC gain determined by resistors R7 and R8. The output of the LM358 is connected to ADC6 of the microcontroller.

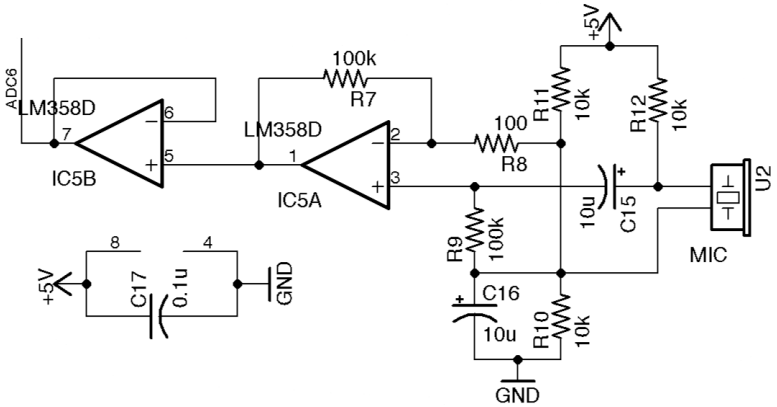


Figure 11.6 Audio Input and Amplifier

11.7 Temperature Sensor and Thumbwheel Potentiometer

To evaluate the analog to digital conversion capabilities of the microcontroller, a LM35 temperature sensor and a thumbwheel potentiometer are provided on the board. The two are connected to the same analog channel, ADC7 of the microcontroller. Either can be selected by using jumper J2. The LM35 is powered by +5V coming from the USB bus. Figure 11.7 shows the circuit.

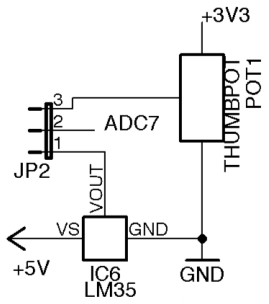


Figure 11.7 Temperature Sensor and Potentiometer

11.8 Light Sensor

The kit uses a LED as a sensor (LED2) to measure light intensity. The LED is reverse biased such that its internal capacitance charges to the voltage applied. It is then connected to an input pin, which measures the time until the voltage across the internal capacitance becomes zero again. This measure of the counter/time is inversely proportional to the light falling on the LED. The circuit is shown in figure 11.8.

11.9 I/O Peripherals – Switches and LEDs

The kit consists of three unicolor LEDs: LED5, LED6 and LED7, one RGB LED: RGB1 and two user pushbutton: S2 and S3. S2 and S3 are connected to PE0 and PE1 respectively. Three NPN BC547 transistors drive the LEDs: T1, T2 and T3, bases of which are connected to PC5, PC6 and PC7 respectively. Choice can be made between the three unicolor LEDs and the RGB LED by placing the appropriate jumper on J1. The anodes of all LEDs are connected to +5V. The LEDs are connected to the Capture/Compare/PWM (CCP) and hence can be used with Pulse Width Modulation for intensity control. The connections are shown in figure 11.9.

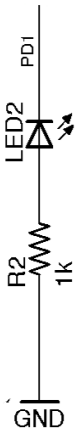
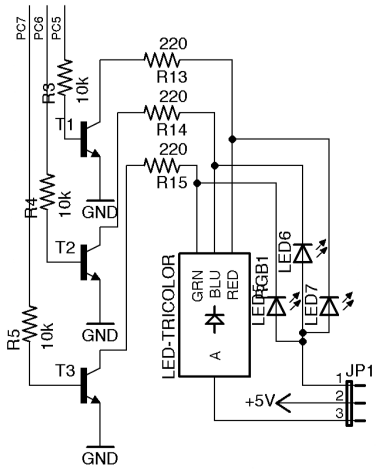


Figure 11.8



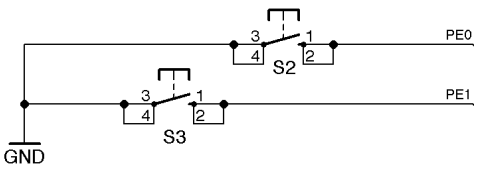


Figure 11.9

12. Stellaris Shuru Component List

Component	Value	Component	Value
R1	220 ohm	F1	PTC Resettable, 500mA
R2	1k ohm	IC1	TI LM3S608 Microcontroller
R3	10k ohm	IC3	LM1117-3.3 LDO Regulator
R4	10k ohm	IC4	FTDI FT232RL
R5	10k ohm	IC5	LM358 Dual Op-amps
R6	10k ohm	IC6	LM35 Temperature Sensor
R7	100k ohm	J1	JTAG 20-Pin
R8	100 ohm	J2	I/O Breakout Pins
R9	100k ohm	JP1	LED Selector
R10	10k ohm	JP2	Analog Sensor Selector
R11	10k ohm	LED1	Red 0805
R12	10k ohm	LED2	Red 0805
R13	220 ohm	LED3	Red 0805
R14	220 ohm	LED4	Red 0805
R15	220 ohm	LED5	Green 0805
R16	220 ohm	LED6	Blue 1206
R17	220 ohm	LED7	Red 0805
C1	10uF/16V Tantalum	POT1	Thumb pot
C2	10uF/16V Tantalum	Q1	8MHz
C3	0.1uF	RGB1	LED Tricolor
C4	0.1uF	S1	10XX Pushbutton - Reset
C5	0.1uF	S2	10XX Pushbutton
C6	0.1uF	S3	10XX Pushbutton
C7	0.1uF	U2	9.7MM Microphone
C8	0.1uF	X1	Mini USB-B
C9	0.1uF		
C10	0.1uF		
C11	22pF		
C12	22pF		
C13	1uF		
C14	0.1uF		
C15	10uF/16V Tantalum		
C16	10uF/16V Tantalum		
C17	0.1uF		

13. Software Tools

Now we come to developing software for your ARM based microcontroller board, the Stellaris Shuru. This and the next section document the entire tool chain installation process and how to acquire the various software components.

In this manual, we describe how to set up your very own ARM tool chain using free and open source tools unbounded by any trial periods or code size limitations.

The tools required to set up your tool chain are:

- The Eclipse Integrated Development Environment (IDE)
- Java Runtime Edition (required by Eclipse IDE).
- Codesourcery G++ Lite cross compilers.
- GNU ARM Eclipse Plugin.
- Texas Instruments Development Package which includes the Stellaris Peripheral Driver Library and LM Flash Programmer required by the Stellaris Shuru development board.
- FTDI Driver for USB-Serial interaction with the target board.

The next section describes how to set up the tool chain on your computer. To successfully set up the IDE, follow each and every step and compare your screen output with the screen snapshots presented in this manual.

14. Setting up the Development Environment

Note: Operating System Required is Windows XP/Vista/7

Before starting with setting up the development environment or tool chain for Cortex M3 microprocessors, we need to first download some of the software modules.

14.1 Downloading Software Modules

1. **Java**
   Install Java from the link: ***http://goo.gl/ekp1M***

2. **Eclipse IDE**:
   Download Eclipse from the link: ***http://goo.gl/9SCXC***
   Move the zip file to "Softwares_Stellaris" folder on your Desktop. Extract it to "C:\Eclipse".

3. **GNUARM Plugin for Eclipse:** Download the plugin from the link: ***http://goo.gl/h5Pj4***
   Move the plugin file to "Softwares_Stellaris" folder on your Desktop.

4. **TI Development Package:** (Stellaris Peripheral Library + FTDI Drivers + LM Flash Programmer): Download the "StellarisWare Driver Library Standalone Package", "Stellaris FTDI driver" and the "LM Flash Programmer" from this link: ***http://goo.gl/PLhVz***
   Install the .exe under "C:\StellarisWare". Install "LM Flash Programmer" under "C:\Program Files".

5. **CodeSourcery G++ Lite:** Download the "Sourcery Code Bench Lite Edition EABI Release" from this link: ***http://goo.gl/za8ta***
   Install the .exe under "C:\Program Files".

6. **FTDI USB Serial Driver:** Download the "VCP Drivers" for supported Architecture from this link: ***http://goo.gl/bPtLX***
   Extract the zip file to "Softwares_Stellaris" folder on your desktop.

14.2 Installation Instructions

14.2.1 Hello Board

Plugging in the board for the first time., the device manager shows up as shown in figure 14.1

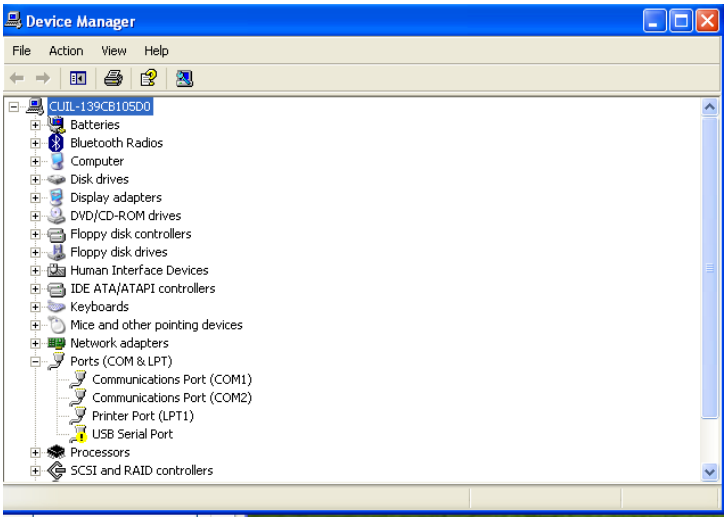


Figure 14.1 Device Manager

In the Ports (COM & LPT) section, an unidentified device named USB Serial Port will show. This is your target board. Install the driver by specifying the location of the "FTDI VCP" subfolder in the Software_Stellaris package. This should enable it to pick up the FTDI driver on its own.

14.2.2 Setting up Eclipse and LM Flash Programmer

i) Fire up the Eclipse IDE. A window similar to that shown in figure 14.2 will crop up.

