TERM PROJECT REPORT

ON

STUDY AND DEVELOPMENT OF AN APPLICATION USING RISC V

Submitted By

DARSHIL MODHA (EC036)

PIYUSH SAINI (EC058)

TITHI SHAH (EC078)

For the term project of

B.Tech.

Sem VII

Under the guidance of

Prof. Nisarg K. Bhatt

Prof. Pinkesh V. Patel

Prof. Ashish B. Pandya

DEPARTMENT OF ELECTRONICS & COMMUNICATION
FACULTY OF TECHNOLOGY
DHARMSINH DESAI UNIVERSITY
NADIAD 387001

CERTIFICATE

This is to certify that the project entitled **STUDY AND DEVELOPMENT OF APPLICATION USING RISC V** is term work carried out in the subject of Term Project is bonafide work of Mr. **DARSHIL MODHA (EC036, ID: 18ECUOG026)**, of B.Tech., Semester VII in the branch of **Electronics & Communication** during the academic year 2021-2022.

Prof. Nisarg K. Bhatt
Project Guide, EC Department

Dr. Purvang D. Dalal
Head, EC Dept.

ACKNOWLEDGEMENT

We would like to express our gratitude to **Prof. Mitesh J. Limachia** who guided us throughout the project. He gave us a goal to work towards as our project wasn't a traditional application-based project. His expertise in the field was extremely helpful as there were a lack of resources. His constant support even during the nonworking hours of the college motivated us to complete our project with passion.

Next, we would like to thank our supporting and understanding Term Project faculties **Prof. Nisarg K. Bhatt, Prof. Pinkesh V. Patel** and **Prof. Ashish B. Pandya**. They provided valuable feedbacks which helped us to improve our project and motivated us throughout the semester.

We would also like to thank our Head of Department **Dr. Purvang D. Dalal** for providing such an innovative and motivating environment. The atmosphere encouraged us to research a lot of new things and motivated us to learn about a completely new architecture.

We would also like to extend our gratitude to **Mr. Ian Jyh-Ming Feng** from Andesight Tech Support who was very responsive to our queries and consistently helped us in finding solutions.

Last but not the least we are very thankful to **Faculty of Technology D.D.U.** for providing a stage where we can enhance and develop our skills and hence grow overall.

TABLE OF CONTENTS

ACKNOWLEDGEMENT…… i

TABLE OF CONTENTS……. ii

ABSTRACT…….. 1

Chapter 1: Introduction……... 2

1.1 RISC V……. 2

1.2 Features of RISC V…….. 5

1.3 RISC V vs ARM…….. **7**

Chapter 2: Simulation Support……... 8

2.1 Simulators Available…… 8

2.2 AndeSight™ and its features……... **8**

Chapter 3: Installation Guide…….. 12

3.1 Download Guide…….. 12

3.2 Setting up the Software…… 13

3.3 Compiling and Execution…… 17

Chapter 4: Microprocessor…….. 21

4.1 NX25(F) Processor Features…… 21

4.2 Block Diagram……. 23

4.3 Pipeline stages and Activities…….. 24

4.4 Support available……. 25

Chapter 5: General Purpose I/O…… 26

5.1 Introduction……. 26

5.2 Special Register Description…… 27

5.3 Programming……... 30

5.4 Interrupt……... 35

Chapter 6: UART…….. 42

6.1 Introduction…….. 42

6.2 Special Register Description…… 44

6.3 UART virtual window……. 53

6.4 Programming……... 53

Chapter 7: Timer…….. 57
7.1 Introduction…….. 57
7.2 Special Register Description…… 57
7.3 Programming…..…… 62
Chapter 8: Case Study…….. 64
8.1 Overview…….…… 64
8.2 Program…… 64
LIMITATIONS……. 73
CONCLUSION……. 74
FUTURE WORK…….. 75
REFERENCES…….. 76

ABSTRACT

RISC V is one of the most recent area of research and development in regards to an ISA. As it is open source, it has become very popular among students and industries alike to develop various applications. RISC V has potential to revolutionize the SoC industry as it has better efficiency in terms of power consumption and chip die area as compared to existing ISAs. Our project aims to provide basic framework and acts as a stepping stone for beginners who want to work with RISC V based processor. This project gives an insight on how to use and write basic program for the peripherals like GPIO, UART and PIT available in the simulator. It also familiarizes any beginner with Eclipse based simulators that are used to program RISC V based processors.

Firstly, we went through the RISC V ISA documentation to acquaint ourselves with the architecture. Next step was to find an environment that could simulate the RISC V based processors for which we chose AndeSight™ (an eclipse-based IDE) because of its satisfactory documentation and responsive technical support team. The team provided us with the processor (NX25F) specific documentation through which we gained knowledge of all peripherals supported on the device. In order to program the peripherals, we looked through the available configuration and header files for the processor. We wrote some basic programs with a few modifications that are presented throughout the report albeit with some simulator specific limitations.

Our work differs from the usual application-based projects. However, it is still a useful resource for those who are new to this ISA. Not only did it strengthen our knowledge of RISC V ISA but also accustomed us to a new IDE. Even with all the features available, the IDE support for RISC V is limited due to a novel, admittedly an ingenious, architecture. Nevertheless, its persistent and rapid growth promises future aspirants a better support.

Chapter 1
Introduction

1.1 RISC V

An open-source architecture:

RISC-V is an open standard instruction architecture based on the RISC principles. Unlike other processors that follow the CISC architecture, for example, 80386, RISC architecture only has a small number of opcodes at the base level.

RISC-V is one such flexible architecture that allows the user to add their own commands to the instruction set because of its open-source nature. Since, most CPU manufacturers like Intel, AMD, ARM, etc. require monetary compensation in return for access to their design ideologies, it becomes extremely difficult academically to learn about microprocessors. RISC-V was made with the goal of academic usability in mind; as it grew, big companies showed interest in it. Like many RISC designs, RISC-V is also a load-store architecture.

Instruction Set Architecture:

A RISC-V ISA is defined as a base integer ISA, which must be present in any implementation, plus optional extensions to the base ISA. A base is carefully restricted to a minimal set of instructions sufficient to provide a reasonable target for compilers, assemblers, linkers and operating systems, and so provides a convenient ISA and software toolchain "skeleton" around which more customized processor ISAs can be built.

There are two primary base integer variants, RV32I and RV64I, which provide 32-bit and 64-bit address spaces respectively. RV32E is a subset variant of the RV32I base instruction set, mainly used for small microcontrollers, and which has half the number of integer registers. And a future variant of 128-bit address space RV128I which has not been implemented on any hardware as of now. One thing to keep in mind is that RV32I is distinct from RV64I and is not a subset.

Terminology:

A RISC-V hardware platform can contain one or more RISC-V compatible processing cores together with other non-RISC-V-compatible cores, fixed-function accelerators, various physical memory structures, I/O devices, and an interconnect structure to allow the components to communicate. A RISC-V core might have additional specialized instruction-set extensions.

- **Core:** A component with an independent instruction-fetch unit.
- **Coprocessor:** A unit attached with RISC-V core that is mostly sequenced by a RISC-V instruction stream, however, it also has some limited autonomy relative to the instruction stream.
- **Accelerator:** A non-programmable fixed-function unit that can operate autonomously but is specialized for certain tasks.

- **Hart**: A hardware-thread or a hart is a RISC-V execution context that contains a full set of RISC-V architectural registers and that executes its program independently from other harts in a RISC-V system.

RISC V Basic RV32I (32bit variant):

For RV32I, the 32x registers are 32 bits wide. Register x0 is hardwired with all bits equal to 0. General purpose registers x1-x31 hold values that various instructions interpret as a collection of Boolean values, or as two's complement signed binary integers or unsigned binary integers. Table 1-1 shows machine instruction format of different instruction types for RISC V ISA. Table 1-2 shows all the possible machine instruction formats of RV32I variant

Table 1-1 Basic Instruction Formats

31 27	26 25	24 20	19 15	14 12	11 7	6 0	
funct7		rs2	rs1	funct3	rd	opcode	R-type
imm[11:0]			rs1	funct3	rd	opcode	I-type
imm[11:5]		rs2	rs1	funct3	imm[4:0]	opcode	S-type
imm[12|10:5]		rs2	rs1	funct3	imm[4:1|11]	opcode	B-type
imm[31:12]					rd	opcode	U-type
imm[20|10:1|11|19:12]					rd	opcode	J-type

RISC V Extensions and other variants:

As mentioned earlier, a RISC-V system can have additional specialized instruction sets. Some of the type of extensions available are mentioned below:

- "Zifencei" Instruction-fetch fence:
  - It includes the FENCE.I instruction that provides explicit synchronization between writes to instruction memory and instruction fetches on the same hart.
- "Zihintpause" Pause hint:
  - The PAUSE instruction is HINT that indicates the current hart's rate of instruction retirement (retirement here means progress of an instruction throughout its instruction cycle time) should be temporarily reduced or paused.
- RV32E Instruction set:
  - This is a reduced version of the RV32I which halves the number of registers from 32 to 16.
- RV64I Instruction set:
  - Basically, it builds upon the RV32I with more instructions and 64bit word length.
- RV128I Instruction set:
  - It's a straightforward extrapolation from the existing RV32I and RV64I designs which supports a flat 128-bit address space. This is just to future proof the architecture in case a 64-bit space is not enough.

Table 1-2 RV32I Instruction Set

imm[31:12]				rd	0110111	LUI
imm[31:12]				rd	0010111	AUIPC
imm[20|10:1|11|19:12]				rd	1101111	JAL
imm[11:0]		rs1	000	rd	1100111	JALR
imm[12|10:5]	rs2	rs1	000	imm[4:1|11]	1100011	BEQ
imm[12|10:5]	rs2	rs1	001	imm[4:1|11]	1100011	BNE
imm[12|10:5]	rs2	rs1	100	imm[4:1|11]	1100011	BLT
imm[12|10:5]	rs2	rs1	101	imm[4:1|11]	1100011	BGE
imm[12|10:5]	rs2	rs1	110	imm[4:1|11]	1100011	BLTU
imm[12|10:5]	rs2	rs1	111	imm[4:1|11]	1100011	BGEU
imm[11:0]		rs1	000	rd	0000011	LB
imm[11:0]		rs1	001	rd	0000011	LH
imm[11:0]		rs1	010	rd	0000011	LW
imm[11:0]		rs1	100	rd	0000011	LBU
imm[11:0]		rs1	101	rd	0000011	LHU
imm[11:5]	rs2	rs1	000	imm[4:0]	0100011	SB
imm[11:5]	rs2	rs1	001	imm[4:0]	0100011	SH
imm[11:5]	rs2	rs1	010	imm[4:0]	0100011	SW
imm[11:0]		rs1	000	rd	0010011	ADDI
imm[11:0]		rs1	010	rd	0010011	SLTI
imm[11:0]		rs1	011	rd	0010011	SLTIU
imm[11:0]		rs1	100	rd	0010011	XORI
imm[11:0]		rs1	110	rd	0010011	ORI
imm[11:0]		rs1	111	rd	0010011	ANDI
0000000	shamt	rs1	001	rd	0010011	SLLI
0000000	shamt	rs1	101	rd	0010011	SRLI
0100000	shamt	rs1	101	rd	0010011	SRAI
0000000	rs2	rs1	000	rd	0110011	ADD
0100000	rs2	rs1	000	rd	0110011	SUB
0000000	rs2	rs1	001	rd	0110011	SLL
0000000	rs2	rs1	010	rd	0110011	SLT
0000000	rs2	rs1	011	rd	0110011	SLTU
0000000	rs2	rs1	100	rd	0110011	XOR
0000000	rs2	rs1	101	rd	0110011	SRL
0100000	rs2	rs1	101	rd	0110011	SRA
0000000	rs2	rs1	110	rd	0110011	OR
0000000	rs2	rs1	111	rd	0110011	AND
0000 | pred	succ	00000	000	00000	0001111	FENCE
0000 | 0000	0000	00000	001	00000	0001111	FENCE.I
000000000000		00000	000	00000	1110011	ECALL
000000000001		00000	000	00000	1110011	EBREAK
csr		rs1	001	rd	1110011	CSRRW
csr		rs1	010	rd	1110011	CSRRS
csr		rs1	011	rd	1110011	CSRRC
csr		zimm	101	rd	1110011	CSRRWI
csr		zimm	110	rd	1110011	CSRRSI
csr		zimm	111	rd	1110011	CSRRCI

- "M" Standard Extension for Integer Multiplication and Division:
  - Adds instructions for integer multiplication and division.

- “A” Standard extension for atomic instructions:
    - Contains instructions that atomically read-modify-write memory to support synchronization between multiple RISC-V harts running in the same memory space.
- “Zicsr” Control and Status register instructions:
    - As the name suggests, this extension adds more CSR instructions to the set.
- “F” Standard extension for single-precision floating-point:
    - This extension adds f0-f31 floating-point registers, a floating-point control and status register **fcsr**, each 32 bits wide.
- “D” Standard extension for double-precision floating-point:
    - Adds double-precision floating-point computational instructions compliant with IEEE 754-2008 arithmetic standard.
- “Q” Standard extension for quad-precision floating-point:
    - Basically, the floating-point registers are now extended to hold either a single, double or quad-precision floating-point value.
- “C” Standard extension for compressed instruction:
    - The standard compressed instruction-set extension which reduces static and dynamic code size by adding short 16-bit instruction encodings for common operations.

RISC V cores and SoC:

Many cores and SoC already exist made by the people all around the world that use the RISC-V architecture with additional specializations. The main advantage is that one can make whatever type of core they want which caters one’s need.

RISC V Emulators:

Emulators are available, commercial and free-ware, to run code written for the RISC-V architecture in languages like Assembly and C.
Some examples are:

- Spike
- QEMU or rv8

These emulate RISC-V harts on an underlying x86 system, and which can provide either a user-level or a supervisor-level execution environment.

1.2 Features of RISC V:

- The RISC V allows the customization of the base ISA and even create a whole new extension. The users can use one of the many available extensions as per their need and hence giving RISC V its flexibility.
- RISC V is a royalty free ISA. This allows its use in academic purposes without having to pay for the ISA.
- RISC V provides better power efficiency and reduces the chip die area in fabrication. The benchmark of various processors in fig. 1.1 shows the comparison of RISC V based processor with other processors using different ISAs.

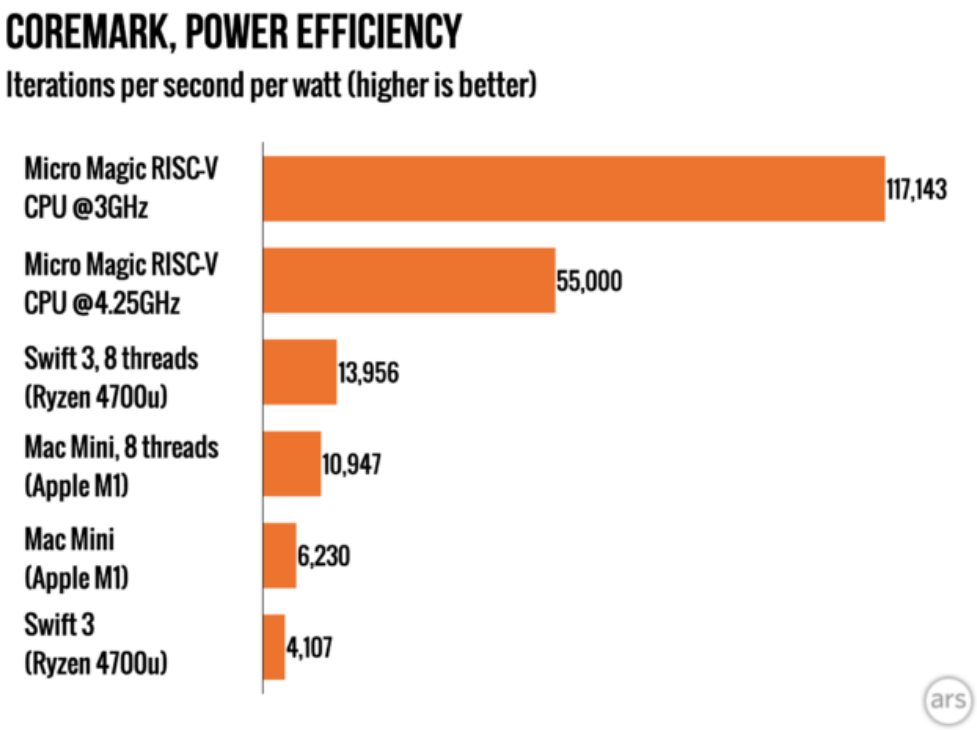


Fig. 1.1 Benchmark for Power Efficiency of Various Processors

- The programs written in RISC V have compact code size in comparison to other famous ISAs. Fig. 1.2 shows this comparison.

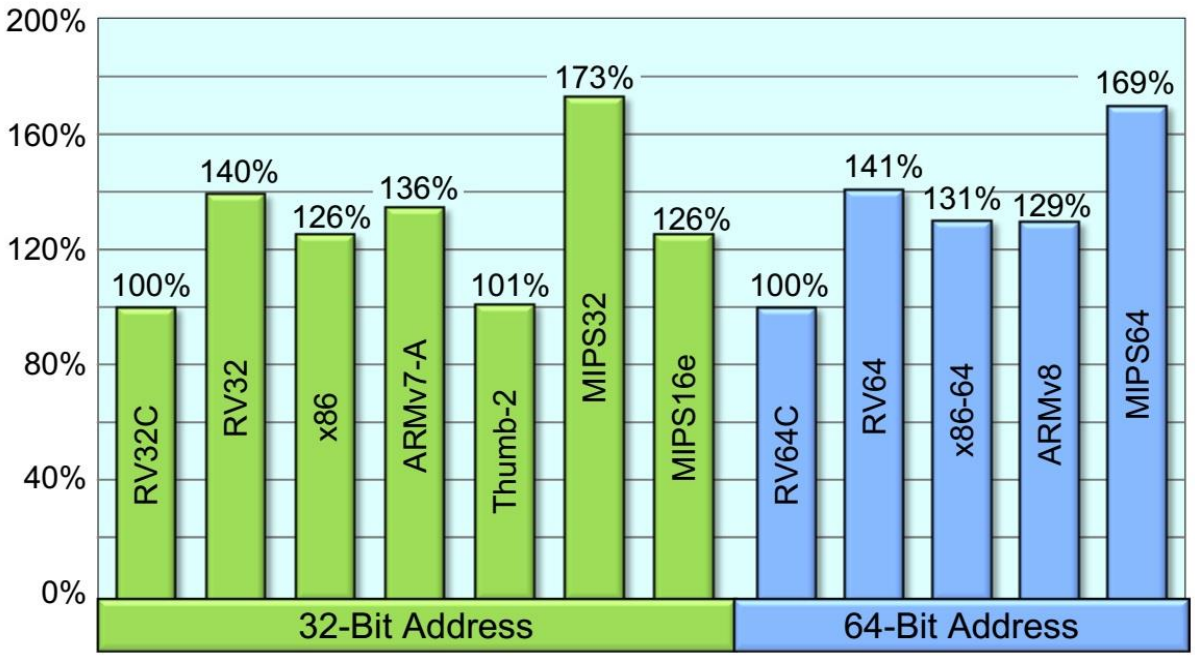


Fig. 1.2 Code Size comparison of different ISAs

1.3 RISC V vs ARM:

At present, ARM based controllers are famous in the embedded application market with x86 leading in the processor market. RISC V needs to compete with both of them in order to gain the attention of the consumers. The table 1-3 shows the difference between RISC V and ARM.

Table 1-3 RISC V vs ARM

RISC V	ARM
Simple, Modular ISA. Additional custom extensions over the base ISA are created for performance enhancement.	More Complex, non-modular ISA. Only recently has ARM started to put efforts in the direction of making its ISA modular.
Smaller and open-source Base ISA leads to easier production of cores	More complex and ISA is only available to its license owners

Chapter 2
Simulation Support

2.1 Simulators available:

- Freedom Studio by SiFive
  - Problem: Only has a single microcontroller support which is made by SiFive. Also, this is under development so it has many bugs which haven't been solved yet. Debugging requires hardware, no direct simulation support.
- Embedded Studio by Segger
  - Problem: Overall less amount of debugging features, no access to SoC registers.
- AndeSight™ by Andestech
  - This is the simulator which was chosen as the appropriate one
  - Reason: Trial version (evaluation version) is available for 90 days with full access to the software and allows creation of programs for premade microcontrollers and microprocessors.

2.2 AndeSight™ and its features:

- Eclipse-based IDE
- Refine UI for ease of use
- LdSaG Editor
- EVB board profiling
- Code Coverage
- Plug-ins internationalization
- Performance Analyzer
- Function code size analysis
- Hotkeys for collecting logs
- Target Management
- Chip Profile Editor
- In-System Programming
- External Plugin APIs
- Advanced Debugging
- RTOS Awareness Debugging
- COPILOT support

Co-design of Hardware and Software:

With the rapidly convergence of the embedded products, to pursuit an easy and flexible design environment and to save efforts of hardware designers and software programmers, AndeSight™ integrates all environments of hardware and software to provide a complete development solution.

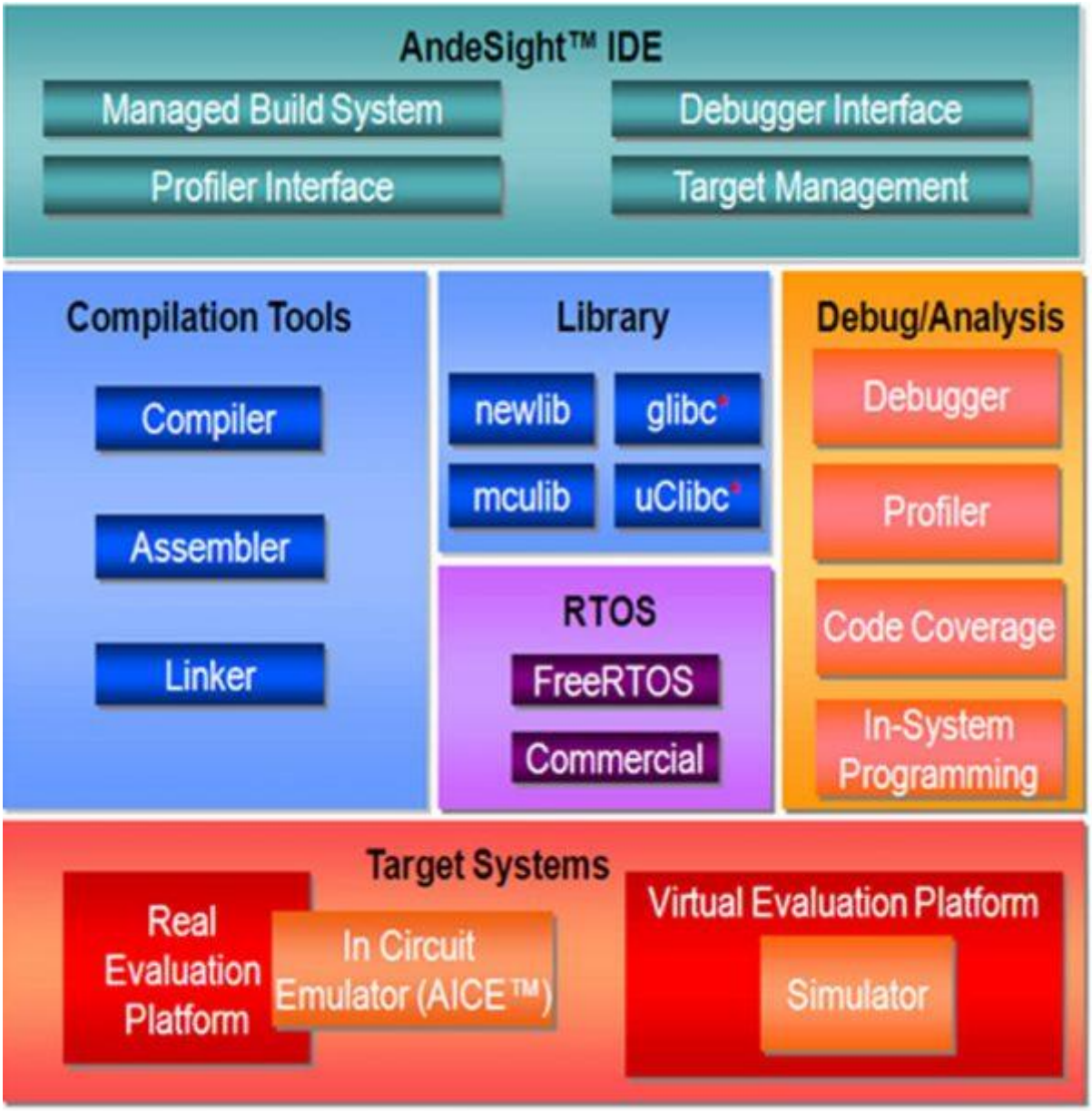


Fig. 2.1 Features included in AndeSight™ STD version

Coder Perspective:

A fully functional Andes Assembly, C and C++ integrated development environment with various perspective views help users on multiple project management. The coder perspective houses Project Explorer View, Code Editor, Target Manager and Console View. Users can create, delete, and change project configurations and manage build system in Project Explorer View.

On the Project Properties window, any project build related options, including toolchain selections, assembler, compiler, linker and debugger options and environment settings can be done through self-explanatory graphical user interfaces. The source code editor comes with syntax highlighting, code folding and auto-complete features which can help users edit the imported or newly created source code.

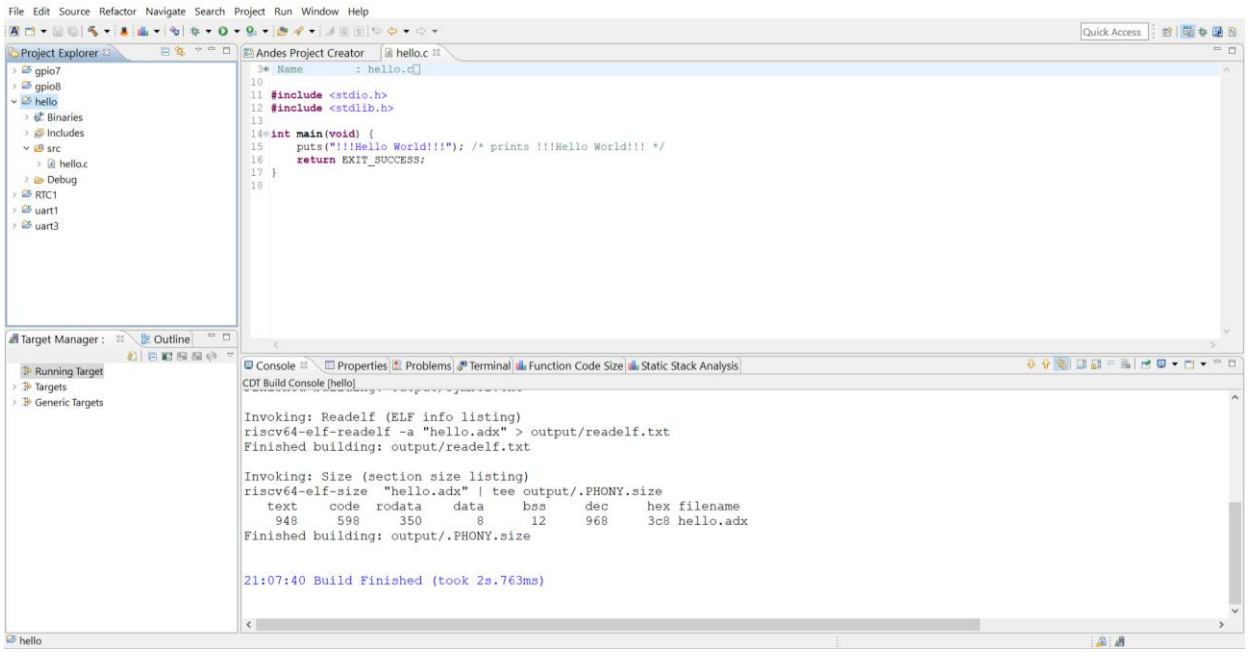


Fig. 2.2 Coder perspective

Configuration Setting:

More options are added in Project Configuration to provide fine-grained control on the executable image generation, such as start-up code tailor, program loading address, and customized linker script file. In addition to compiler, assembler and linker, user can also configure a few useful utility programs from Configuration Settings, such as objdump, readelf, nm objcopy and size.

Debug Perspective:

The Debug Perspective consists of the Code Editor, Target Manager and Debug Views that aid users in diagnosing each step in the program execution. The rich diagnosis features help users monitor system information during debugging process, such as execution stack outlining and active threads for each target on Debug Views and source-level debugging status in Code Editor.

The Memory View provides users memory content display and modification functions with user specified addresses. On the Disassembly View, users can see both high-level source code and its associated assembler code and identify issues in instruction level during debugging process even the program is written in high level programming language.

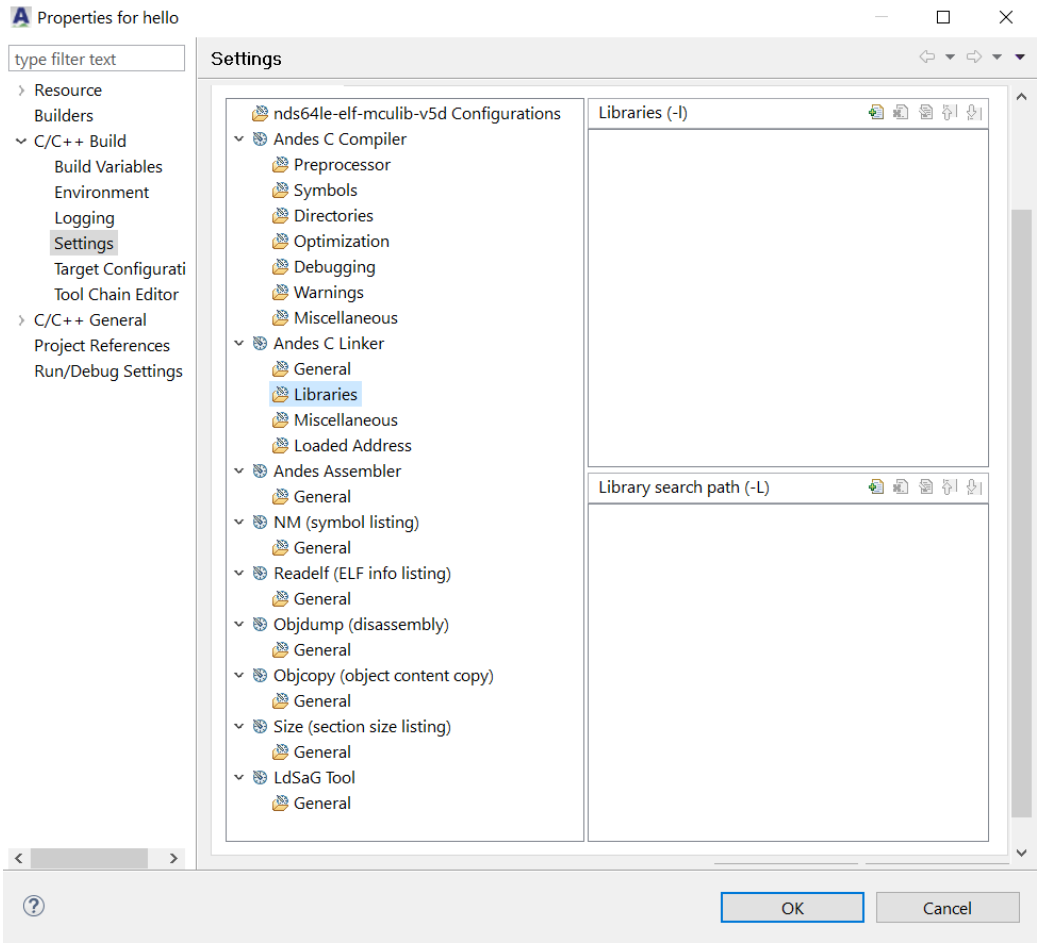


Fig. 2.3 Configuration Settings

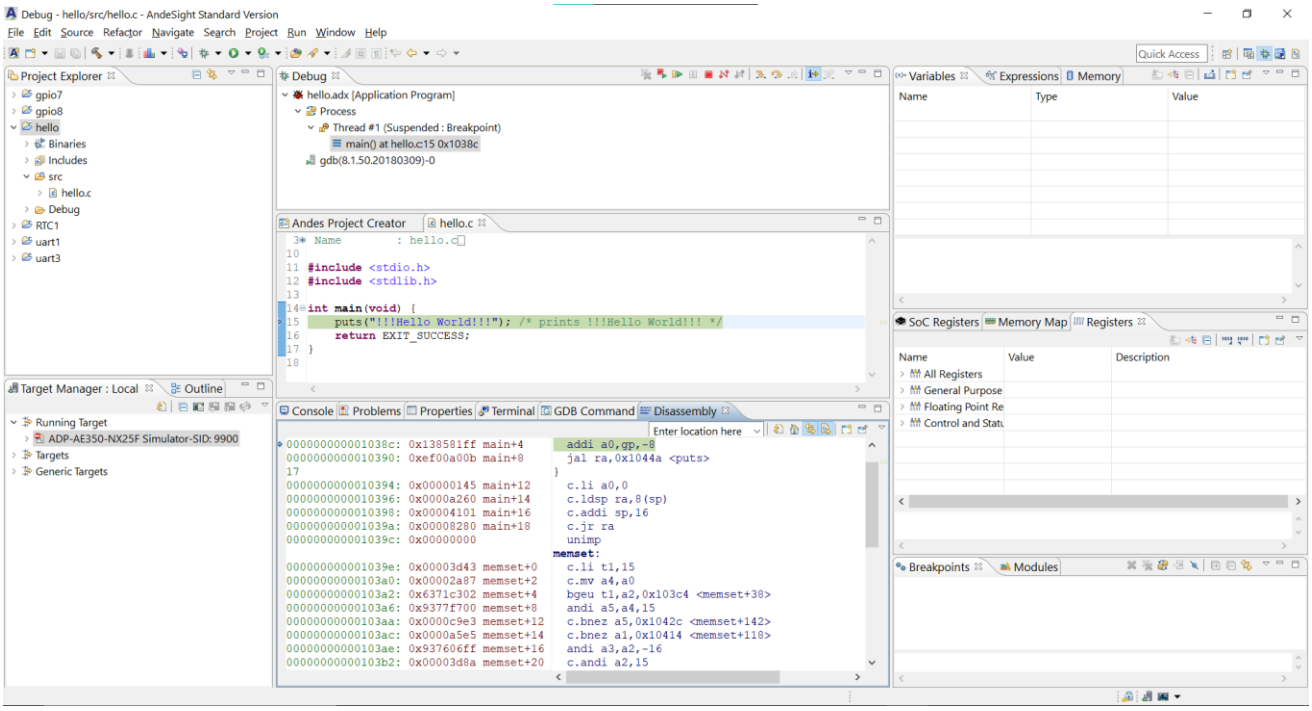


Fig. 2.4 Debug Perspective

Chapter 3
Installation Guide

3.1 Download Guide:

Andestech avails the users to use their evaluation product of AndeSight for a period of 3 months. In order to download the evaluation software, go to their official website http://www.andestech.com/en/support-downloads/andesight-ide-download/ and fill the **request form** with the necessary details.