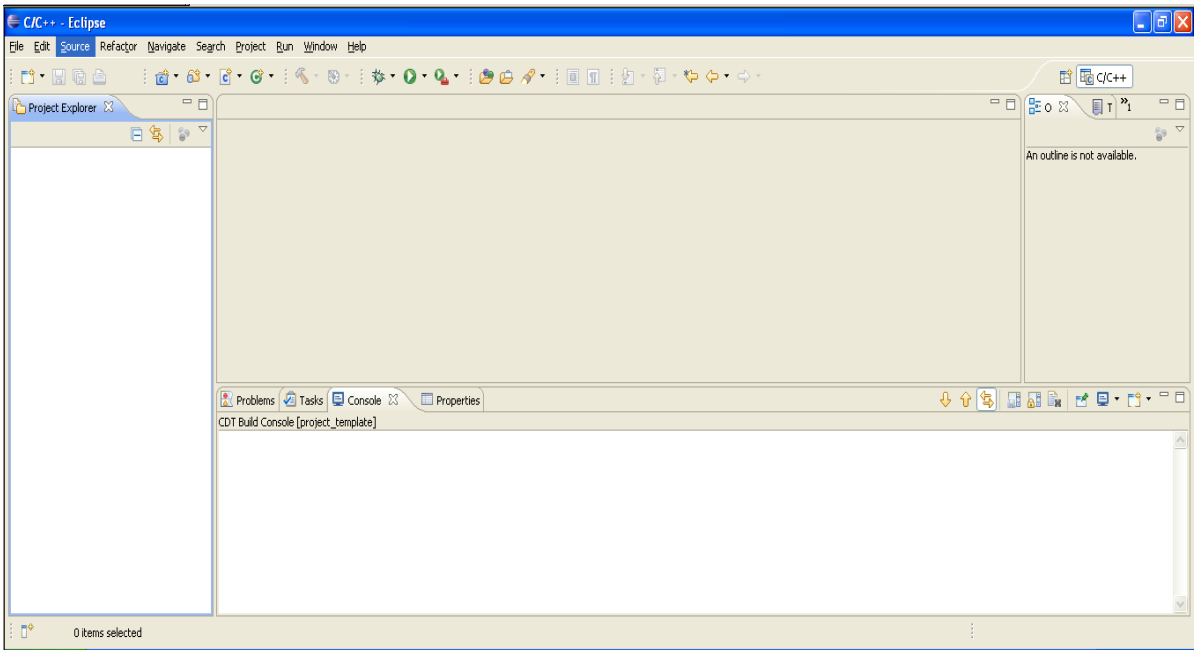


Figure 14.2 Eclipse IDE

ii) Installing the GNU ARM Plugin

- Go to Help -> Install New Software -> Add -> Archive
- A window similar to that shown in figure 14.3 will show.
- Browse to the GNUARM plugin zip file downloaded under “Softwares_Stellaris”.
- Click Ok and Next to install the plugin.
- A window similar to that shown in figure 14.4 will show.

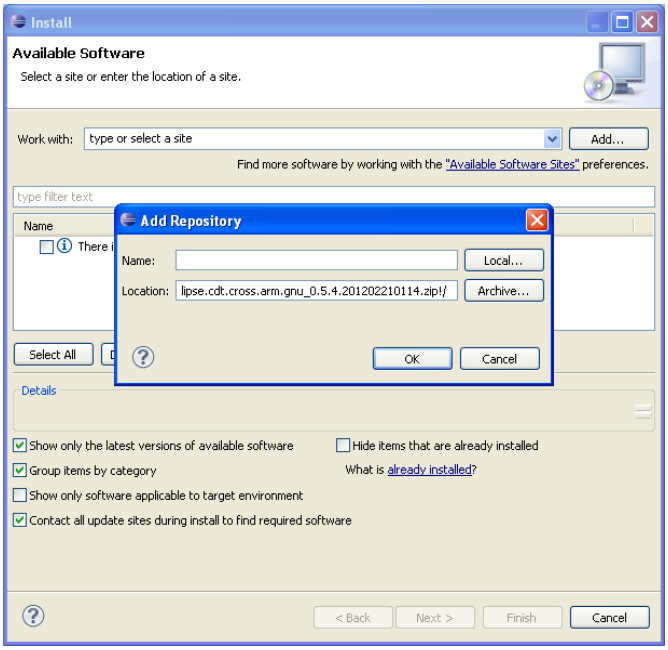


Figure 14.3 Add Repository

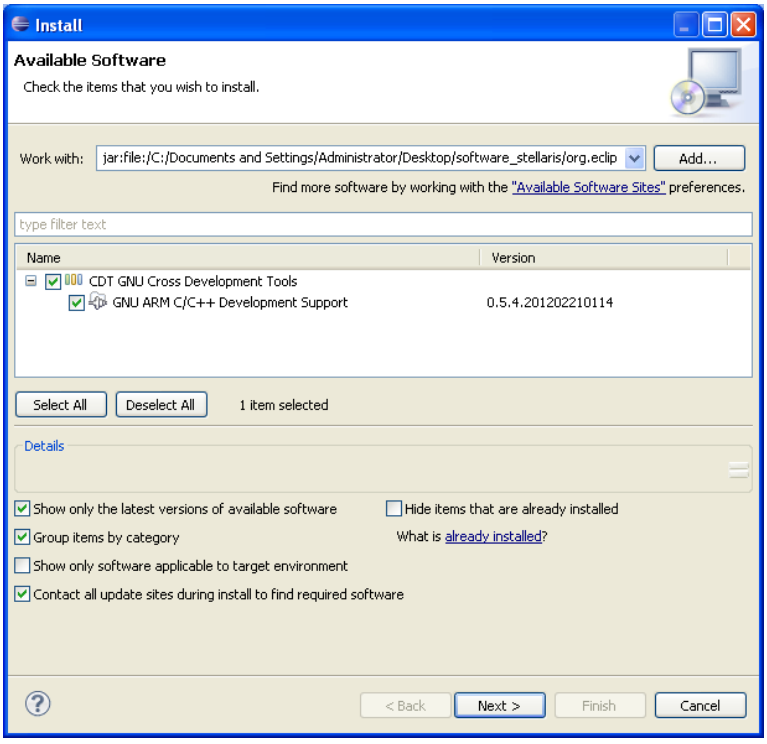


Figure 14.4 Plugin Install Options

iii) Creating your first project

- Go to File -> New -> C Project.
- Enter the project name. Make sure the selections are as shown in figure 14.5
- Click Next.
- Make sure the selection are same as shown in figure 14.6
- Then, press Finish.

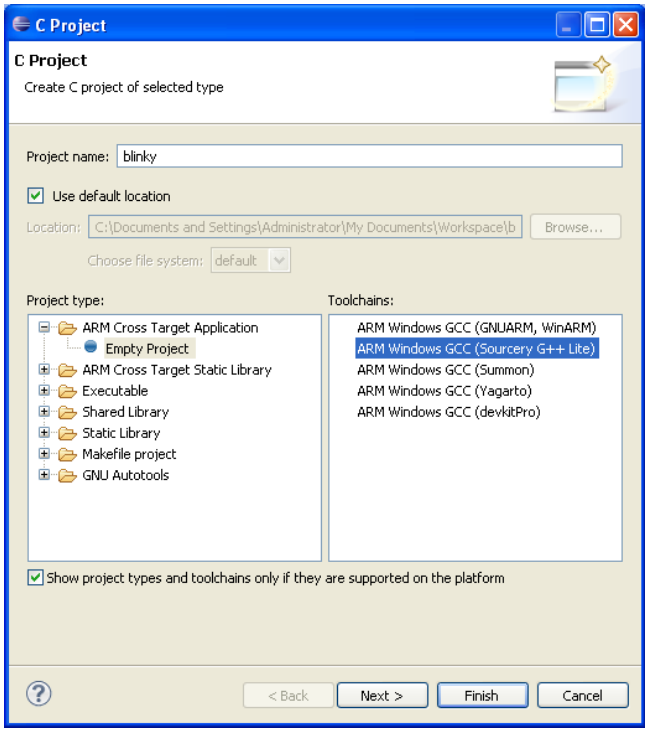


Figure 14.5 New Project Window

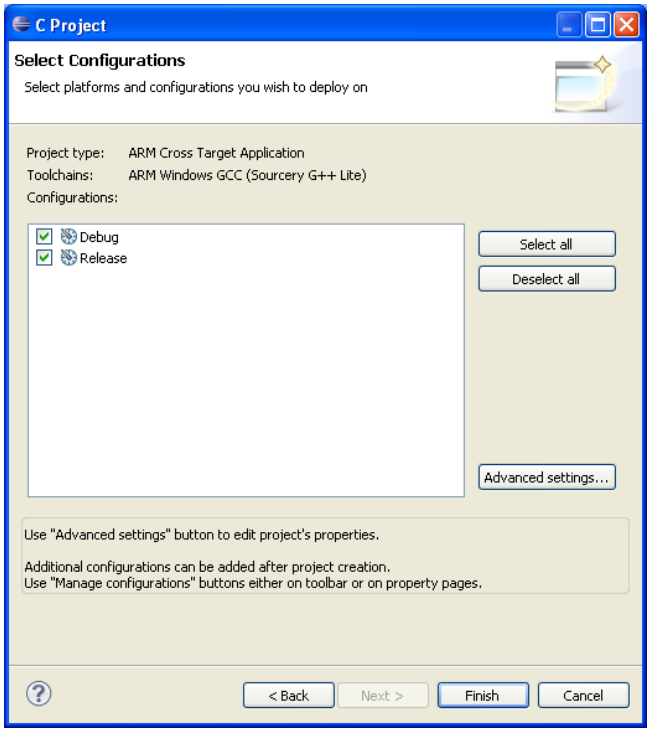


Figure 14.6 Configuration Window

iv) Importing the existing file system.

- Go to File -> Import -> General -> File System.
- A window similar to that shown in figure 14.7 will crop up.
- Press Next.

•Browse and select the any project from the lm3s608 folder under “Softwares_Stellaris” on your desktop. Make sure same option as shown in Figure 14.8 are selected. Press Finish

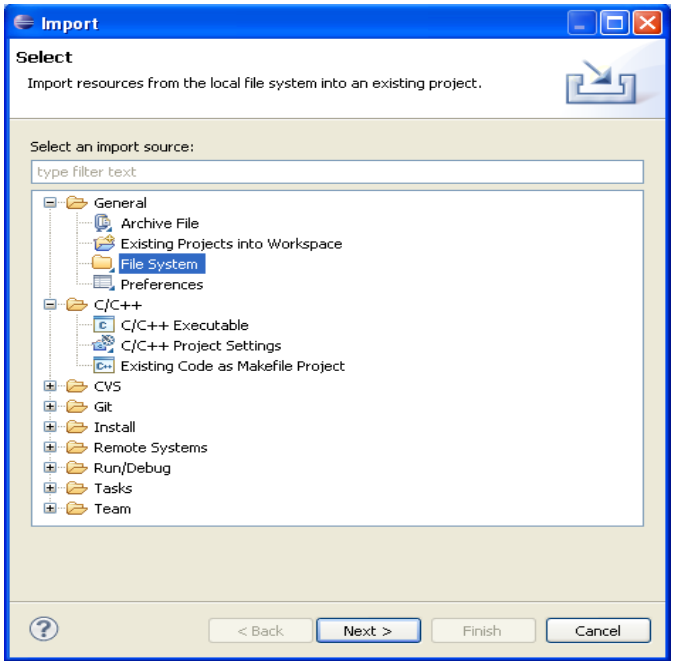


Figure 14.7 Import Window

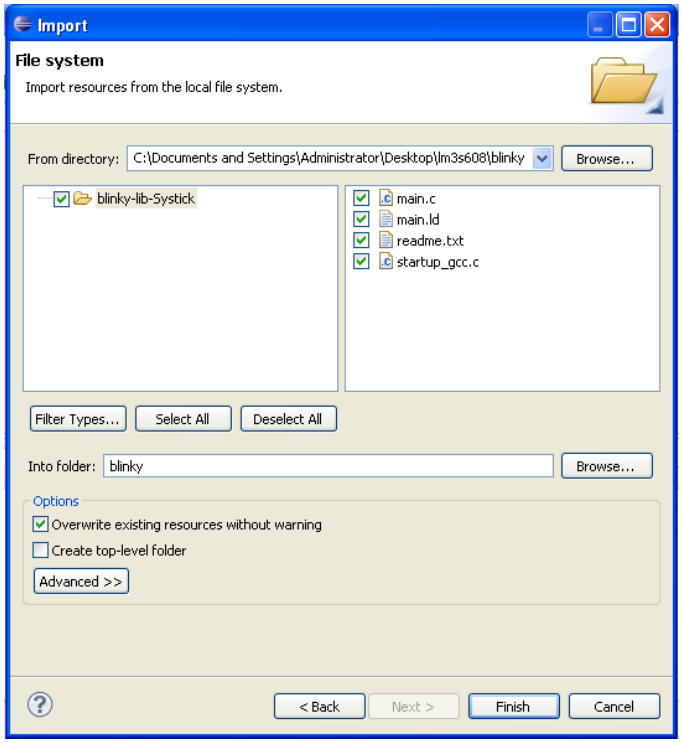


Figure 14.8 File System Window

So till now, we have set up a new project and instructed the Eclipse IDE as to what cross compiler to use for development. Your screen should look similar to that shown in Figure 14.9

.

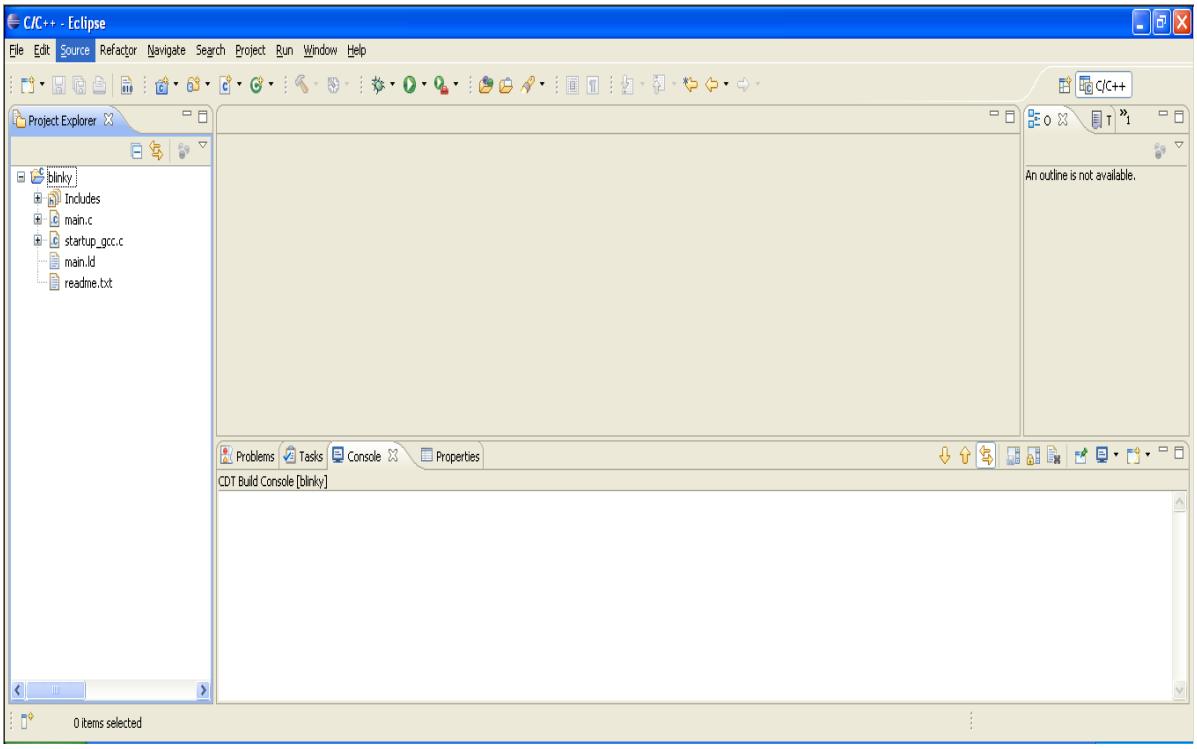


Figure 14.9 Eclipse IDE Window

v) Setting up project properties

- Go to Project Properties.
- Under Settings -> Tool Settings -> ARM Sourcery Windows GCC C Compiler -> Directories. Include the directory paths as shown in figure 14.10 and 14.11.

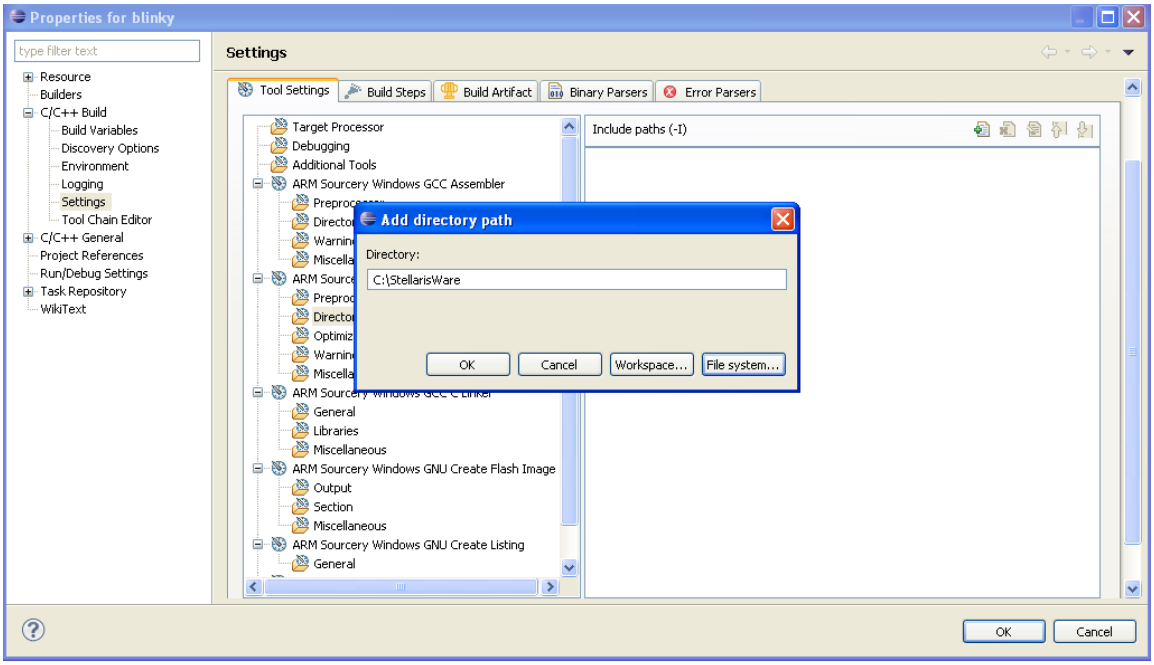


Figure 14.10 Project Properties

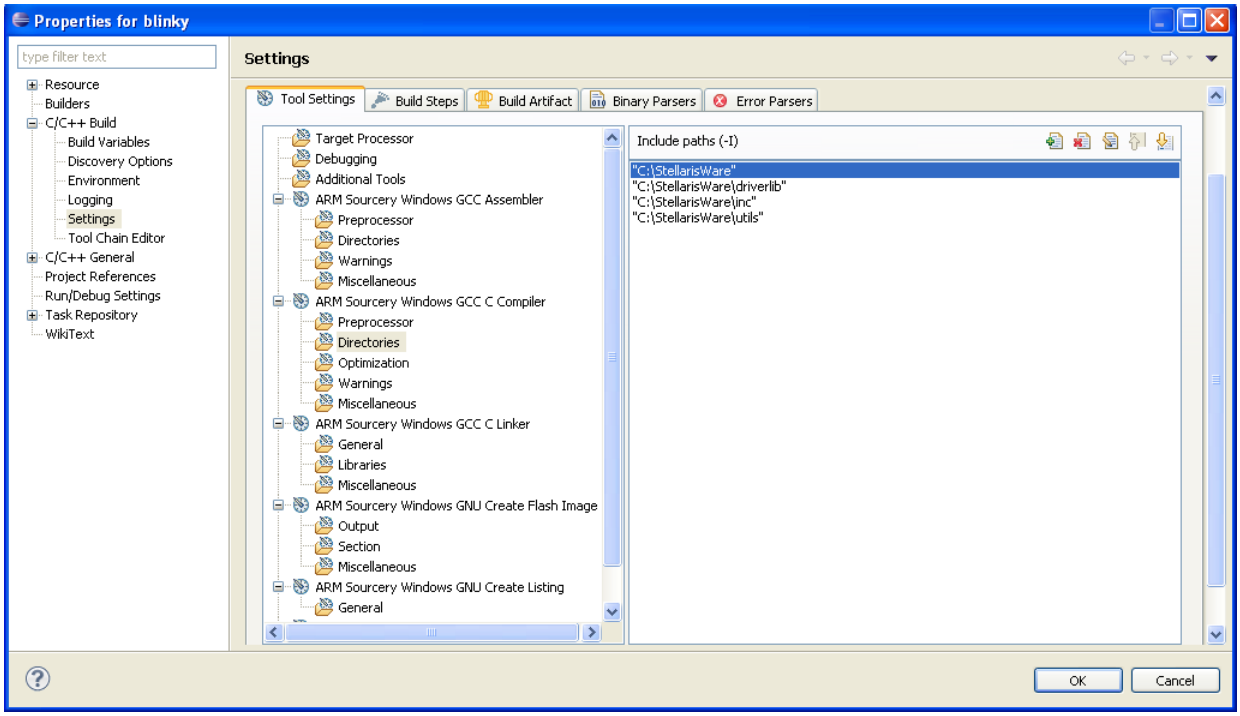


Figure 14.11 Directories

- In the Project Properties window, go to Tool Setting -> ARM Sourcery Windows GCC C Compiler -> Preprocessors and define the symbol “gcc” as shown in figure 14.12.