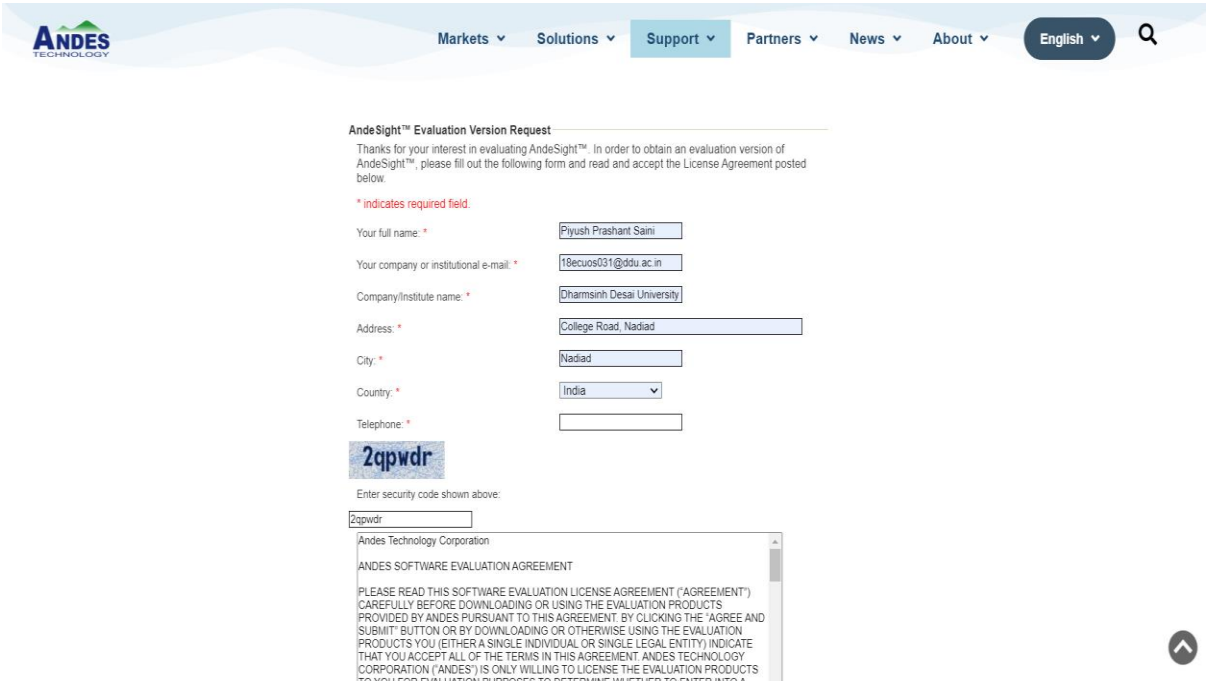


Fig. 3.1 AndeSight Evaluation Version Request Form

After filling the details, click on **Agree and Submit** to send the developers access request for evaluation product.

The software access will be sent via registered email ID (ensure to use the organization mail ID) as shown in Fig.3.2.

Download the software via the URL link provided in the mail. The steps for installation are provided on their website http://forum.andestech.com/viewtopic.php?f=6&t=1046.

Activation code for the evaluation period is attached in the mail and is needed to run the software.

AndeSight Evaluation Edition Request Notification Mail External Inbox ×

Andes Support Team <support@andestech.com>
to me

Dear Piyush Saini :
Your request for AndeSight™ evaluation version has been received and approved. This is the license file for Dharmsinh Desai University as attached.

Request ID : 6902
License Type: Active Code
Expiration Date: 2021/10/18

Please click the following link to download the AndeSight™ evaluation version. This link is active for 7 days from now on.
URL:http://quick.andestech.com/andesight/get_file/64d10d49f26d2503aed40f2dc8562008
Quick Start: "AndeSight STD v3.2.0 Quick Start"
We recommend that you use "Free Download Manager" tool to make the download smooth.
If there is any problem, please contact Andes at support@andestech.com

Best Regards,
Andes Technology Corporation

Fig. 3.2 Received Email for AndeSight

3.2 Setting up the Software:

After successfully installing, it is needed to set up the software before using the simulator.

1) **Step 1**: From the **connection configuration**, choose the simulator **SID**. Choose **C** for the project language. C++ can be chosen if the user wants to develop projects related to Operating System.
   From the chip profile, choose one CPU that the user wants to work with. Here, NX25F is chosen for the project. After choosing the options, click on **Create Project**.

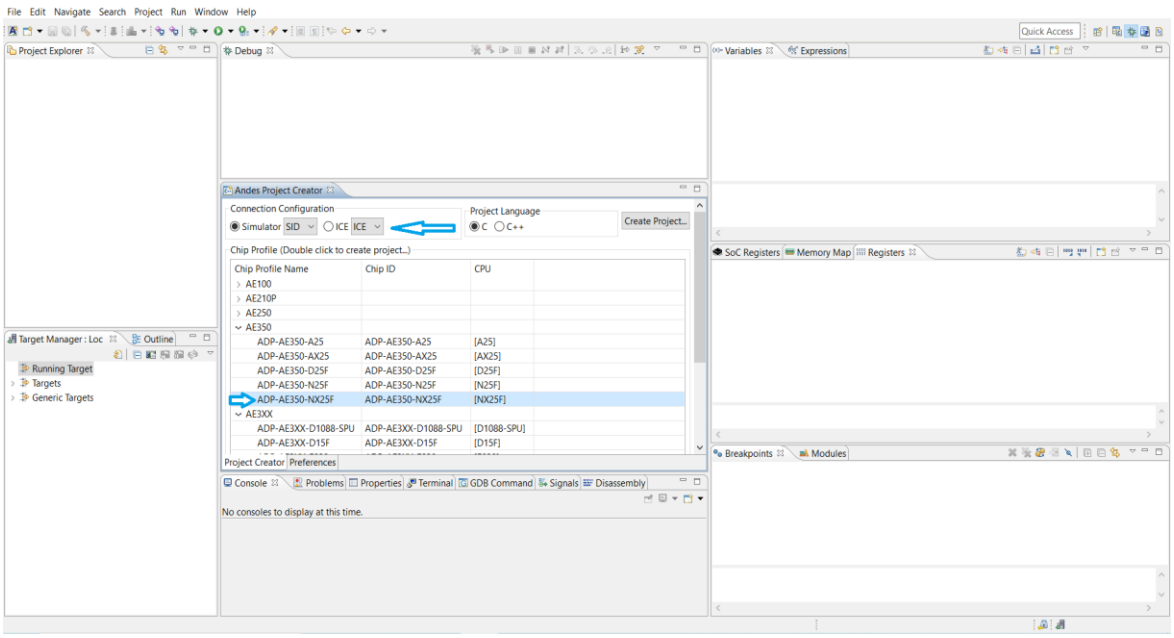


Fig. 3.3 Choosing the CPU for project.

2) **Step 2:** In the new window, give a project name and choose basic **Hello World ANSI C Project.** Choose one of the available RISC V toolchain. Toolchain allows emulation of RISC V without needing the actual hardware. Click on **Finish** after choosing the required options.

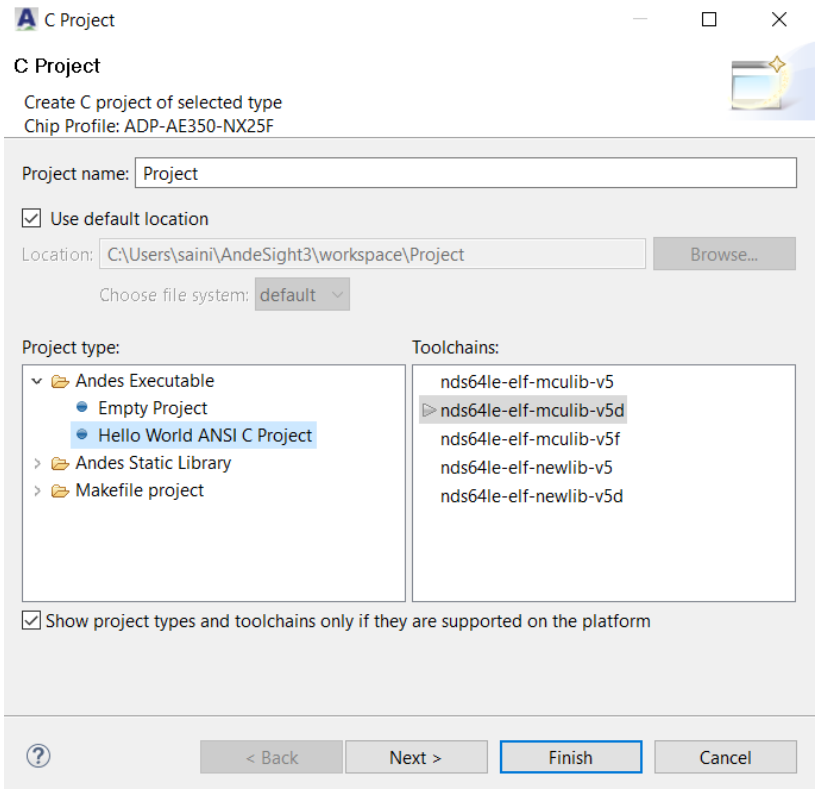


Fig. 3.4 Choosing the Toolchain

3) **Step 3:** After creating the project, it is necessary to set up the environment and include all the necessary header files into the project. Open the folder where AndeSight software was installed. Now, create a new folder. This folder will contain all the header files.

   Inside the folder where AndeSight was installed, go to the path **amsi/bsp/v5/ae350** and copy the following header files into the newly created folder:

   i) ae350.h
   ii) platform.h
   iii) timer.h

   The version of toolchain might differ from user to user. In case a CPU which supports older version of toolchain then copy the files from v3. Also, chip profile of AE350 family was chosen and hence the respective header file was copied into the folder. It might change as per the user.

Now, from the path **amsi/driver/v5/ae350**, copy the following header files into the folder:

i) dma_ae350.h
ii) gpio_ae350.h
iii) i2c_ae350.h
iv) spi_ae350.h
v) rtc_ae350.h
vi) pwm_ae350.h
vii) usart_ae350.h
viii) wdt_ae350.h

4) **Step 4:** Right click on the created Project folder in the **Project Explorer** tab. Then select **Build Settings** as shown.

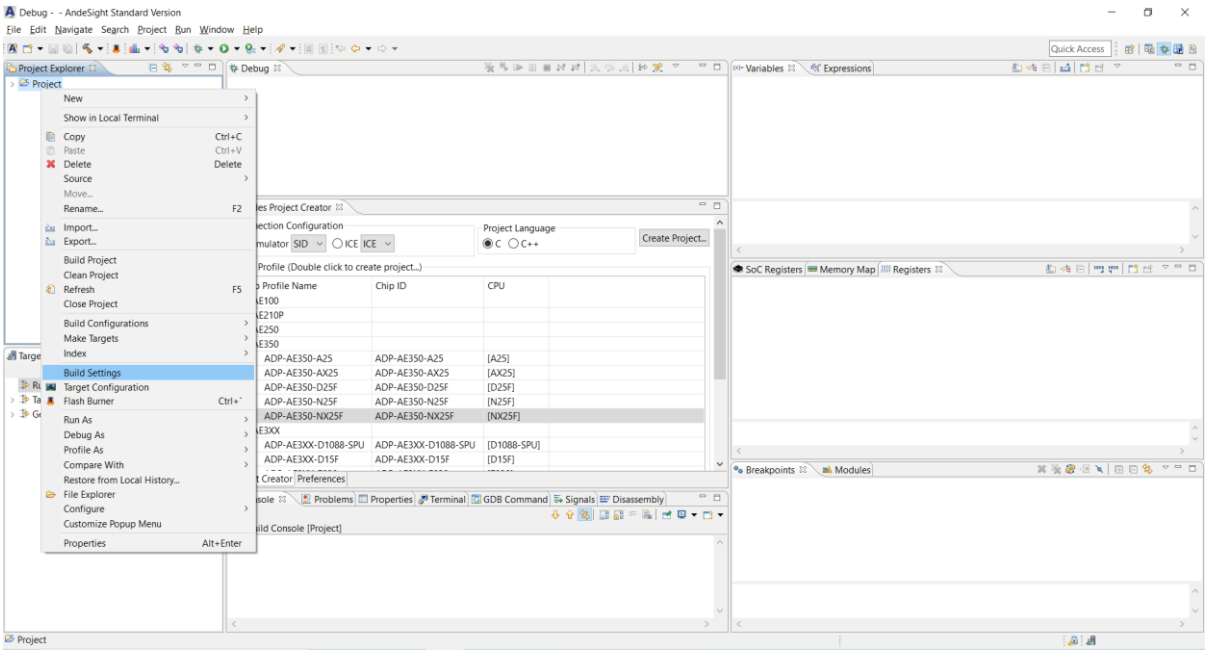


Fig. 3.5 Choose Build Settings

5) **Step 5:** Now choose **Settings** and then click on **Directories** under **Andes C Compiler**. Then click on **Add** icon as shown by step 3 in Fig. 3.6.1.

After clicking on the icon, a dialog box pops up as shown in fig. 3.6.2. Here, click on **File System** and browse the folder where all the header files were copied. By adding the files into this setting, the header files can be directly imported into the project. The compiler will directly use these header files while compiling.

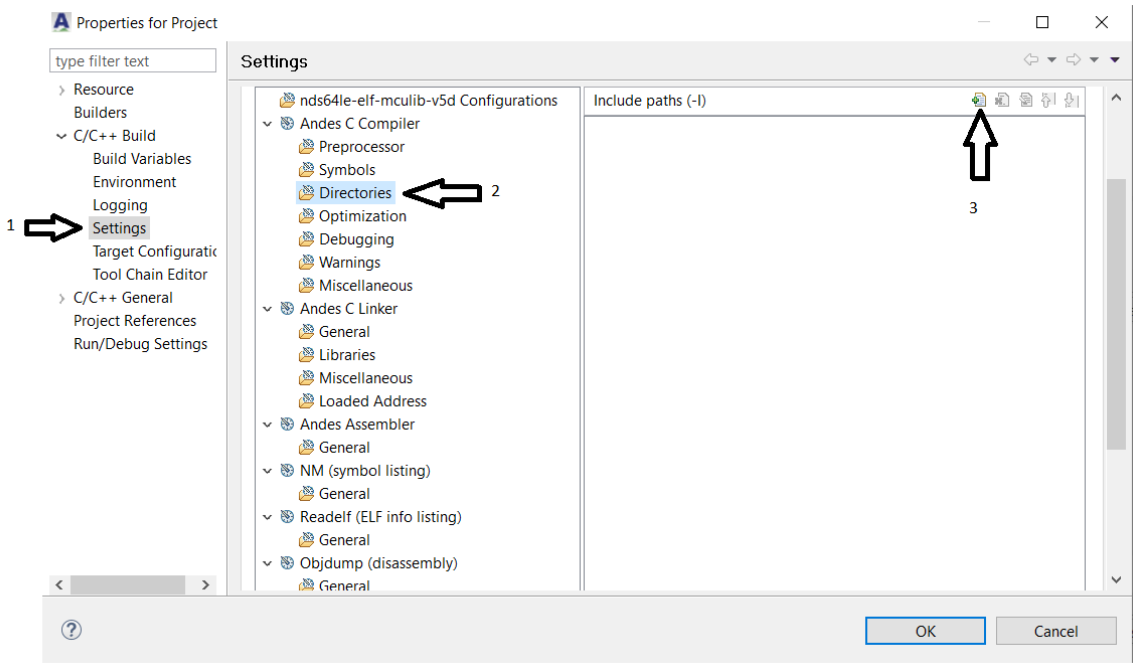


Fig. 3.6.1 Including additional files

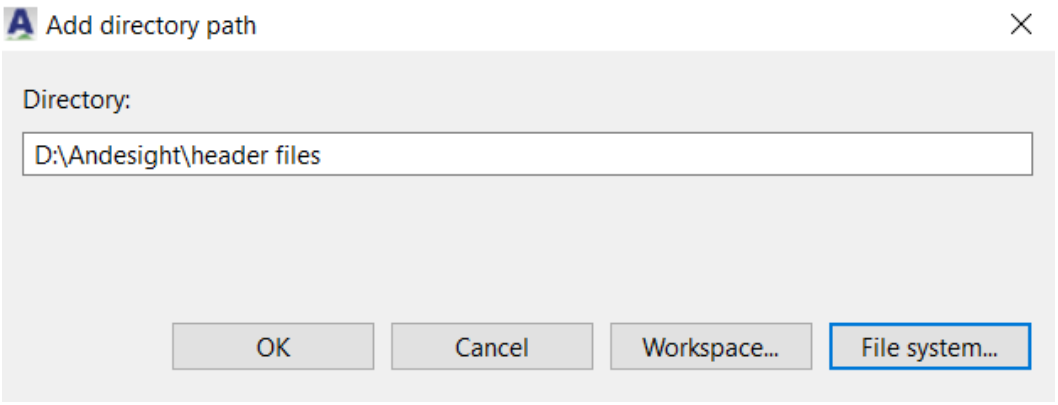


Fig. 3.6.2 Specify the Path to Header Files folder

6) **Step 6:** Now, click on **Libraries** option and then on **Add** icon as shown by step 2 in Fig.3.7. This setting specifies the location of header files where the linker needs to check while linking the project. Specify the path to the header files as done in Fig.3.6.2 and then click on OK to finish the setup of the project.

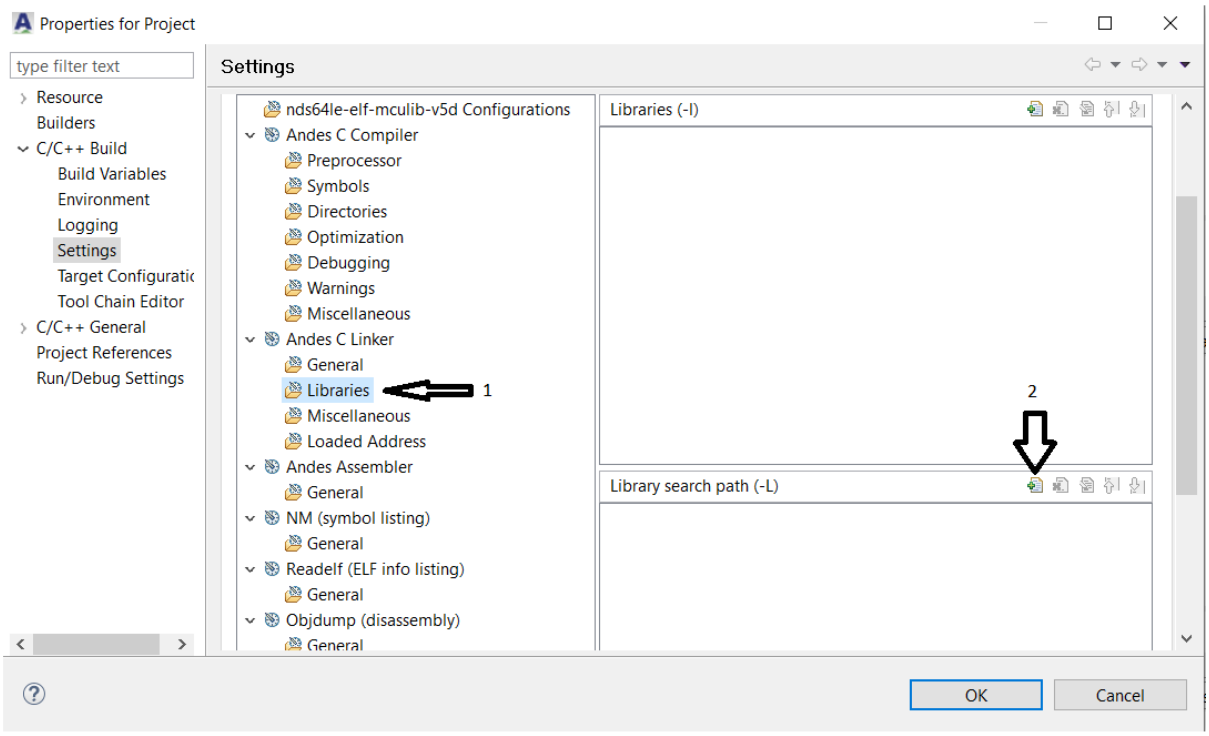


Fig. 3.7 Add the path to Header Files to the linker

The steps 5 and 6 needs to be performed every time a new project is created. These steps ensures that the user can access all the special registers available in a specific CPU.

3.3 Compiling and Execution:

The main program is available under **src**. This is shown in Fig.3.8 along with the basic "Hello World" program using C.

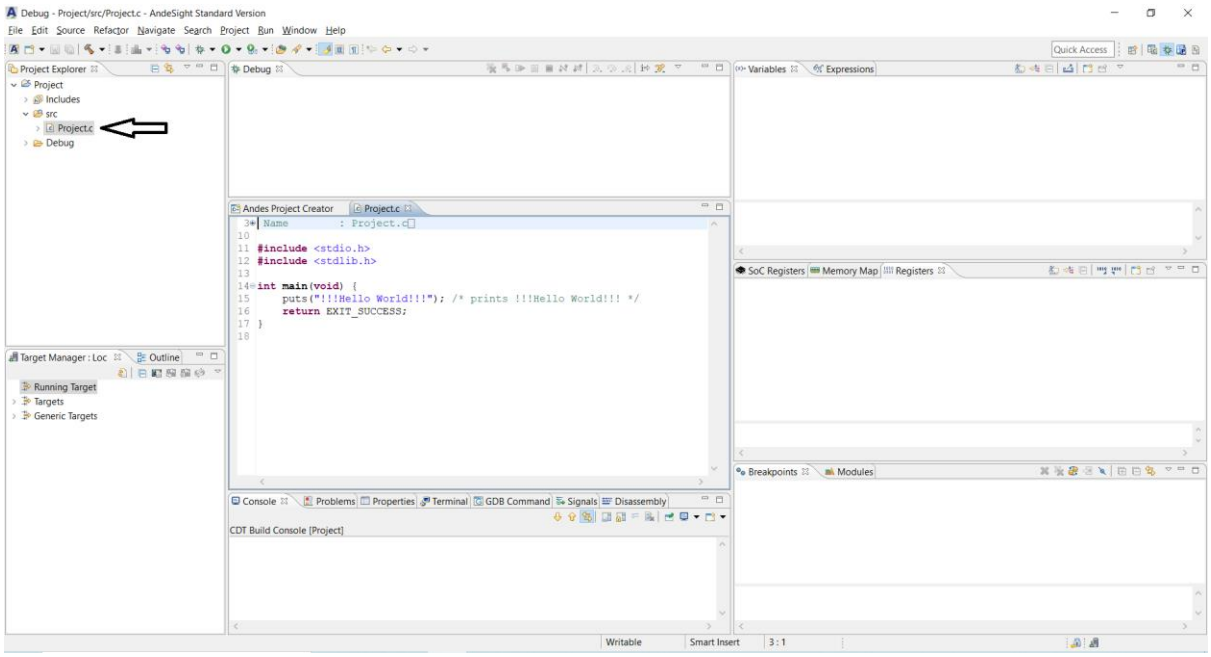


Fig. 3.8 Opening the main program

In order to execute any program written, follow the given steps:

1) **Step 1:** Click on the hammer icon **Build** in order to build the program on the chosen toolchain.

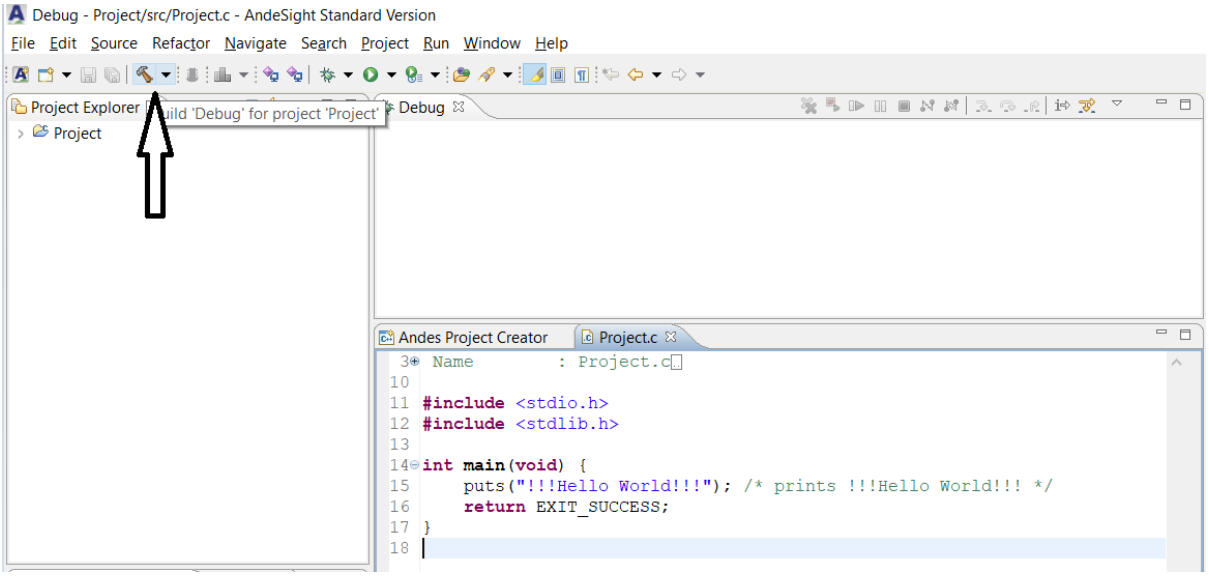


Fig.3.9 Building the project

2) **Step 2:**
   a) In order to run the program, click on the **Run** button.

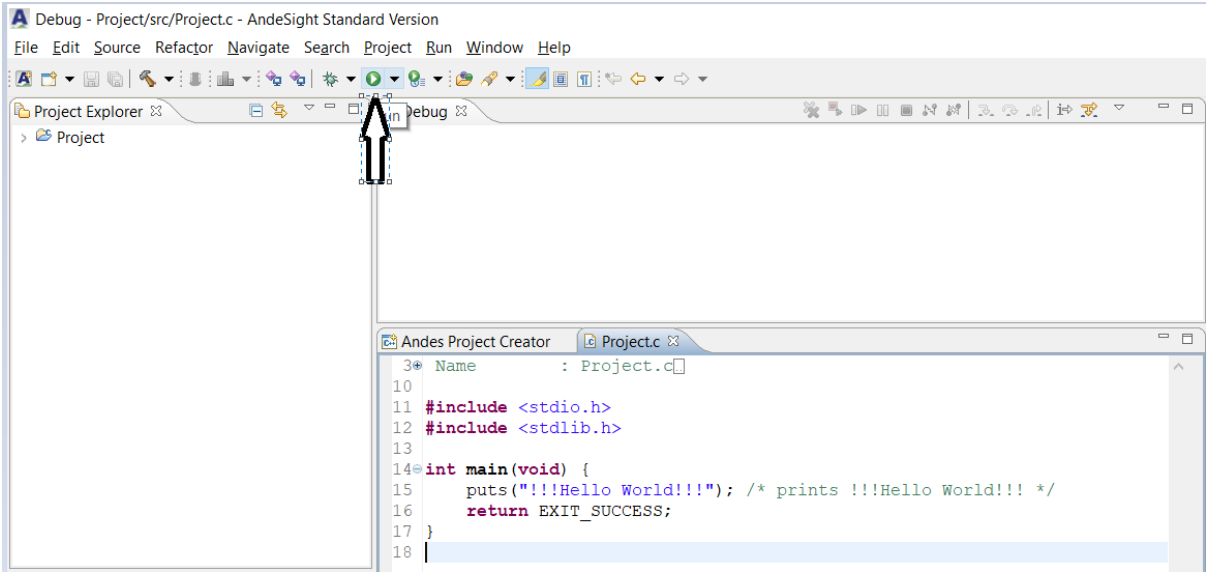


Fig.3.10.1 Running the program

The output of the program is shown in Fig.3.10.2.

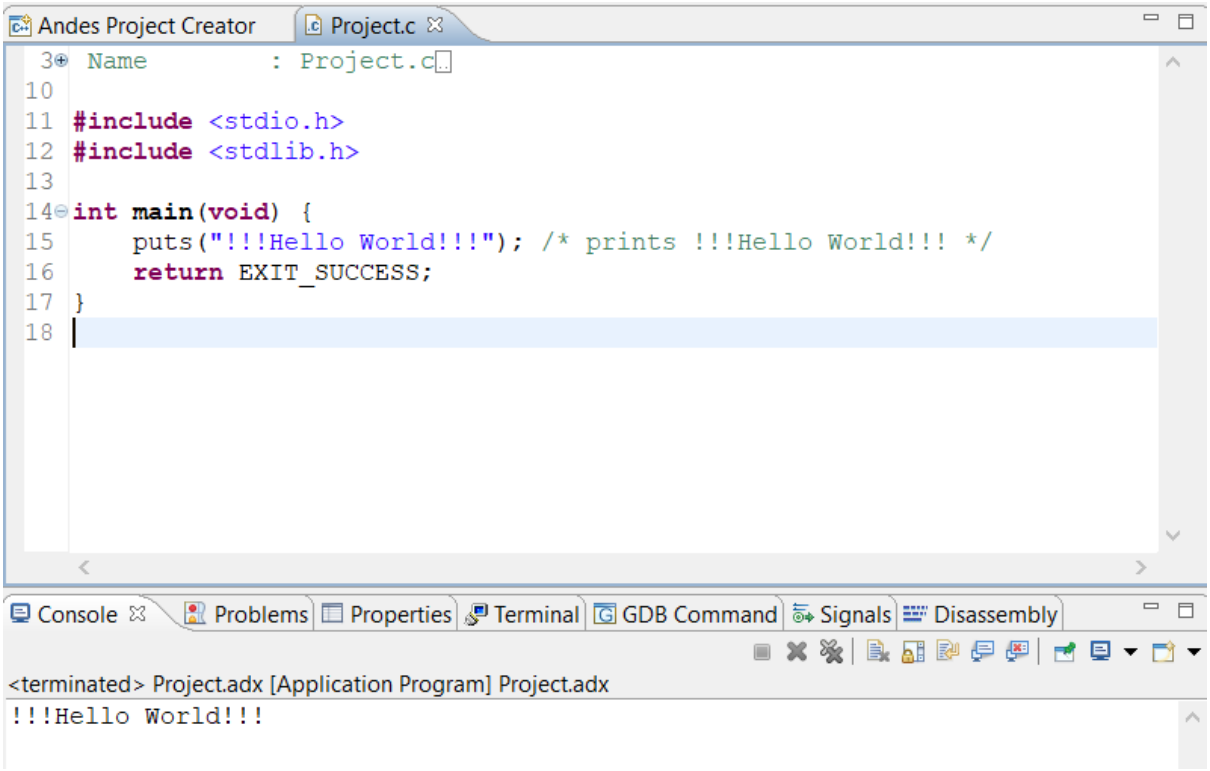


Fig.3.10.2 Output of the program

b) To debug the program, click on the bug like icon **Debug**. Debug option allows single stepping of the program and the user can understand the execution one line at a time. This is a useful option to remove logical errors that most programmers commit.