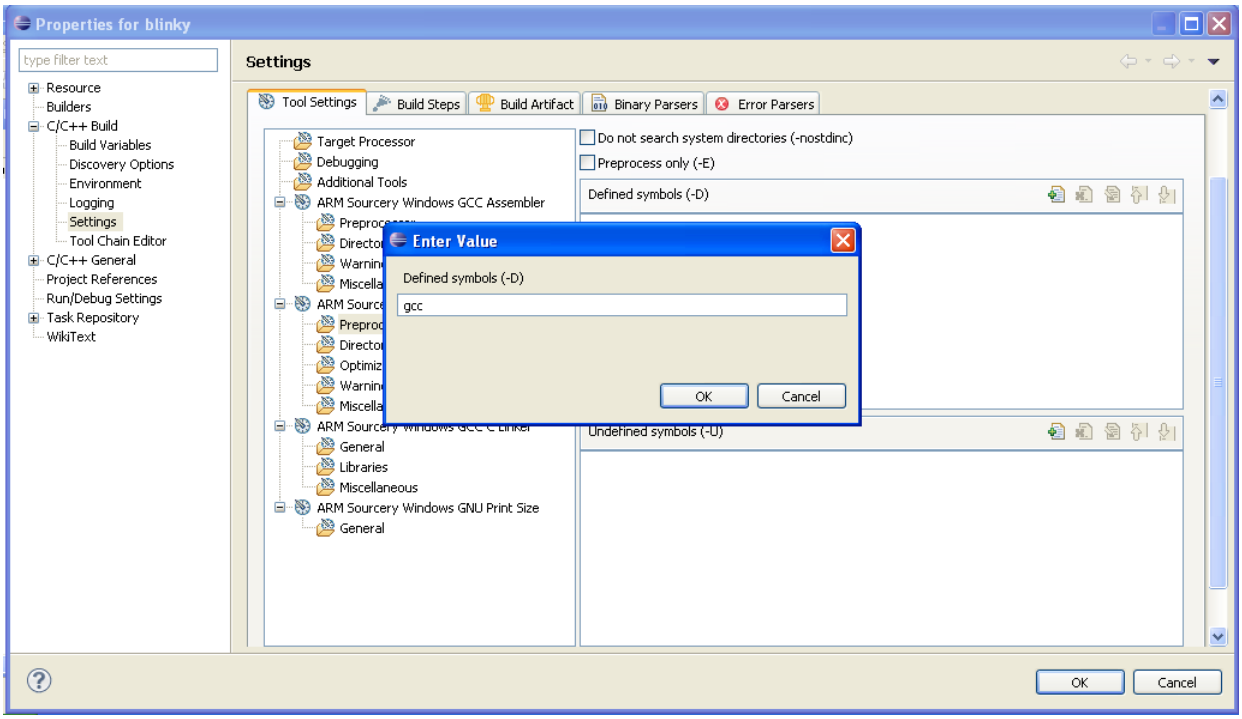


Figure 14.12 Preprocessors

- Setting up code optimization. In the Project Properties window, go to Tool Setting -> ARM Sourcery Windows GCC C Compiler -> Optimization.
- From the drop down menu next to Optimization Levels, select "Optimize Size (-Os)". Figure 14.13.

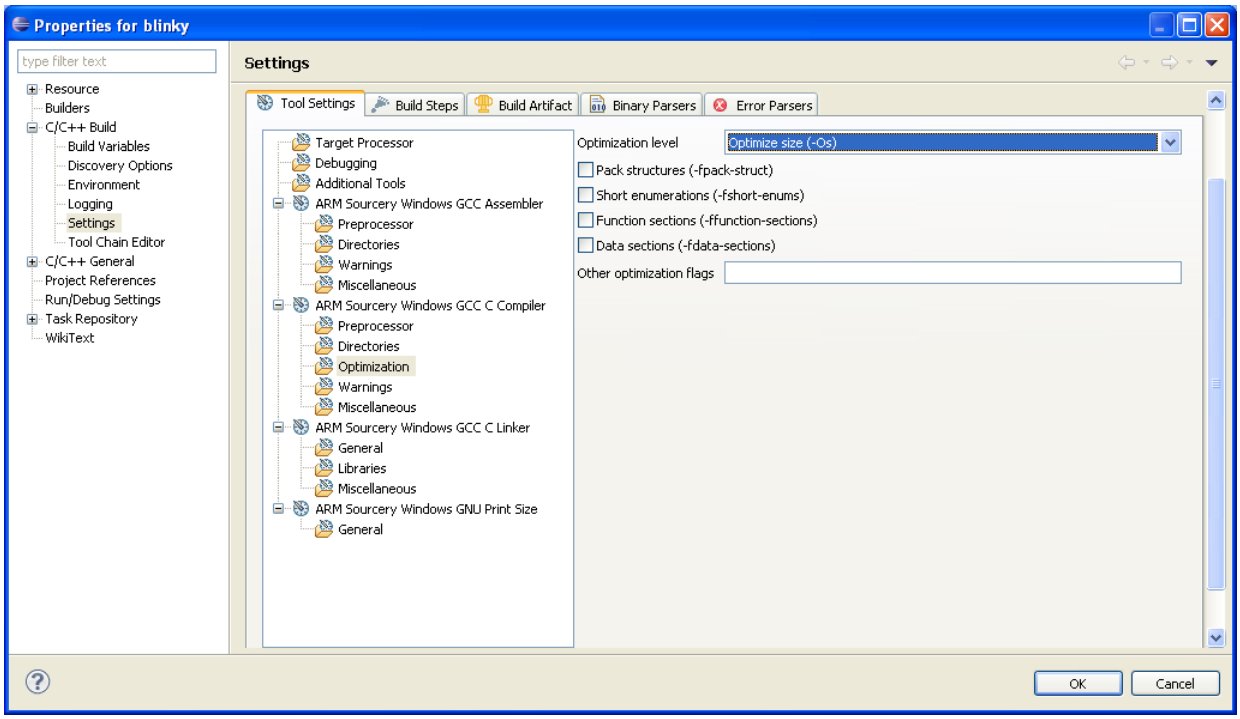


Figure 14.13 Optimization

- Including the linker. In the Project Properties window, go to Tool Setting -> ARM Sourcery Windows GCC C Linker -> General.
- Browse and select the linker file copied to the workspace. In our case, it is "main.ld".
- Make sure the same options are selected as shown in figure 14.14.

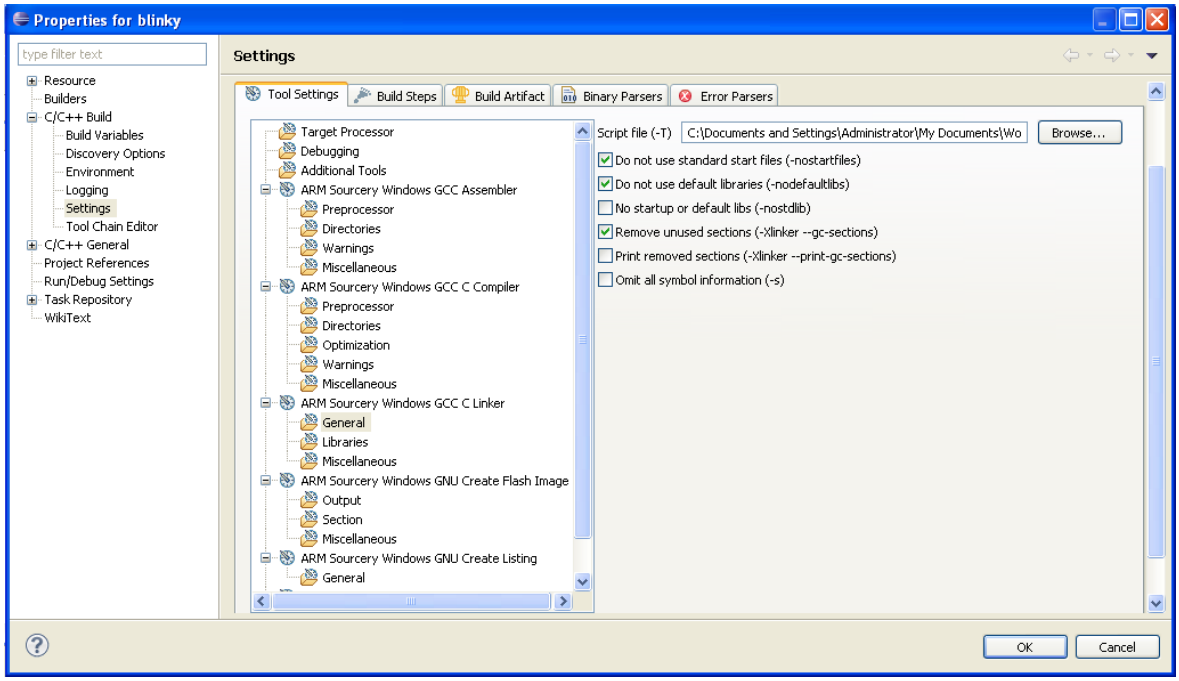


Figure 14.14 Linker File

- Add the libraries. In the Project Properties window go to Tool Setting -> ARM Sourcery Windows GCC C Linker ->Libraries.
- Select Add Directory Path. A window similar to that shown in Figure 14.15 will be displayed.
- Add the two directory paths as shown in figure 14.16.
    - C:/StellarisWare
    - C:/StellarisWare/driverlib/gcc-cm3