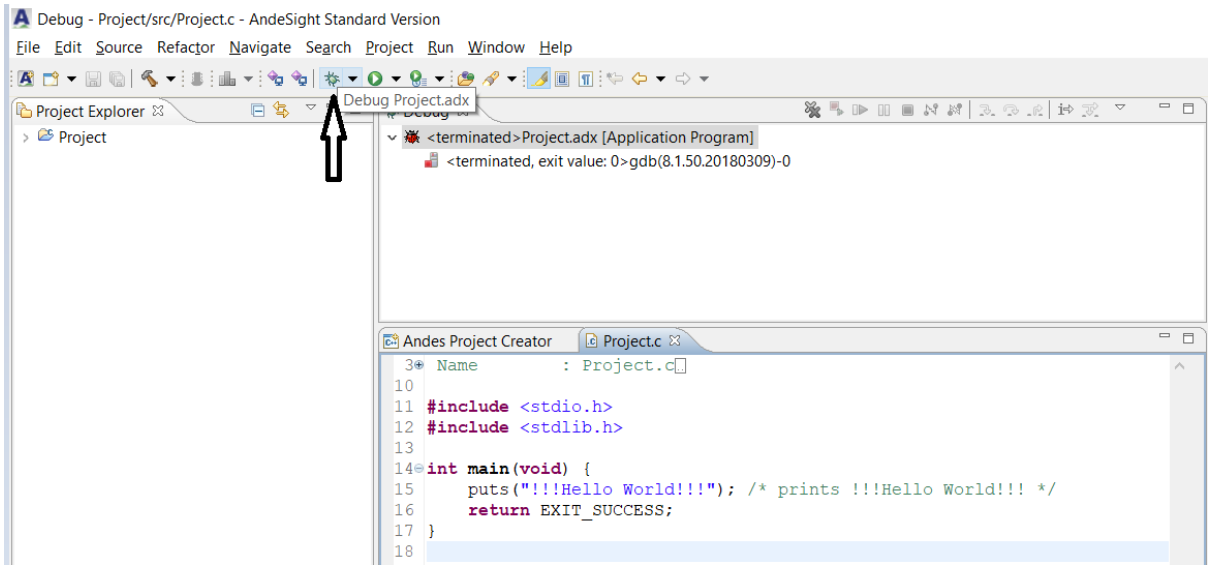


Fig. 3.11.1 Debugging the project

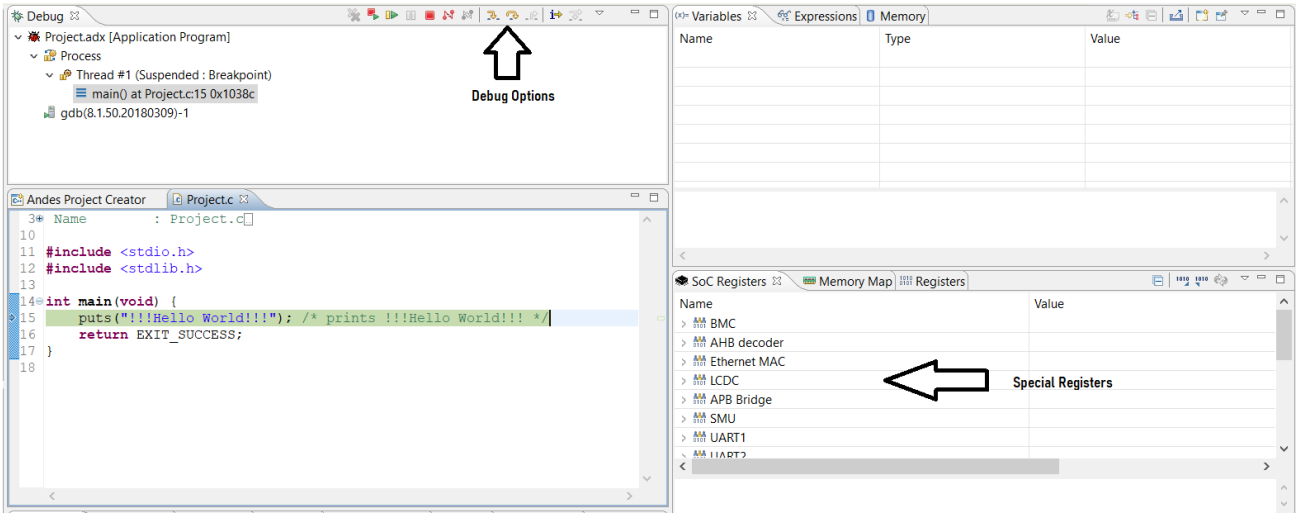


Fig. 3.11.2 Debug screen

Fig. 3.11.2 shows the debug screen. The various debug options like **Step Into** and **Step Over** are available at the top of the screen. The program can be made to free run using **Resume** option. The right side of the screen shows the **Special Registers**. The values of individual bits can be observed using these registers.

The current line to be executed is shown in green along with a small arrow on the left side of the line.

In order to come out of debugging mode, simply click on red square **Terminate** and the debugging stops.

Chapter 4
Microprocessor

4.1 NX25(F) Processor Features

For the project, NX25(F) was used. The main features processor are as follows:

CPU Core

• 5-stage in-order execution pipeline
• Hardware multiplier
  – radix2: 1-bit/cycle
  – radix4: 2-bit/cycle
  – radix16: 4-bit/cycle
  – radix256: 8-bit/cycle
  – fast: two-stage pipelined.
• Hardware divider
• Optional branch prediction
  – 4-entry return address stack (RAS)
  – Choice of
    * Static branch prediction, or
    * Dynamic branch prediction
      · 32/64/128/256-entry branch target buffer (BTB)
      · 256-entry branch history table
      · 8-bit global branch history
• Machine mode and optional User mode
• Optional performance monitors
• Misaligned memory accesses
• RISC-V physical memory protection

AndeStar V5 ISA

• RISC-V RV64I base integer instruction set
• RISC-V “C” standard extension for compressed instructions
• RISC-V “M” standard extension for integer multiplication and division
• Optional RISC-V “A” standard extension for atomic instructions
• Optional RISC-V “N” standard extension for user-level interrupt and exception handling
• Optional RISC-V “F” and “D” standard extensions for single/double-precision floating-point
• Andes Performance extension
• Andes CoDense extension

Memory Subsystem

• I & D-Caches
  – Cache size: 4KiB/8KiB/16KiB/32KiB/64KiB

  – Cache line size: 32 bytes
  – Set associativity: Direct-mapped/2-way/4-way
  – Custom cache control operation through CSR read/write
• I & D local memories
  – Size: 4KiB to 16MiB
  – Optional local memory (LM) slave port
  – Interface: RAM or AHB-Lite
• Memory subsystem soft-error protection
  – Protection scheme: parity-checking or error-checking-and-correction
  – Automatic hardware error correction
  – Protected memories:
    * I-Cache tag RAM and data RAM
    * D-Cache tag RAM and data RAM
    * I & D local memories

Bus

• Interface Protocol
  – Synchronous AHB, or
  – Synchronous AXI4
• 64-bit data width
• Configurable address width: 32–64 bits

Power Management

• Wait-for-interrupt (WFI) mode

Debug

• RISC-V External Debug Support
• Configurable number of breakpoints: 2/4/8
• External JTAG debug transport module
  – JTAG: IEEE Std 1149.1 style 4-wire JTAG interface
  – Serial: Andes 2-wire serial debug interface

Platform-Level Interrupt Controller (PLIC)

• Configurable number of interrupts: 1–1023
• Configurable number of interrupt priorities: 3/7/15/31/63/127/255
• Configurable number of targets: 1–16
• Andes Vectored Interrupt extension

4.2 Block Diagram

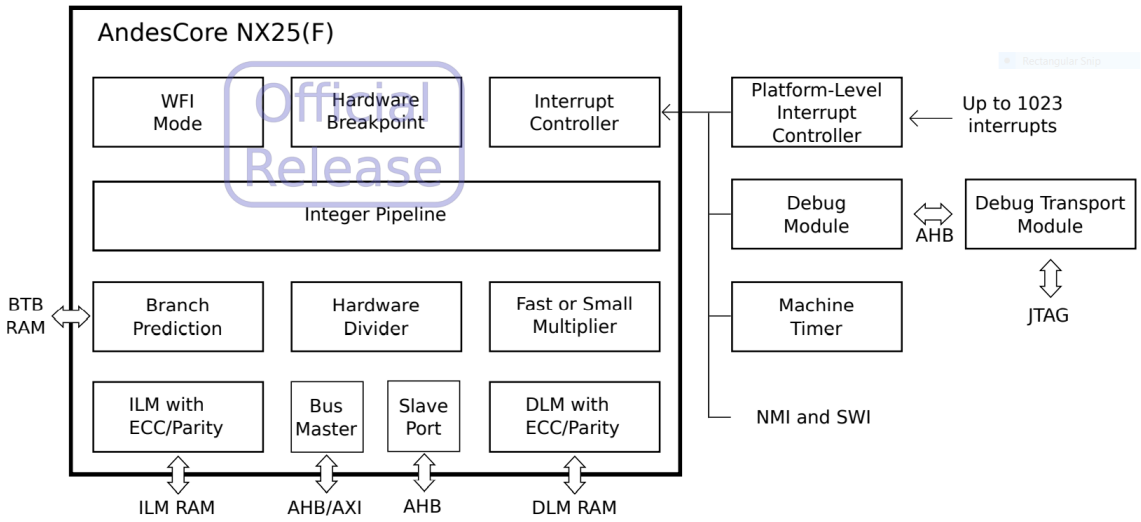


Fig 4.1 NX25(F) Block Diagram

4.2.1 Major Components

The following list describes the major components of the NX25(F) processor:

ACE Andes Custom Extension

ALU Arithmetic Logic Unit

BIU Bus Interface Unit

CSR Control and Status Register

DCU Data Cache Unit

DLM Data Local Memory Controller

FASTMUL Fast Multiplier

FPU Floating Point Unit

ICU Instruction Cache Unit

IFU Instruction Fetch Unit

ILM Instruction Local Memory Controller

IPIPE Integer Pipeline

LSU Load Store Unit

MDU Multiplication and Division Unit

PM Physical Memory Protection Unit

RF Register File

TRIGM Trigger Module

4.3 Pipeline Stages and Activities

The NX25(F) processor implements a five-stage pipeline architecture. The following figure shows the pipeline stages.

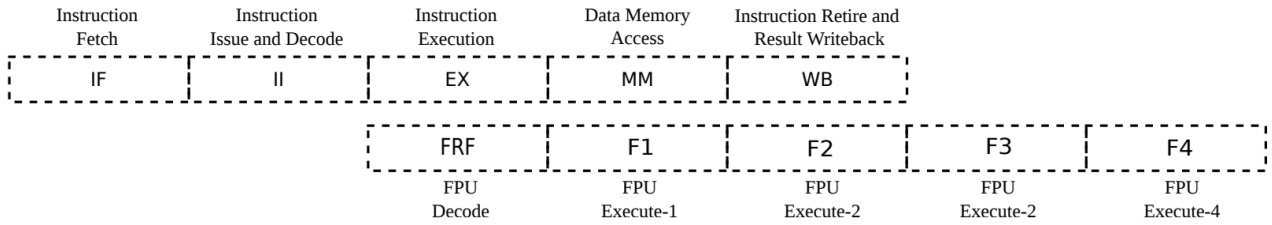


Fig. 4.2 Pipeline Stages

The pipeline activities of the corresponding stages are:

IF - Instruction Fetch

• Fetching an instruction word from ILM/I-Cache/Bus
• Dynamic branch prediction

II - Instruction Decode and Issue

• 16/32-bit instruction alignment
• Instruction decoding
• Register file read
• Resolving data dependency
• Static branch prediction

EX - Instruction Execution

• ALU instruction execution
• Load/Store address generation

MM—Memory Access
• DLM/D-Cache access
• Division instruction execution
• Multiplication instruction execution
• Branch resolution

WB - Instruction Retire and Result Write-Back

• Interrupt resolution
• Instruction retire
• Register file write back
• ILM access
• Bus access
• ACE instruction execution

FRF - FPU Instruction Decode

• Instruction decoding
• Register file read

F1~F4—FPU Instruction Execution

• Floating-point arithmetic execution
• Data exchange between Integer/FPU pipelines

4.4 Support Available

NX25(F) consists of the following peripherals and protocols:

1. DMA Support
2. GPIO Support
3. I2C Support
4. PIT Support
5. RTC Support
6. SPI Support
7. UART Support
8. WDT Support

Chapter 5
General Purpose Input/Output

5.1 Introduction

5.1.1 Description

Andeshape ATCGPIO100 is a general purpose I/O (GPIO) controller which supports upto 32 channels with independently programmable input/output control.

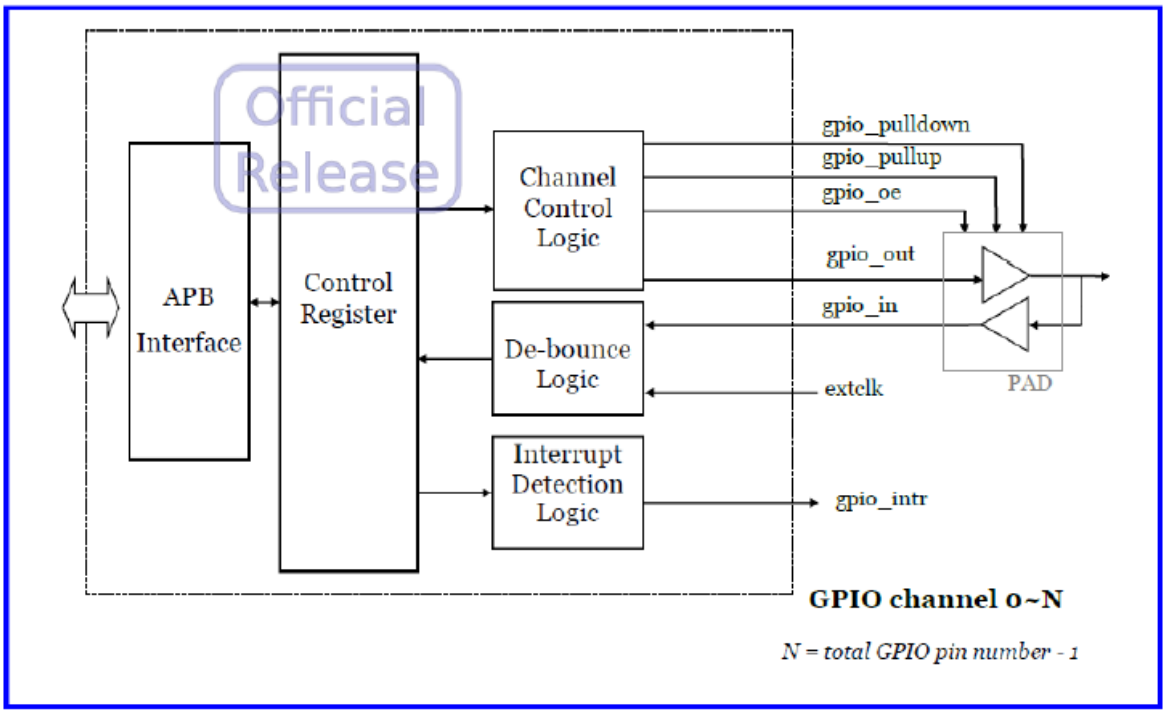


Fig. 5.1 Functional Block Diagram of ATCGPIO100

Each of the 32 channels can be individually programmed as either input or output. When configured as

i) Input – The controller samples the input signal and could generate an interrupt.
ii) Output – Could drive the output signal with pull-up/pull-down.

ATCGPIO100 also provides de-bounce function which filters out the glitches for input channels. The de-bounce duration is programmable as well.

5.1.2 Features

- Compliant with AMBA™ 2 APB protocol specification.
- Up to 32 independent I/O channels.
- Support of programmable pull-up/pull-down for each channel.
- Support of programmable de-bounce function for each input channel.
- Each input channel can be programmed as an interrupt input source.
- The interrupt source could be level or edge triggered.

5.2 Special Register Description

The abbreviations used for the **Type** column are as:

RO : Read Only
R/W : readable and writable
W1C : write 1 to clear
WO : write only (read as zero)

5.2.1 Configuration Register

This register defines the options available for the specific GPIO controller as defined by the manufacturer.

Table 5-1 Configuration Register

Name	Bit	Type	Description	Reset
Pull	31	RO	Pull option 0x0: Pull option is not configured 0x 1: Pull option is configured	Configuration dependent
Intr	30	RO	Interrupt option 0x 0: interrupt option is not configured 0x 1: interrupt option is configured	Configuration dependent
Debounce	29	RO	De-bounce option 0x 0: de-bounce option is not configured 0x 1: de-bounce option is configured	Configuration dependent
-	28:6	-	Reserved	-
ChannelNum	5:0	RO	Number of channels	Configuration dependent

5.2.2 Channel Direction Register

Defines the direction of GPIO channel independently.

Table 5-2 Channel Direction Register

Name	Bit	Type	Description	Reset
ChannelDir	N:0	R/W	GPIO channel direction 0x0: input 0x1: output	0x0

5.2.3 Channel Data-In Register

Contains the data that was given to the GPIO input. The input data can be read by reading this register.

Table 5-3 Channel Data-In Register

Name	Bit	Type	Description	Reset
DataIn	N:0	RO	Channel data input register	0x0

5.2.4 Channel Data-Out Register

The data that needs to be outputted from GPIO is fed into this register.

Table 5-4 Channel Data-Out Register

Name	Bit	Type	Description	Reset
DataOut	N:0	R/W	GPIO data output	0x0

5.2.5 Channel Data-Out Clear Register

Table 5-5 Channel Data-Out Clear Register

Name	Bit	Type	Description	Reset
DoutClear	N:0	WO	GPIO data-out clear; write 1 to clear the corresponding output channels.	0x0

5.2.6 Channel Data-Out Set Register

Table 5-6 Channel Data-Out Set Register

Name	Bit	Type	Description	Reset
DoutSet	N:0	WO	GPIO data-out set; write 1 to set the corresponding output channels.	0x0

5.2.7 Pull Enable Register

Enables the option of choosing between pull-up/pull-down on the corresponding GPIO channel.

Table 5-7 Pull Enable Register

Name	Bit	Type	Description	Reset
PullEn	N:0	R/W	GPIO pull enable; write 1 to enable pull-up/pull-down of the corresponding channels.	0x0

5.2.8 Pull Type Register

Defines the type of pull control on the enabled channels.

Table 5-8 Pull Type Register

Name	Bit	Type	Description	Reset
PullType	N:0	R/W	GPIO pull control 0x0: pull-up 0x1: pull-down	0x0

5.2.9 De-bounce Enable Register

Enables the de-bounce feature for the input data in the respective GPIO channel.

Table 5-9 De-bounce Enable Register

Name	Bit	Type	Description	Reset
DeBounceEn	N:0	R/W	Data-in de-bounce enable; write 1 to enable de-bouncing of the corresponding channels.	0x0

5.2.10 De-bounce Control Register

This register controls the de-bounce period of the input.

Table 5-10 De-bounce Control Register

Name	Bit	Type	Description	Reset
DBClkSel	31	R/W	GPIO de-bounce clock source selection. Select pclk (the faster clock) as the de-bounce clock source would shorten the de-bounce latency. 0x0: extclk 0x1: pclk	0x0
-	30:8	-	Reserved	-
DBPreScale	7:0	R/W	GPIO pre-scale base, to scale the de-bounce clock source before it is used as the actual de-bounce clock. The de-bounce period would be multiplied by (DBPreScale+1); e.g., setting DBPreScale to 3 would filter out pulses which are less than 4 de-bounce clock period.	0x0

5.3 Programming

5.3.1 Output channel Programming

The following program shows how to write a basic Output channel programming. The program turns ON the even LEDs at bit[0], bit[2], bit[4] and bit[6].

```
#include <stdio.h>
#include <stdlib.h>
#include <ae350.h>

/*Simple LED blinking program to demonstrate GPIO pin function using C
code*/

int main(void)
{
    int a,i;
    a=AE350_GPIO->CFG; //configuration register

    //print the value for debug purposes and check the availability of the
    purposes which are:
    //pull-up/down, interrupt enable, de-bounce, no. of channels
    printf("Output of configuration register = 0x%x\n",a);

    AE350_GPIO->CHANNELDIR=0x00000001;//configure GPIO pin 0 as output pin

    //infinite loop will result into continuous blink of an LED
    while(1)
    {
        AE350_GPIO->DATAOUT=0x55;
        //GPIO even numbered pins are ON
        for(i=0;i<200000;i++); //delay
        AE350_GPIO->DATAOUT=0x0;
        //GPIO even numbered pins are turned OFF
        for(i=0;i<200000;i++); //delay
    }
}
```

Debugging:

Build the program and then click on the debug option to start the debugging as shown in Fig. 5.3. Then, go to **Running Target** to select the GUI for GPIO (shown as 1 in image).

The GPIO GUI (as shown in Fig. 5.2) lights the corresponding LED for when the output bit is 1.

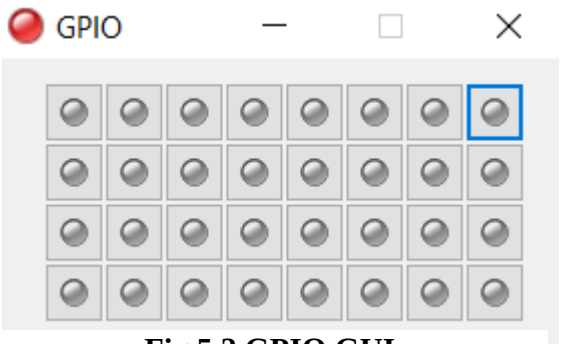


Fig.5.2 GPIO GUI

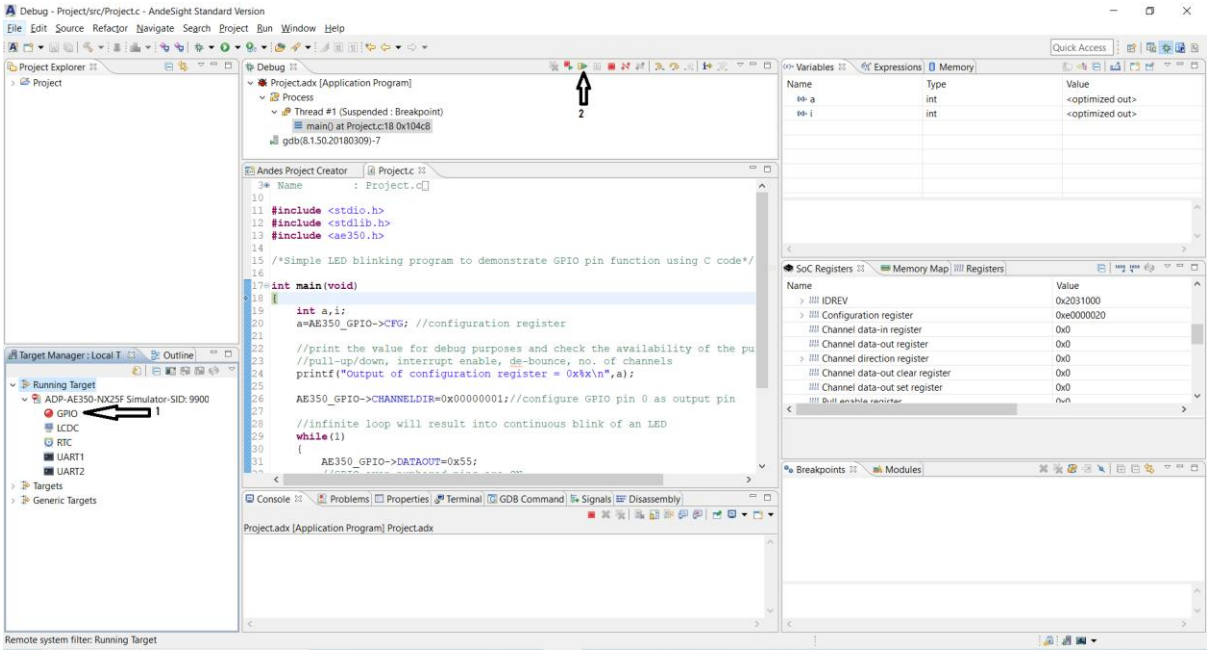
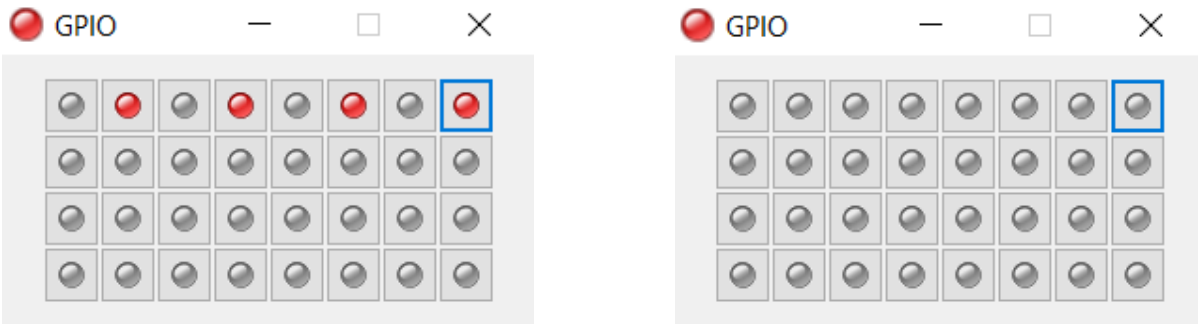


Fig. 5.3 Debugging the Output Channel Program

To run the program, simply click on **resume** (as shown as 2 in figure). This free runs the program just like it would run on the actual hardware.

Output:



a) **b)**

Fig. 5.4 Output of the program a) Even positioned LEDs are ON and b) After a delay the same are turned OFF

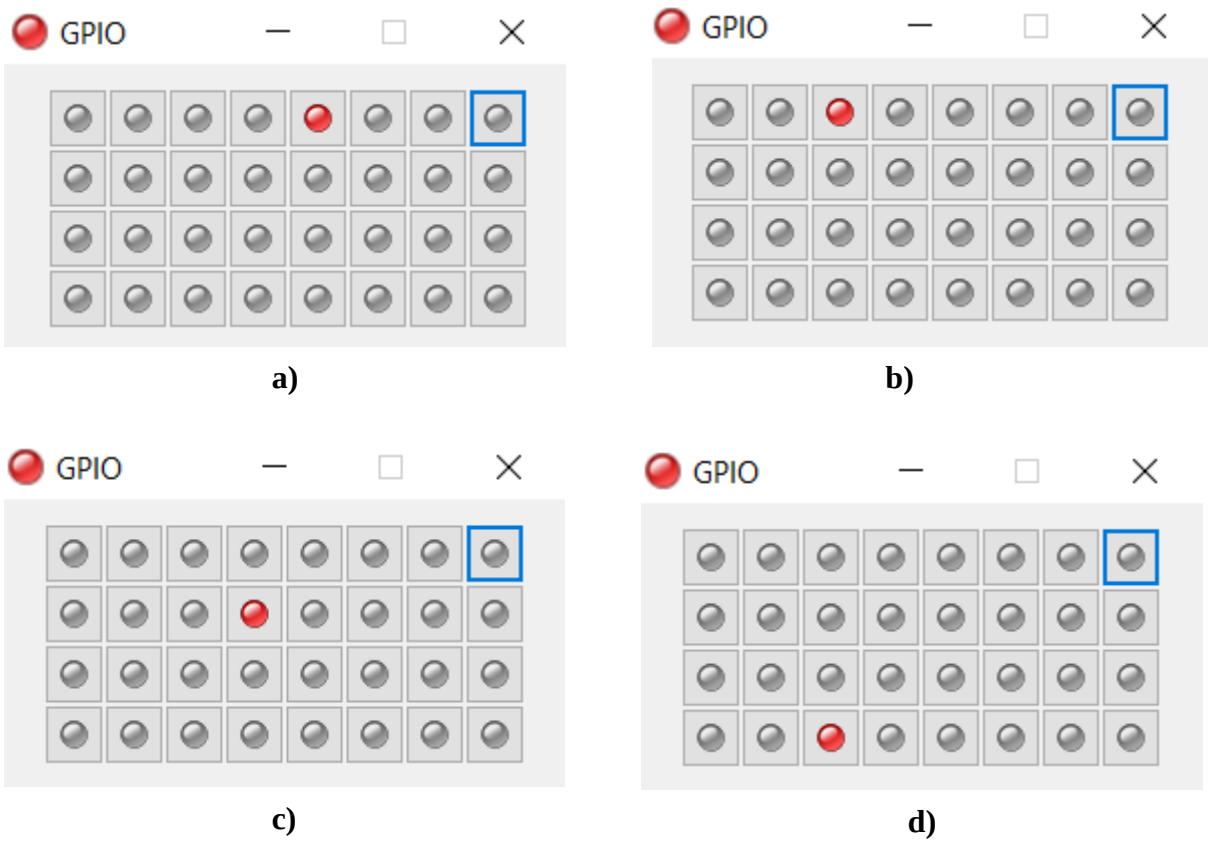
5.3.2 Sample Programs

a) Blinking LED train pattern:

```
#include <stdio.h>
#include <stdlib.h>
#include <ae350.h>

/*C program to get Blinking LED train pattern*/
/*This program can be used to test all the output ports at once*/

int main(void)
```

```
{
    int i,j;
      AE350_GPIO->CHANNELDIR=0xFFFFFFFF;
    //configure all the GPIO channels as output

    //infinite loop will result into continuous blink of an LED
    while(1)
    {
        for(j=0;j<=32;j++)
            {
                  AE350_GPIO->DATAOUT=(1<<j);
                  //creates the required pattern
                  for(i=0;i<200000;i++); //delay
            }
    }
}
```

Output:



a) **b)**

c) **d)**

Fig. 5.5 The LED train pattern starts from bit[0] and scrolls to bit[31] as shown from a) to d).

b) Blinking pattern using Data-Out Set and Reset Register:

```
#include <stdio.h>
#include <stdlib.h>
#include <ae350.h>

/*C program to get Blinking LED train pattern*/
/*This program can be used to test all the output ports at once*/
```

```
int main(void)
{
    int i,j;
      AE350_GPIO->CHANNELDIR=0xFFFFFFFF;//configure all the GPIO channels
as output

    //infinite loop will result into continuous blink of an LED
    while(1)
    {
        for(j=0;j<=31;j++)
            {
                  AE350_GPIO->DOUTSET|=(1<<j);  //creates the required
pattern
                  for(i=0;i<200000;i++); //delay
            }
        for(j=31;j>=0;j--)
        {
            AE350_GPIO->DOUTCLEAR|=(1<<j);      //creates the required
pattern
            for(i=0;i<200000;i++); //delay
        }
    }
}
```

Output:

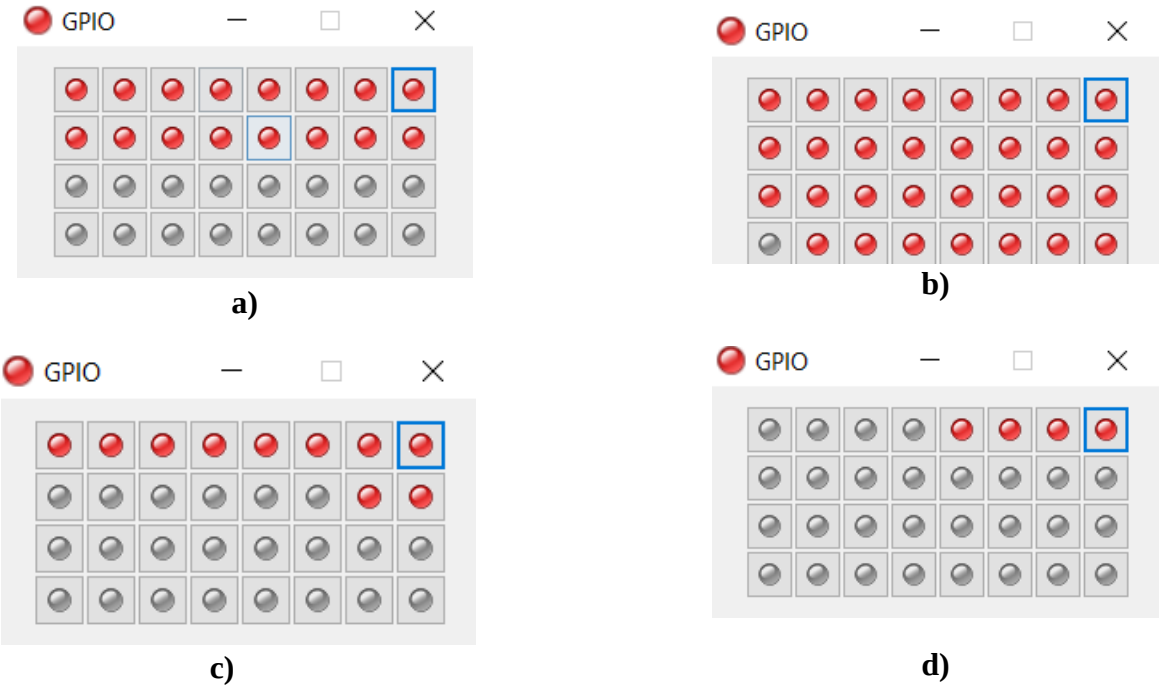


Fig. 5.5 The LED train pattern starts from bit[0] and turns ON all LEDs to bit[31] and then turns them OFF as shown from a) to d).

c) Custom Blinking pattern

```
#include <stdio.h>
#include <stdlib.h>
#include <ae350.h>

void pattern1()
{
```

```
	int i,j;
	AE350_GPIO->DATAOUT=0x01010101;	//creates the required pattern
	for(i=0;i<90000;i++); //delay
	for(j=1;j<8;j++)
	{
		AE350_GPIO->DATAOUT=(AE350_GPIO->DATAOUT)<<1;
		for(i=0;i<90000;i++); //delay
	}
}

void pattern2()
{
	int i,j;
    for(j=0;j<=31;j++)
	{
		AE350_GPIO->DOUTSET|=(1<<j);  //creates the required pattern
		for(i=0;i<10000;i++); //delay
	}

}

void pattern3()
{
	int i,j;
	for(j=1;j<4;j++)
	{
		AE350_GPIO->DOUTSET=0xFFFFFFFF;
		for(i=0;i<90000;i++); //delay
		AE350_GPIO->DOUTCLEAR=0xFFFFFFFF;
		for(i=0;i<90000;i++); //delay
	}
	AE350_GPIO->DOUTSET=0xFFFFFFFF;
}

void pattern4()
{
	int i,j;
	for(j=31;j>=0;j--)
	  {
		AE350_GPIO->DOUTCLEAR|=(1<<j);	//creates the required
pattern
		for(i=0;i<10000;i++); //delay
	  }

}

int main(void)
{
	AE350_GPIO->CHANNELDIR=0xFFFFFFFF;//configure all the GPIO channels
as output

    //infinite loop will result into continuous blink of an LED
    while(1)
    {
      pattern1();
      pattern2();
      pattern3();
      pattern4();
    }
}
```

5.4 Interrupt

Along with the basic GPIO programming of the ports, each port can be individually programmed for interrupts. Interrupts can be programmed to be triggered as either level triggered or edge.