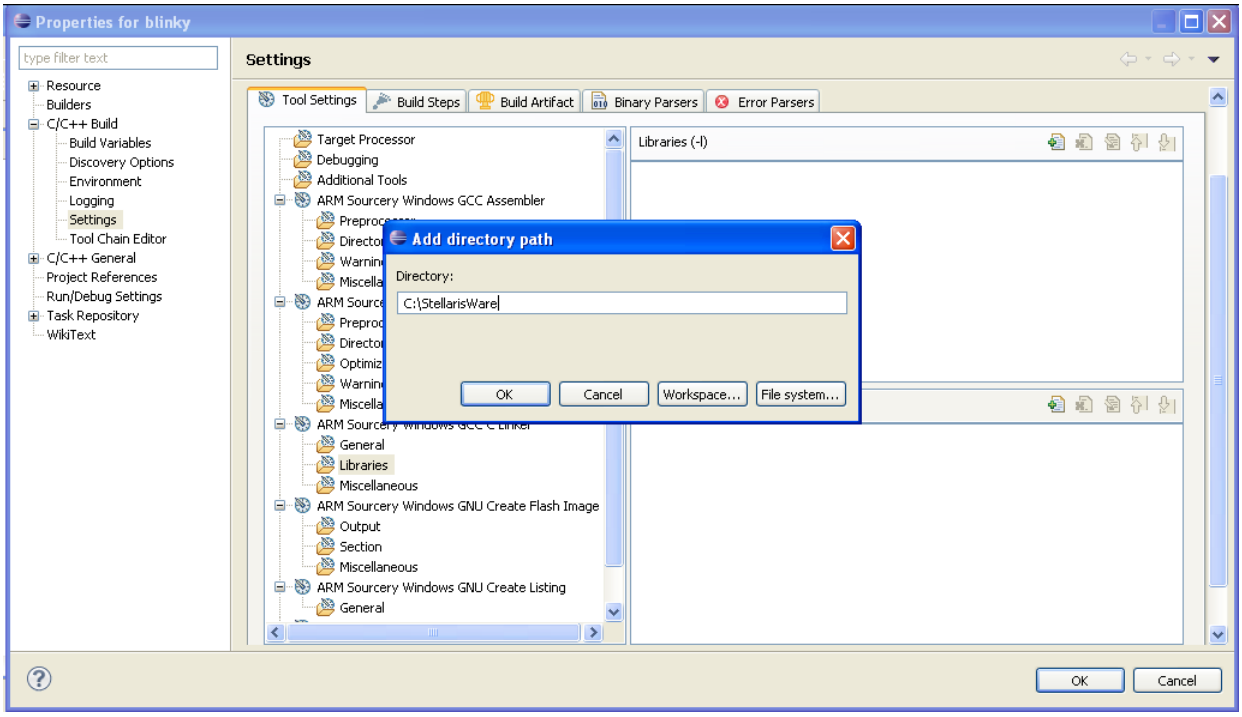


Figure 14.15 Add Directory Path

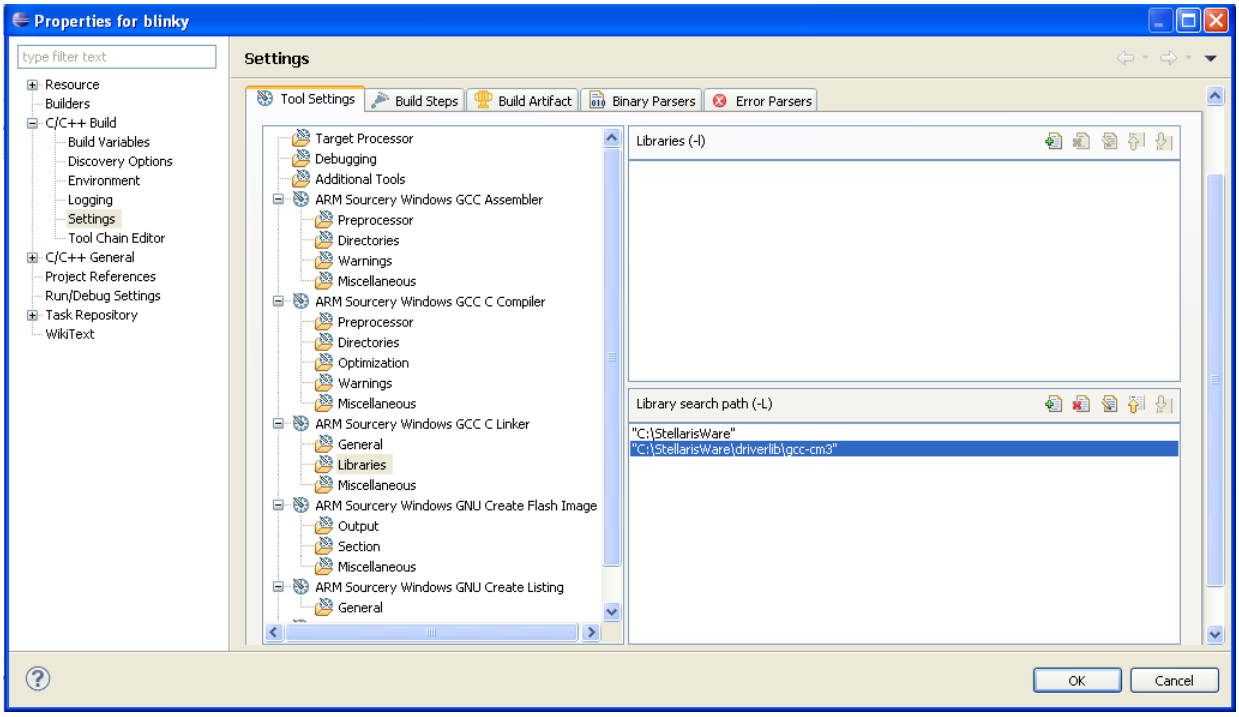


Figure 14.16 Add Directory Path

- Include source location. In the Project Properties window, go to C/C++ General -> Paths and Symbols -> Source Location -> Link Folder. Include the *driverlib* and *utils* folder, which contain the C source files. These are placed inside the StellarisWare folder in the C: drive. Shown in figure 14.17 and 14.18
- Include the files as shown in Figure 14.19 and select Apply.