In order to enable and configure interrupts, following registers are used.

5.4.1 Interrupt Enable Register

Enables the interrupt operation on the processor for use. Interrupt on individual ports can be enabled as per the need.

Table 5-11 Interrupt Enable Register

Name	Bit	Type	Description	Reset
IntEn	N:0	R/W	GPIO interrupt enable; write 1 to enable interrupts of the corresponding channels.	0x0

5.4.2 Channel Interrupt Status Register

Gives the interrupt status of corresponding channels.

Table 5-12 Channel Interrupt Status Register

Name	Bit	Type	Description	Reset
IntrStatus	N:0	R/W1C	The interrupt status of the corresponding channel; write 1 to clear. 0x0: No interrupt 0x1: Interrupt	0x0

5.4.3 Interrupt Mode Register

Defines the interrupt trigger mode of the corresponding ports. Interrupt mode register 0 defines the trigger mode for ports 0 to 7, Interrupt mode register 1 defines for ports 8 to 15 and so on. 4 such Interrupt mode registers are used. Table 5-13 shows the interrupt mode register 0 along with all the available trigger modes.

#Note: Before using interrupt for any GPIO port, it is necessary to configure the ports as input ports using channel direction register.

Table 5-13 Interrupt Mode Register0

Name	Bit	Type	Description	Reset
-	31	-	Reserved	-
Ch7IntrM	30:28	R/W	Channel 7 interrupt mode	0x0
-	27	-	Reserved	-
Ch6IntrM	26:24	R/W	Channel 6 interrupt mode	0x0
-	23	-	Reserved	-
Ch5IntrM	22:20	R/W	Channel 5 interrupt mode	0x0
-	19	-	Reserved	-
Ch4IntrM	18:16	R/W	Channel 4 interrupt mode	0x0
-	15	-	Reserved	-
Ch3IntrM	14:12	R/W	Channel 3 interrupt mode	0x0
-	11	-	Reserved	-
Ch2IntrM	10:8	R/W	Channel 2 interrupt mode	0x0
-	7	-	Reserved	-
Ch1IntrM	6:4	R/W	Channel 1 interrupt mode	0x0
-	3	-	Reserved	-
Ch0IntrM	2:0	R/W	Channel 0 interrupt mode 0x0: No operation 0x2: High-level 0x3: Low-level 0x5: Negative-edge 0x6: Positive-edge 0x7: Dual-edge 0x1, 0x4: Reserved	0x0

5.4.4 Interrupt Programming

Interrupts can be used in two ways in the program:

i) Polling the interrupt status register for the corresponding port or
ii) Checking the interrupt status register using an “if” statement.

Sample Program:

```
#include <stdio.h>
#include <stdlib.h>
#include <ae350.h>

void input_initilization()
{
	AE350_GPIO->CHANNELDIR=0x00;
	//configure all the GPIO channels as input
```

```
	AE350_GPIO->PULLEN=0x01;	//enable pull-type for channel 0
	AE350_GPIO->PULLTYPE=0x00;	//the channel 0 has pull-up type
	AE350_GPIO->DEBOUNCEEN=0x01;	//enable de-bounce for channel 0
	AE350_GPIO->DEBOUNCECTRL=0x80000003;
	//pclk as debounce clock source and use filter to remove glitches
}

void interrupt_initilization()
{
	AE350_GPIO->INTREN=0x01;	//enable channel0 interrupt
	AE350_GPIO->INTRMODE0=0x06;
	//channel0 interrupt mode as positive edge
}

void interrupt()
{
	printf("Interrupt occured\n");
	AE350_GPIO->INTRSTATUS=0x01;	//write 1 to clear the bit
}

int main(void)
{
	input_initilization();
	interrupt_initilization();
	while(!(AE350_GPIO->INTRSTATUS&0x01));
	//polling the interrupt status of channel0
	interrupt();
}
```

Debugging:

In order to simulate the interrupt on software, GPIO GUI can be used. Click on the port for which interrupt is to be generated. This action is basically providing an external input to the specific port.

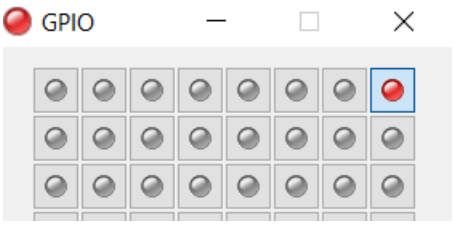


Fig.5.6 Providing Interrupt on channel0 using GPIO GUI

The output of the sample program is shown in fig. 5.7

The priority of the interrupts is defined by the way they are checked in the program. The first checked interrupt is serviced first and second is second and so on.

Limitation:

i) The software allows occurrence of only one interrupt at any given time. Two interrupts can never occur in the software.

ii) There is no global interrupt to know about the interrupt occurrence. The interrupts can be either polled or an "if" statement is to be used. So, there might be huge delay between occurrence and service of the interrupt.

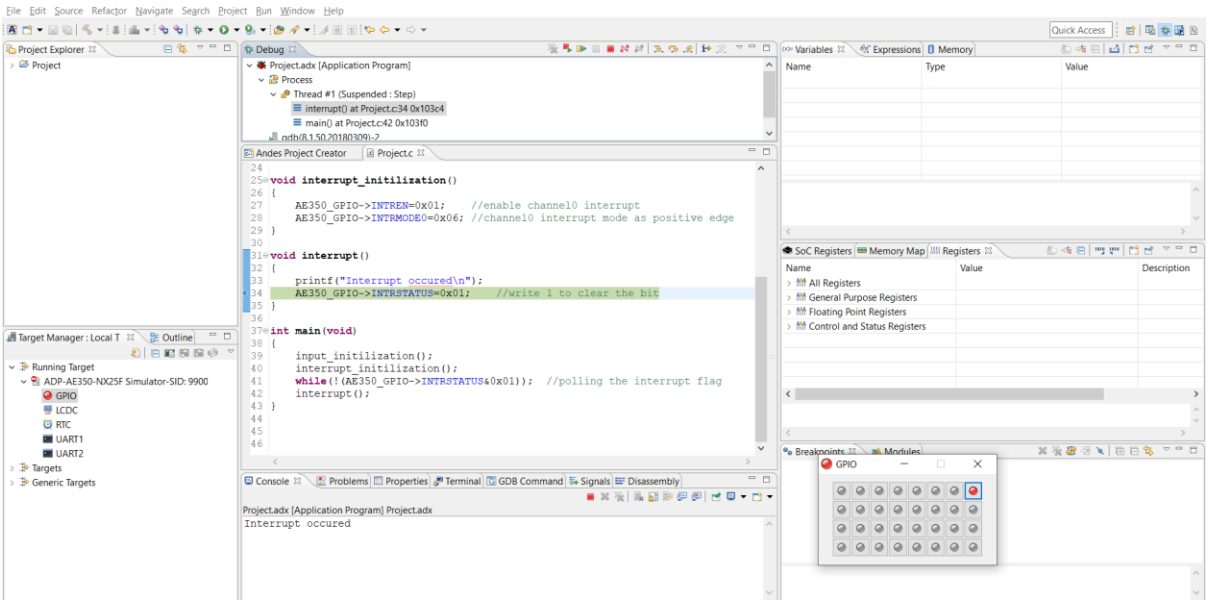


Fig.5.7 Interrupt at channel0

Interrupt Program:

```
#include <stdio.h>
#include <stdlib.h>
#include <ae350.h>

void input_initilization()
{
	AE350_GPIO->CHANNELDIR=0xFFFFFF00;
	//configure GPIO channels 0 to 7 as input and others as output
	AE350_GPIO->PULLEN=0xFF;
	//enable pull-type for channels 0 to 7
	AE350_GPIO->PULLTYPE=0x00;
	//the channels 0 to 7 has pull-up type
	AE350_GPIO->DEBOUNCEEN=0xFF;
	//enable de-bounce for channels 0 to 7
	AE350_GPIO->DEBOUNCECTRL=0x80000003;
	//pclk as de-bounce clock source and use filter to remove glitches
}

void interrupt_initilization()
{
	AE350_GPIO->INTREN=0x07;	//enable channels 0,1 and 2 interrupts
	AE350_GPIO->INTRMODE0=0x656;
	/* channel0 interrupt mode as positive edge
	 * channel1 interrupt mode as negative edge
	 * channel2 interrupt mode as positive edge
	 */
}

void interrupt()
{
	if(AE350_GPIO->INTRSTATUS == 0x02)  //if channel1 interrupt occurred
	{
		printf("Interrupt on channel 1 occurred\n");
		AE350_GPIO->INTRSTATUS=0x02;  //write 1 to clear the flag
	}
	if(AE350_GPIO->INTRSTATUS == 0x01)  //if channel0 interrupt occurred
```

```
	{
		printf("Interrupt on channel 0 occurred\n");
		AE350_GPIO->INTRSTATUS=0x01;  //write 1 to clear the flag
	}
	if(AE350_GPIO->INTRSTATUS == 0x04)  //if channel2 interrupt occurred
	{
		printf("Interrupt on channel 2 occurred\n");
		AE350_GPIO->INTRSTATUS=0x04;  //write 1 to clear the flag
	}
}

void pattern()
{	//if no interrupt occurred then run this basic LED pattern
	int i,j;
	for(j=8;j<32;j++)
	{
		AE350_GPIO->DOUTSET|=(1<<j);
		for(i=0;i<10000;i++);
	}
	for(j=31;j>7;j--)
		{
			AE350_GPIO->DOUTCLEAR|=(1<<j);
			for(i=0;i<10000;i++);
		}
}

int main(void)
{
	input_initilization();
	interrupt_initilization();
	while(1)
	{
		if(AE350_GPIO->INTRSTATUS == 0)
		//check whether any interrupt occurred
			pattern();
		else
			interrupt();
	}
}
```

Output:

Fig. 5.8.1 shows the normal working of the program. No interrupt has yet occurred in the program. Fig. 5.8.2 shows the scenario after the interrupt at channel 1 has occurred and been serviced. Fig. 5.8.3 shows the final scenario where interrupt at both channel 0 and 2 occurred one after the other.