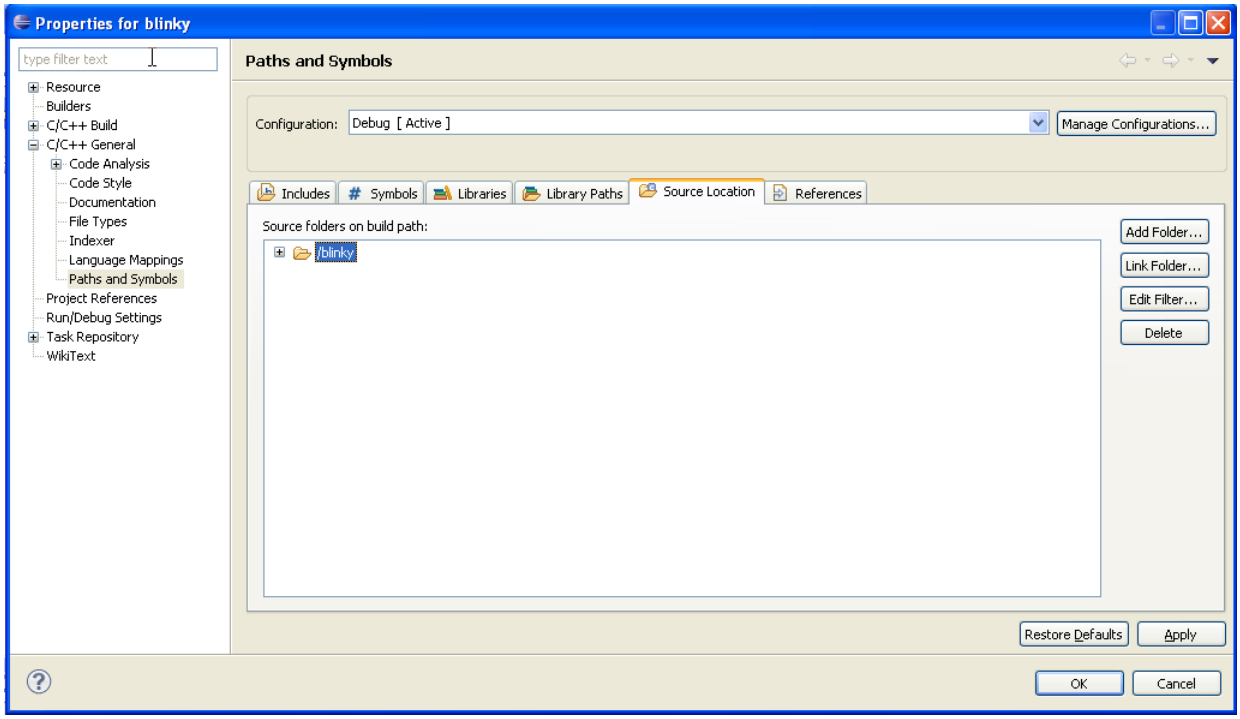


Figure 14.17 Paths and Symbols

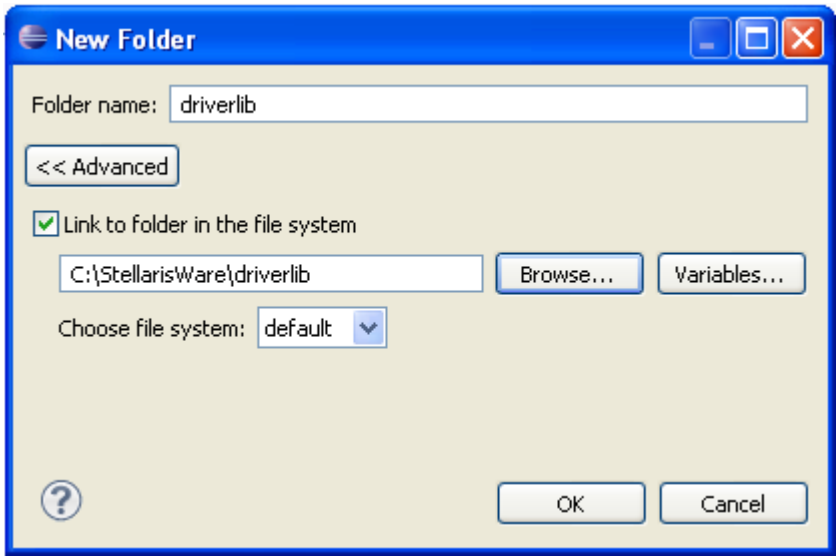


Figure 14.18 Link Folder

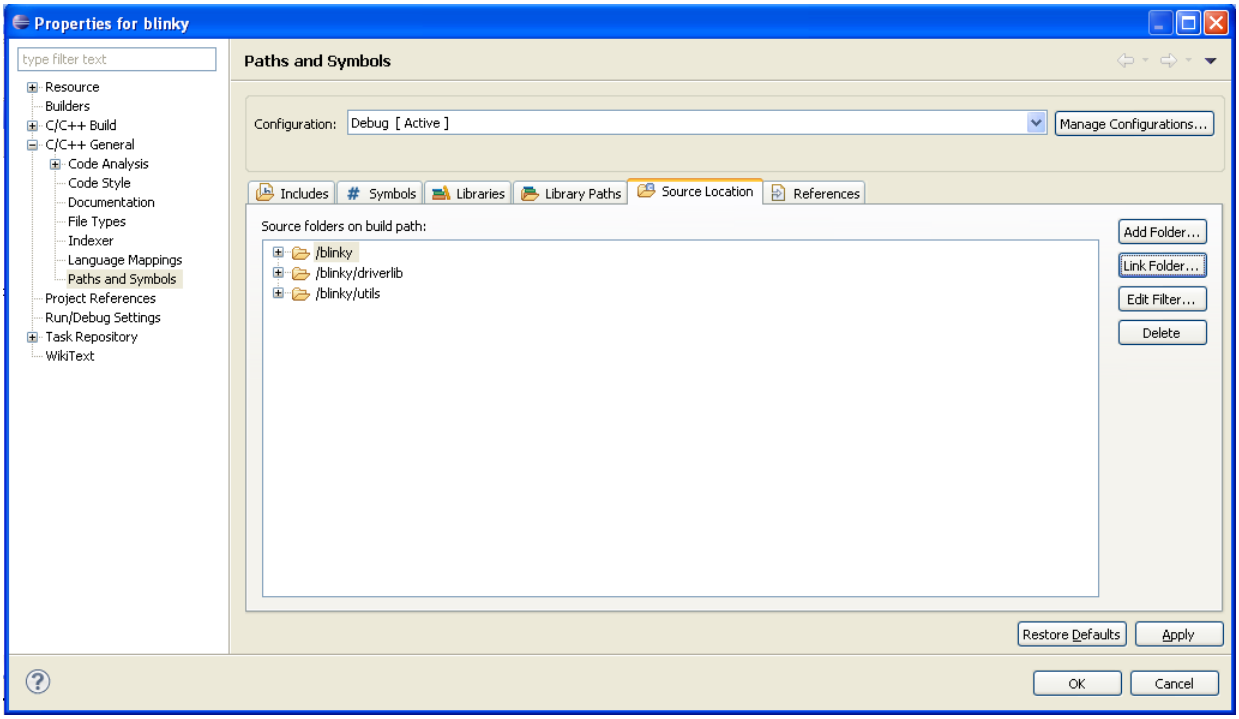


Figure 14.19 Paths and Symbols

- Post Build Steps. In the Project Properties window, go to C/C++ Build -> Settings -> Build Steps -> Post Build Steps Command and type the following:

  arm-none-eabi-objcopy -S -O binary "${ProjName}.elf" "${ProjName}.bin"

- Your windows should look similar to that shown in figure 14.20. Press OK.

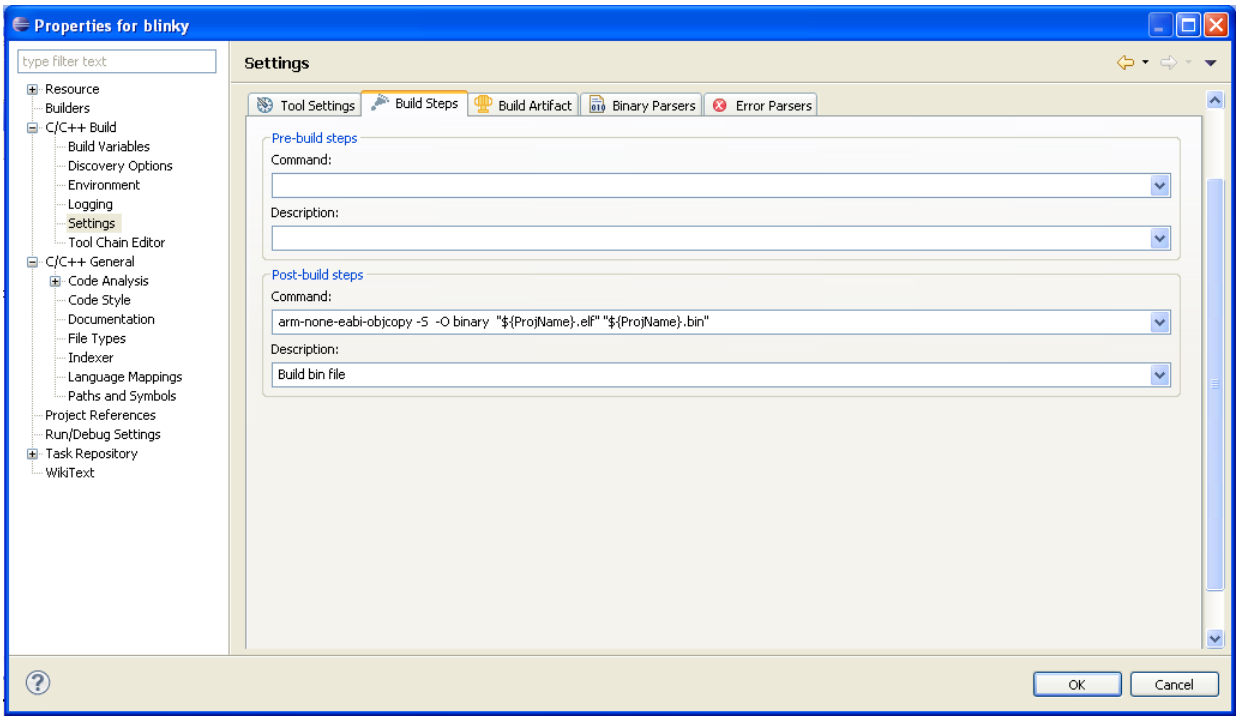


Figure 14.20 Post Build Steps

- After returning back to the main Eclipse IDE window, in your workspace, select main.c
- Your screen should look similar to that shown in Figure 14.21

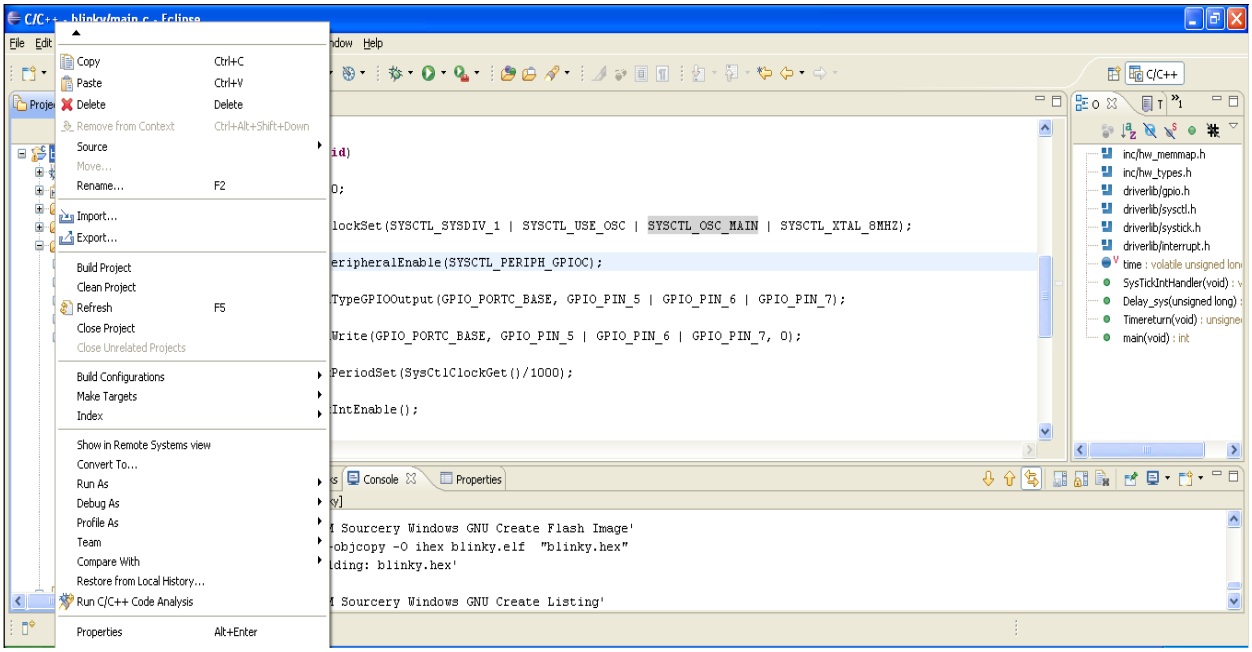


Figure 14.20 Main Eclipse IDE Window

- Right click on the project opened in the workspace, go to Build Configurations and build and clean the project. Shown in figure 14.21. If everything goes right, it will build and clean with ease.

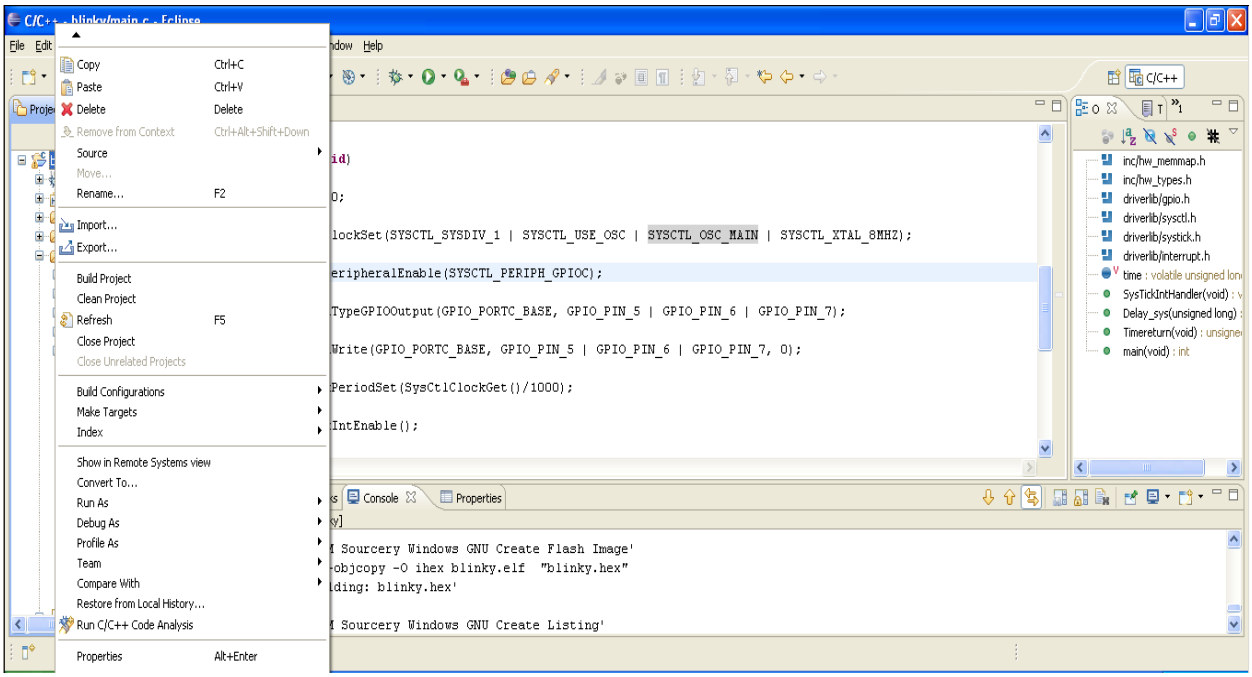


Figure 14.21 Build and Clean Project

vi) Integrating LM Flash Programmer with Eclipse IDE

- Go to Run -> External Tools -> External Tools Configuration as shown in Figure 14.22
- A window similar to that shown in Figure 14.23 should be displayed.
- In the window, set the location to LMFlash.exe located under C:/Program Files/Texas Instruments
- Clicking Run, launches the LM Flash Programmer.

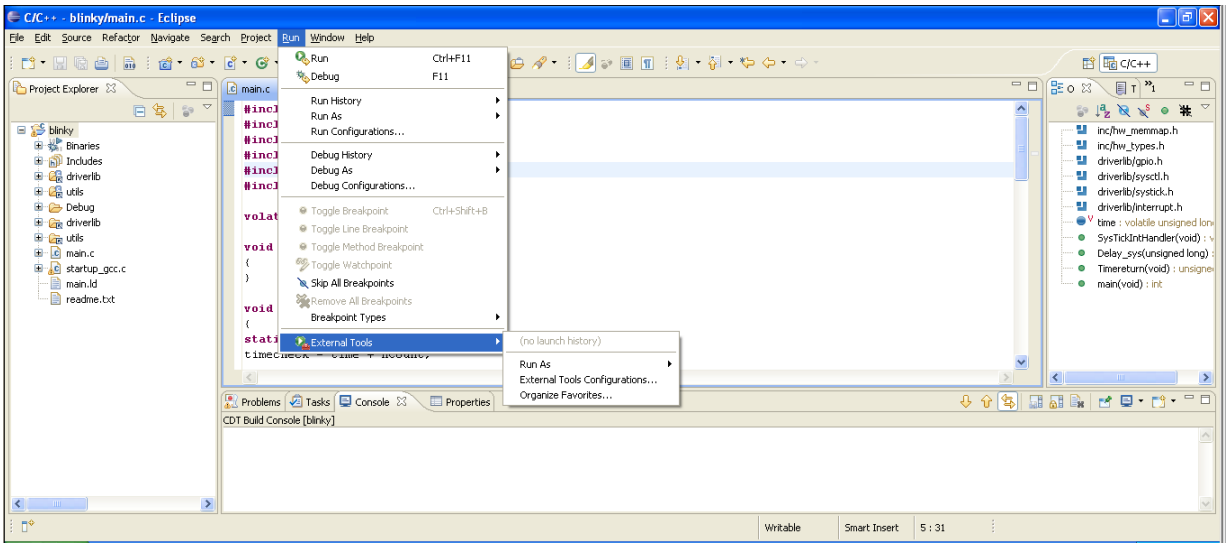


Figure 14.22 External Tool

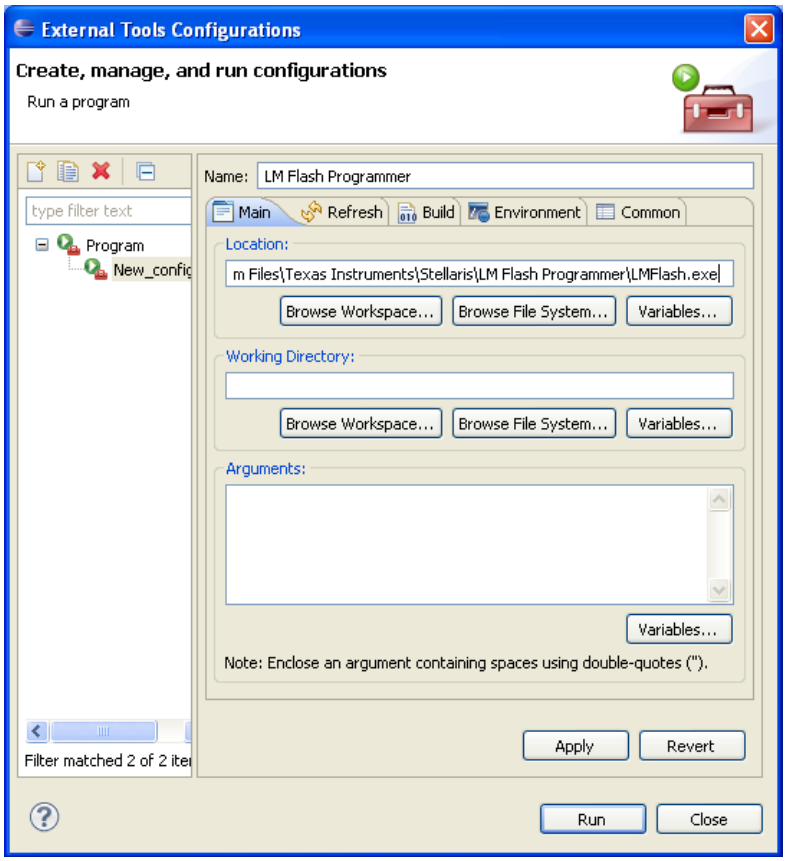


Figure 14.23 External Tools Configuration

14.2.3 LM Flash Programmer

- After clicking on Run, a window similar to that shown in figure 14.24 should display on the screen. This is the LM Flash Programmer.
- In the configuration tab, chose the COM port depending on the port the device is attached to.

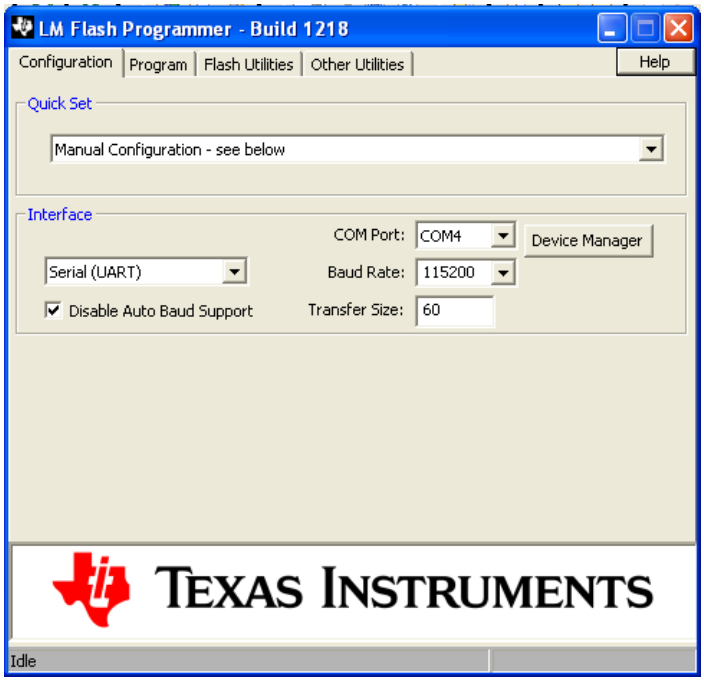


Figure 14.24 LM Flash Programmer (Configuration Tab)

- Go to the Program tab, and select the .bin file by browsing to your project in the Eclipse workspace folder.
- Chose the same options as shown in Figure 14.25 with the same parameters. Make sure the address offset is 0800.

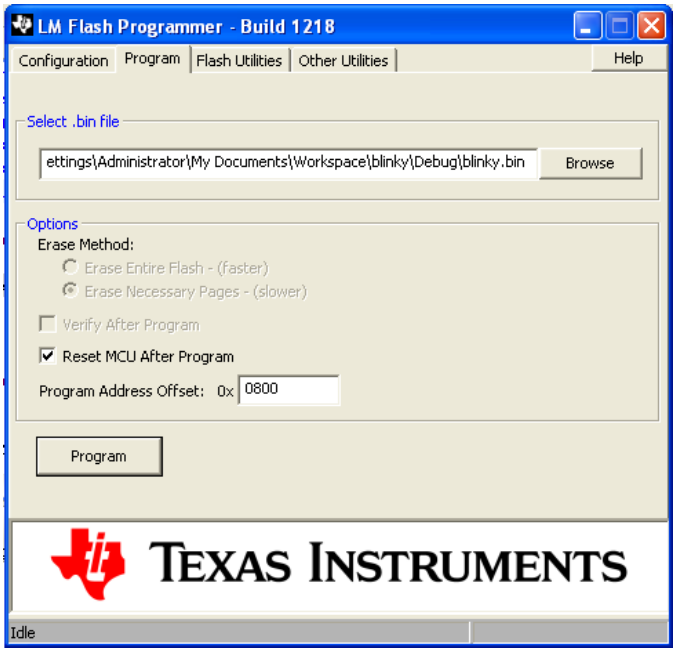


Figure 14.25 LM Flash Programmer (Program Tab)

- Clicking on Program starts transferring the application binary file to the Stellaris Shuru board as shown in Figure 14.26.

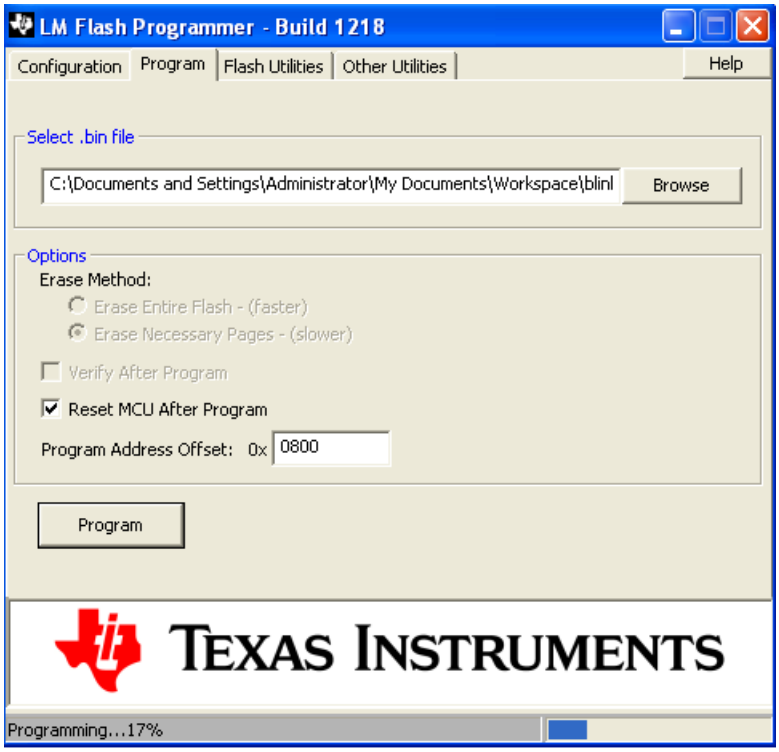


Figure 14.26 Programming the Target

- If everything goes right, you have just programmed your very first program on an ARM powered microcontroller. Congratulations!

15. List of Possible Experiments

We have compiled a small to-do-experiment list around the Stellaris Shuru kit, which helps in evaluating each and every component on the board and every feature available on the Stellaris ARM core like timers, watchdog, brownout detector etc. Of course, a lot more experiments with the kit can be done apart from those listed.

1. Blink a LED.
2. Blink a LED using delay generated using the SysTick timer.
3. Blink LEDs in a controlled pattern.
4. System clock real time alteration using the PLL modules.
5. Control intensity of a LED using PWM implemented in software.
6. Control intensity of a LED using PWM implemented in hardware.
7. Control intensity of RGB LED to generate composite colors using PWM implemented in software.
8. Control intensity of RGB LED to generate composite colors using PWM implemented in hardware.
9. Control a LED using a switch by polling method.
10. Control a LED using a switch by interrupt method and flash the LED once every five switch presses.
11. UART Echo Test.
12. Control intensity of LED on parameters received over UART.
13. Take analog readings on rotation of a rotary potentiometer connected to an ADC channel.
14. Temperature indication on RGB LED.
15. Display temperature on PC by sending values over UART.
16. Mimic light intensity sensed by the light sensor by varying the blinking rate of a LED.
17. Plot light intensity sensed by the light sensor on PC.
18. Evaluate the various sleep modes by putting core in sleep and deep sleep modes.
19. System reset using the Watchdog timer in case something goes wrong.
20. Sample sound using a microphone and display sound levels on LEDs.
21. Filter the sound input using three filters of high, medium and low frequencies and show output on LEDs, one for each filter.
22. Sample sound and plot amplitude v/s time on PC.
23. Sample sound and analyze its spectrum using FFT and plot amplitude v/s frequency results on a PC.
24. Implement a Real time clock using the 32-Bit timers and output time over UART.

From the above list, we have included two experiments in this module to help you get started quickly with the StellarisWare Peripheral Driver Library. Experiment numbers one and eleven have been explained in the next section.

16. Sample Experiments

16.1 Blink a LED

/*This experiment blinks the LED connected on pin PC5 at a constant rate with delay of one second using the System Control function*/

```
/* Defines the base address of the memories and peripherals */
#include "inc/hw_memmap.h"

/* Defines the common types and macros */
#include "inc/hw_types.h"

/* Defines and Macros for GPIO API */
#include "driverlib/gpio.h"

/* Prototypes for the system control driver */
#include "driverlib/sysctl.h"

int main(void)
{
    /* Set the clocking to directly run from the crystal at 8MHz */
    SysCtlClockSet(SYSCTL_SYSDIV_1 | SYSCTL_USE_OSC | SYSCTL_OSC_MAIN |
    SYSCTL_XTAL_8MHZ);

    /* Set the clock for the GPIO Port C */
    SysCtlPeripheralEnable(SYSCTL_PERIPH_GPIOC);

    /* Set the type of the GPIO Pins as Output on which the LEDs are connected*/
    GPIOPinTypeGPIOOutput(GPIO_PORTC_BASE, GPIO_PIN_5 | GPIO_PIN_6 | GPIO_PIN_7);

    /* GPIO Pins 5, 6, 7 on PORT C initialized to be LOW */
    GPIOPinWrite(GPIO_PORTC_BASE, GPIO_PIN_5 | GPIO_PIN_6 | GPIO_PIN_7, 0);

    while(1)
    {
        /* Make Pin Low */
        GPIOPinWrite(GPIO_PORTC_BASE, GPIO_PIN_5, 0);

        /* Delay for a second using System Control function*/
        SysCtlDelay(SysCtlClockGet());

        /* Make Pin High */
        GPIOPinWrite(GPIO_PORTC_BASE, GPIO_PIN_5, GPIO_PIN_5);

        /* Delay for a second using System Control Function */
        SysCtlDelay(SysCtlClockGet());
    }
}
```

16.2 UART Echo Test

/*This experiment evaluates the basic UART functionality. If the character 'a' is received over UART, the controller responds back with the character 'b'. */

```
/* Defines the base address of the memories and peripherals */
#include "../../../inc/hw_memmap.h"

/* Defines the common types and macros */
#include "../../../inc/hw_types.h"

/* Defines and Macros for GPIO API */
#include "../../../driverlib/gpio.h"

/* Prototypes for the system control driver */
#include "../../../driverlib/sysctl.h"

int main(void)
{
    /*Set the clocking to directly run from the crystal at 8MHz*/
    SysCtlClockSet(SYSCTL_SYSDIV_1 | SYSCTL_USE_OSC | SYSCTL_OSC_MAIN |
SYSCTL_XTAL_8MHZ);

    /*Enable the GPIO port to which the UART0 module is attached and enable the UART0
    module*/
    SysCtlPeripheralEnable(SYSCTL_PERIPH_GPIOA);
    SysCtlPeripheralEnable(SYSCTL_PERIPH_UART0);

    /* Make the GPIO pins be UART peripheral controlled. */
    GPIOPinTypeUART(GPIO_PORTA_BASE, GPIO_PIN_0 | GPIO_PIN_1);

    /*Set UART configuration to run at 115.2kbps, one byte length, no stop bits and parity none
    and clock equal to the System clock.*/
    UARTConfigSetExpClk(UART0_BASE, SysCtlClockGet(), 115200
            (UART_CONFIG_WLEN_8 | UART_CONFIG_STOP_ONE |
             UART_CONFIG_PAR_NONE));

    while(1)
    {
        */If 'a' is received over UART, reply back with a 'b'*/
        if(UARTCharGet(UART0_BASE)=='a')
            UARTCharPut(UART0_BASE,'b');
    }

}
```

17. Further Reading

If you wish to read more about the ARM Processor Family and specifically more about the Cortex-M3, we recommend you some good reading material in no specific preference order. These books and articles have been used in the development of this manual too.

1. *"Definitive Guide to the ARM Cortex M3",* Joseph Yiu, Second Edition, 2010. Elsevier Inc.
2. *"ARM System-On-A-Chip Architecture",* Steve Furber, Second Edition, 2000, Addison Wesley.
3. *"ARM System Developers Guide: Designing and Optimizing System Software",* Andrew N. Sloss, Dominic Symes, Chris Wright, 2004, Elsevier Inc.
4. *"ARM Assembly Language",* William Hohl, 2009, CRC Press
5. *"Embedded System and Real Time Interfacing to the ARM Cortex-M3",* Jonathan M. Valvano, 2011, CreateSpace.
6. *"Stellaris Ware Peripheral Driver Library User Guide",* Build 8555, Texas Instruments

18. Contact Us

In case, you want any help on the ARM processor family, Texas Instruments Stellaris family of microcontroller or any software issues or glitches, which may crop us, you can contact any of the two persons listed below.

- Dhananjay V. Gadre, Associate Professor, NSIT. *dhananjay (dot) gadre (at) gmail (dot) com*
- Shanjit Singh Jajmann, UG, NSIT. *shanjitsingh (at) gmail (dot) com*
- Rohit Dureja, UG, NSIT. rohit (dot) dureja (at) gmail (dot) com

You can also post your queries on the TI e2e Community and get them solved instantly by an expert team of TI design engineers and product enthusiasts and ARM supporters.