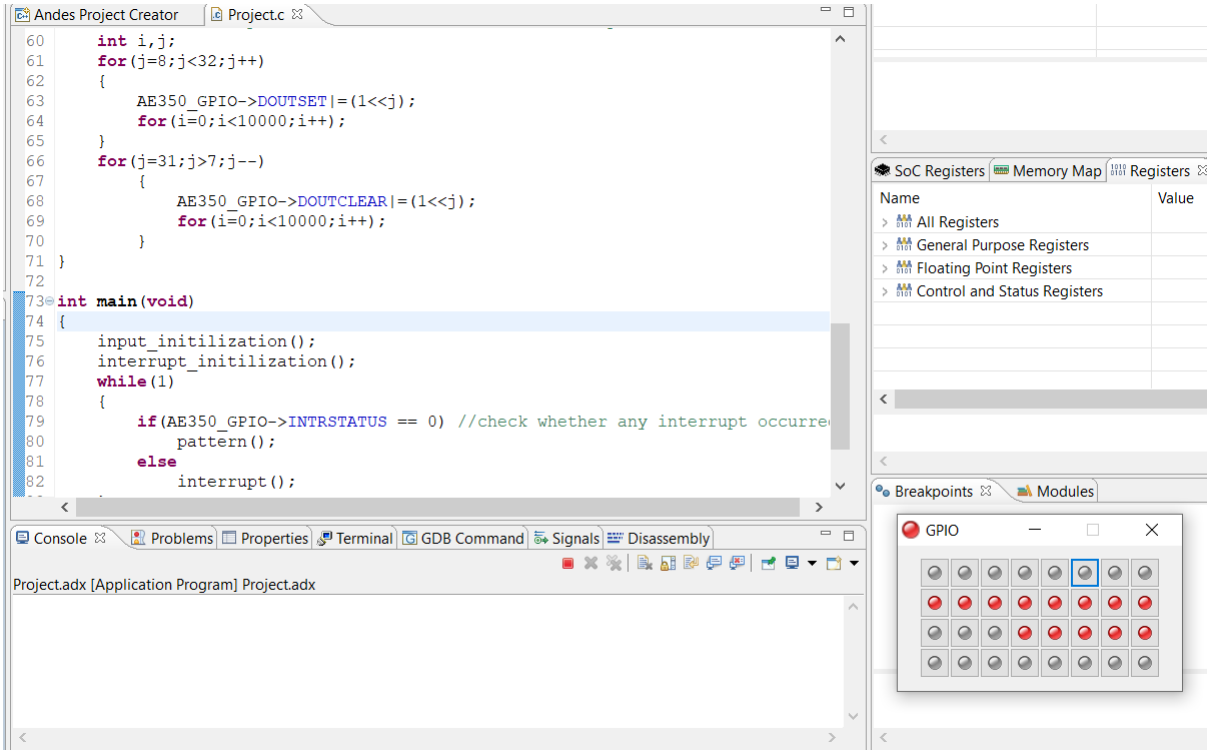


Fig.5.8.1 No interrupt occurred

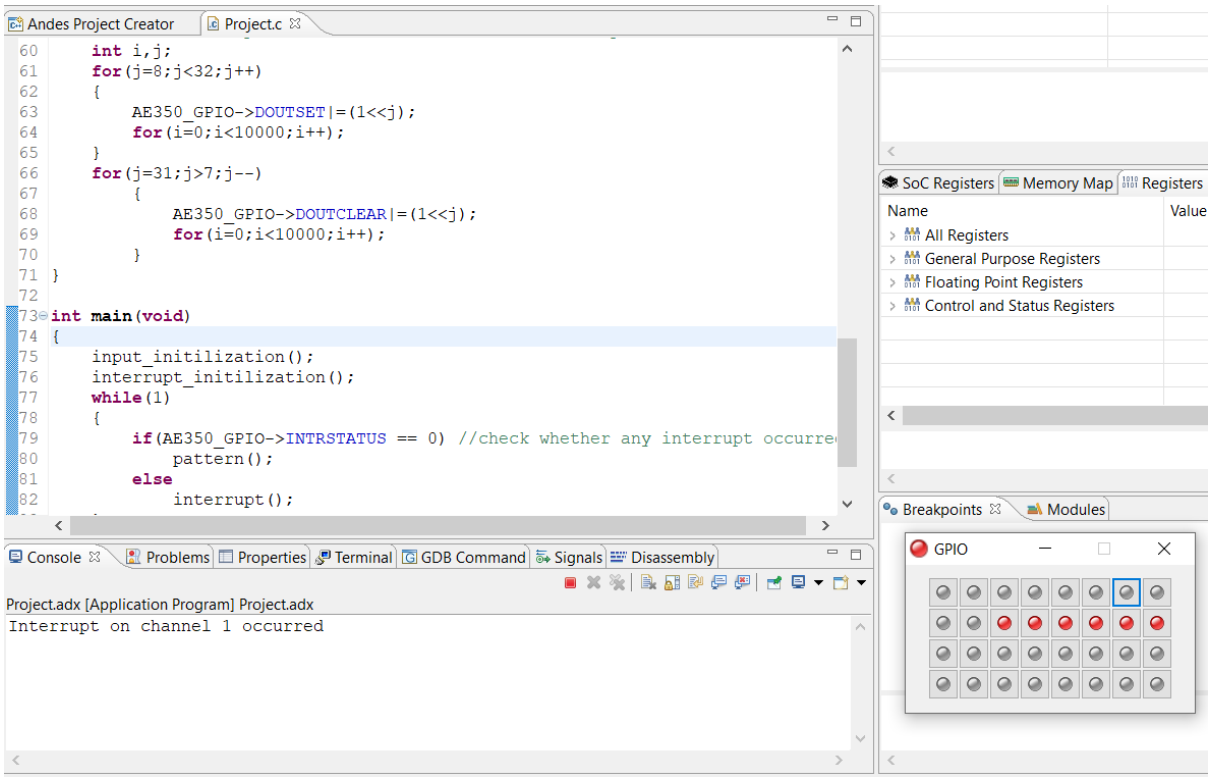


Fig.5.8.2 Interrupt at channel 1 occurred

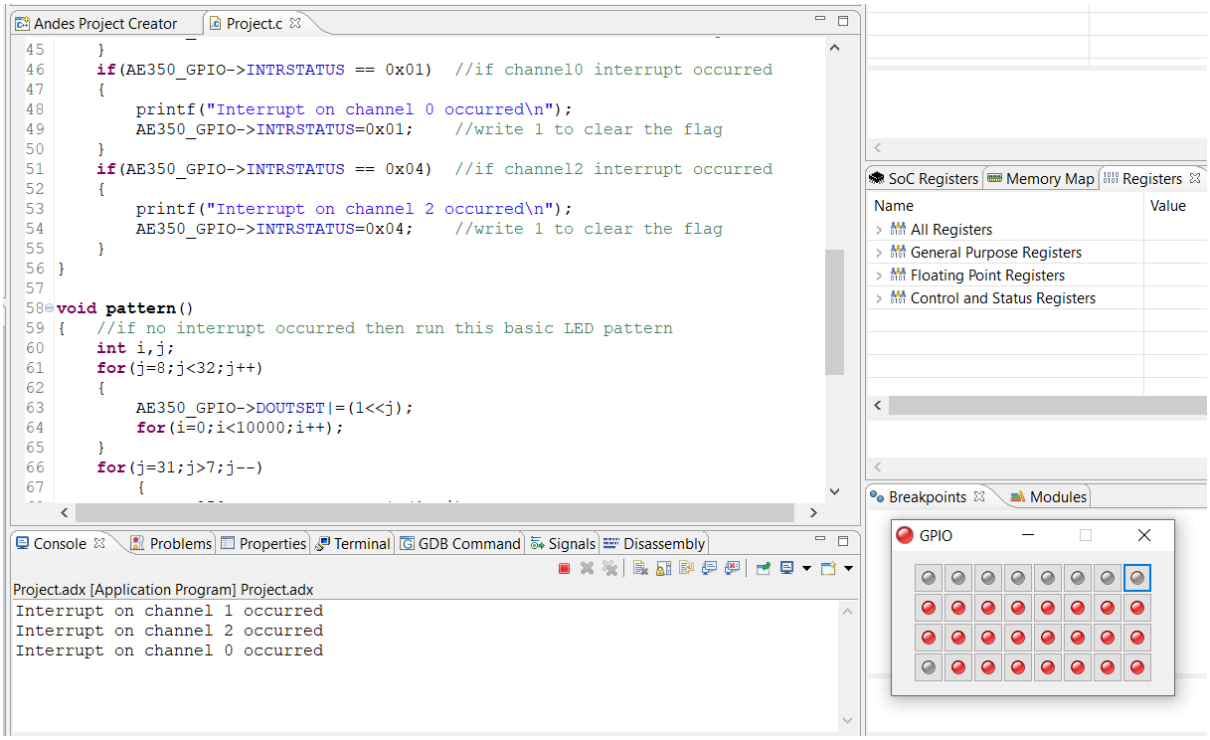


Fig.5.8.3 Interrupt at channel 2 and 0 occurred

Chapter 6
UART

6.1 Introduction:

6.1.1 Description:

AndeShape™ ATCUART100 is a Universal Asynchronous Receiver/Transmitter (UART)

The ATCUART100 controller is a serial communication controller which provides asynchronous serial interface for a peripheral device or a modem. The controller comprises a transmitter, a receiver, a Baud Rate Generator, a Modem Controller, a Register File and an APB interface.

6.1.2 Features:

- AMBA 2.0 APB interface for registers access
- Hardware configurable 16, 32, 64 and 128 bytes transmit/receive FIFOs
- Over-sampling frequency is programmable (even multiples ranging from 8x to 32x)
- Programming sequence compatible with the 16C550D UART
  - Supports 5 to 8 bits per character
  - Supports 1, 1.5 and 2 STOP bits
  - Supports even, odd and stick parity bits
  - Supports DMA function
  - Supports programmable baud rate
  - Supports modem control interface
  - Supports complete status reporting capabilities
  - Supports line breaks, parity errors, framing errors and data overrun detection

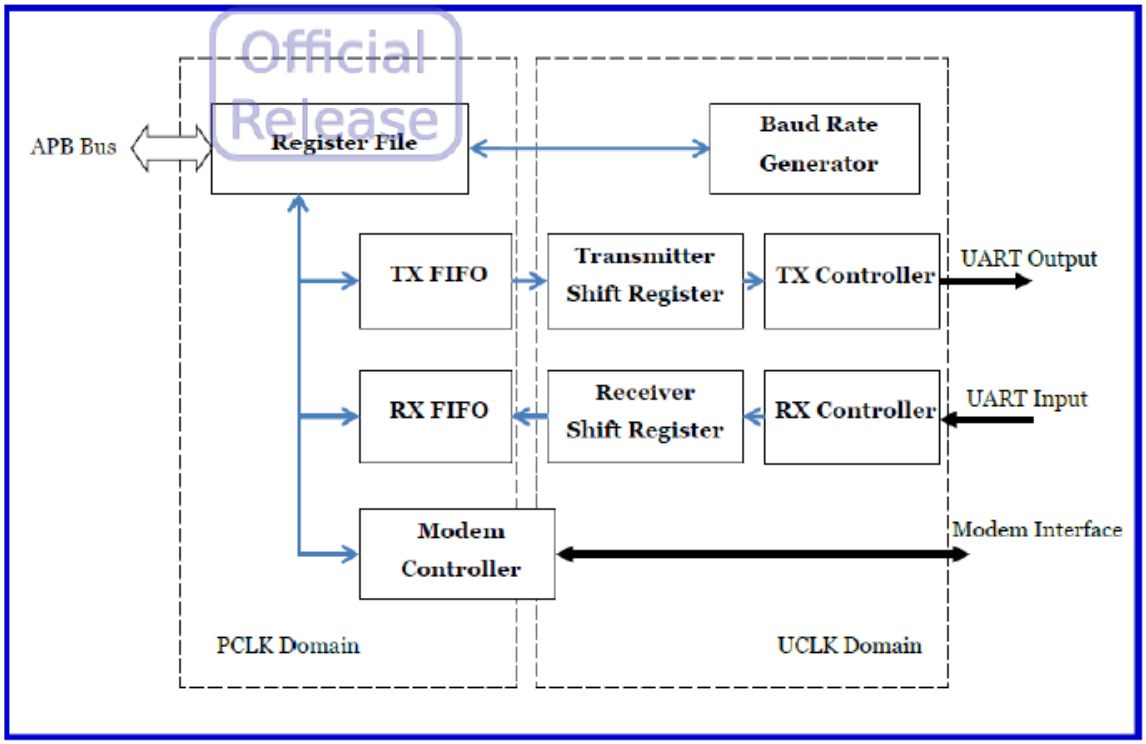


Fig. 6.1 ATCUART100 Block Diagram

6.1.3 Transmitter

The transmitter comprises a Transmitter FIFO (TX FIFO), a Transmitter Shift Register (TSR), and a Transmitter Controller (TX controller). The TX FIFO holds data to be transferred through the serial interface. The TX FIFO can store up to 128 characters depending on hardware configurations and programming settings. The TSR reads a character from the TX FIFO for the next transmission. The TSR functions as a parallel-to-serial data converter, converting the outgoing character to serial bit streams.

For each character transmission, the TX Controller generates a START bit, an optional parity bit, and some number of STOP bits. The generation of parity bit and STOP bit can be programmed by the Line Control Register. The TX FIFO is by default a one entry buffer called Transmitter Holding Register (THR). It needs to be enabled to work in multi-entry FIFO mode (FIFOE in the FIFO Control Register).

6.1.4 Receiver

The receiver comprises a Receiver FIFO (RX FIFO), Receiver Shift Register (RSR), and a Receiver Controller (RX Controller). The RX Controller uses the oversampling clock generated by Baud Rate Generator to perform sampling at the center of each bit transmission. The received bits are shifted into the RSR for serial-to-parallel data conversion and the received character is stored into the RX FIFO.

The RX FIFO is by default a one entry buffer called the Receiver Buffer Register (RBR). It needs to be enabled to work in multi-entry FIFO mode (FIFOE in the FIFO Control Register) with up to 128 characters depending on hardware configurations. The RX controller also detects some error conditions for each data transmission including parity error, framing error, data overrun, or line break.

6.1.5 Baud Rate Generator

The Baud Rate Generator takes the UART clock (uclk) as the source clock and divides it by a divisor. The divisor value is 16-bit in size and stored in two separate programming registers, each holding an 8-bit value. The most significant byte (MSB) is held in the Divisor Latch MSB (DLM) register and the least significant byte (LSB) is held in the Divisor Latch LSB (DLL) register.

The ratio of the sampling clock frequency to the baud rate is the oversampling ratio, which is stored in the Over Sample Control Register (OSCR). The default value for OSCR is 16. This is typically good enough and do not need further adjustment.

The formula for the divisor value is as follows:

The divisor value = the frequency of uclk / (desired baud rate * OSCR)

6.2 Special Register Description

6.2.1 Summary of Registers:

The registers shown below are available in UART peripheral. They can be used to program the two available UART peripherals.

Table 6-1 Registers in UART peripheral

Offset	Name	Description
+0x00	IdRev	ID and Revision Register
+0x04~ 0x0C	-	Reserved
+0x10	Cfg	Hardware Configure Register
+0x14	OSCR	Over Sample Control Register
+0x18~0x1C	-	Reserved
+0x20		DLAB = 0
	RBR	Receiver Buffer Register (Read only)
	THR	Transmitter Holding Register (Write only)
		DLAB = 1
	DLL	Divisor Latch LSB
+0x24		DLAB = 0
	IER	Interrupt Enable Register
		DLAB = 1
	DLM	Divisor Latch MSB
+0x28	IIR	Interrupt Identification Register (Read only)
	FCR	FIFO Control Register (Write only)
+0x2C	LCR	Line Control Register
+0x30	MCR	Modem Control Register
+0x34	LSR	Line Status Register
+0x38	MSR	Modem Status Register
+0x3C	SCR	Scratch Register

6.2.2 Hardware configuration register:

This register defines options regarding UART controller.

Table 6-2 Hardware Configuration

Name	Bit	Type	Description	Reset
-	31:2	-	Reserved	0x0
FIFO_DEPTH	1:0	RO	The depth of RXFIFO and TXFIFO 0: 16-byte FIFO 1: 32-byte FIFO 2: 64-byte FIFO 3: 128-byte FIFO	Configuration dependent

6.2.3 Over sample control register:

The oversample control register defines clock ratio between the sampling block and baud rate.

Table 6-3 Over sample register

Name	Bit	Type	Description	Reset
-	31:5	-	Reserved	0x0
OSC	4:0	R/W	Over-sample control The value must be an even number; any odd value writes to this field will be converted to an even value. **OSC = 0**: The over-sample ratio is 32 **OSC ≤ 8**: The over-sample ratio is 8 **8 < OSC < 32**: The over sample ratio is OSC	0x10

6.2.4 Receiver buffer register (DLAB=0):

The RBR has two modes, the FIFO mode and the BUFFER mode. Bit0 of the FIFO Control Register (FIFOE) controls the selection between these two modes. When FIFOE is 1 (FIFO mode), the RBR is a RXFIFO. The depth of RXFIFO is configurable by the macro ATCUART100_FIFO_DEPTH, see The Depth of FIFO for details. When FIFOE is 0 (BUFFER mode), the RBR is just a byte buffer.

Table 6-4 Receiver buffer register

Name	Bit	Type	Description	Reset
-	31:8	-	Reversed	0x0
RBR	7:0	RO	Receive data read port	0x0

6.2.5 Transmitter buffer register (DLAB=0):

The THR has two modes, the FIFO mode and the BUFFER mode. FCR bit0 (FIFOE) controls the selection between these two modes. When FIFOE is 1 (FIFO mode), the THR is a TXFIFO. The depth of TXFIFO is configurable by the macro ATCUART100_FIFO_DEPTH, see The Depth of FIFO for details. When FIFOE is 0 (BUFFER mode), the THR is a byte buffer.

Table 6-5 Transmitter buffer register

Name	Bit	Type	Description	Reset
-	31:8	-	Reserved	0x0
THR	7:0	WO	Transmit data write port	0x0

6.2.6 Interrupt enable register (DLAB=0):

Description for each field in the interrupt register

Table 6-6 Interrupt enable register

Name	Bit	Type	Description	Reset
-	31:4	-	Reserved	0x0
EMSI	3	R/W	Enable modem status interrupt The interrupt asserts when the status of one of the following occurs: The status of modem_rin, modem_dcdn, modem_dsrn or modem_ctsn (If the auto-cts mode is disabled) has been changed. If the auto-cts mode is enabled (MCR bit4 (AFE) = 1), modem_ctsn would be used to control the transmitter.	0x0
ELSI	2	R/W	Enable receiver line status interrupt	0x0
ETHEI	1	R/W	Enable transmitter holding register interrupt	0x0
ERBI	0	R/W	Enable received data available interrupt and the character timeout interrupt 0: Disable 1: Enable	0x0

6.2.7 Divisor Latch LSB (DLAB=1):

The Divisor Latch holds the divisor value for generating the sampling clock from the UART clock source (uclk). The size of the Divisor Latch is 16 bits (two bytes) and this register holds the least significant byte of the Divisor Latch. Please see Baud Rate Generator for details. The valid value of the Divisor Latch should be between 1 and 65535 (216-1), inclusive.

Table 6-7 Divisor latch LSB

Name	Bit	Type	Description	Reset
-	31:8	-	Reserved	0x0
DLL	7:0	R/W	Least significant byte of the Divisor Latch	0x1

6.2.8 Divisor Latch MSB (DLAB=1):

The Divisor Latch holds the divisor value for generating the sampling clock from the UART clock source (uclk). The size of the Divisor Latch is 16 bits (two bytes) and this register holds the most significant byte of the Divisor Latch. Please see Baud Rate Generator for details. The valid value of the Divisor Latch should be between 1 and 65535 (216-1), inclusive.

Table 6-8 Divisor latch MSB

Name	Bit	Type	Description	Reset
-	31:8	-	Reserved	0x0
DLM	7:0	R/W	Most significant byte of the Divisor Latch	0x0

6.2.9 Interrupt identification register:

This register identifies the type of interrupt that occurred. Table 6-10 defines the interrupt ID (INTRID) of the interrupt that occurred. The lower nibble can be used to identify the interrupt type.

Table 6-9 Interrupt Identification Register

Name	Bit	Type	Description	Reset
-	31:8	-	Reserved	0x0
FIFOED	7:6	RO	FIFOs enabled These two bits are 1 when bit 0 of the FIFO Control Register (FIFOE) is set to 1.	0x0
-	5:4	-	Reserved	0x0
INTRID	3:0	RO	Interrupt ID See Table 13 for encodings	0x1

Table 6-10 Interrupt Identification Register

Interrupt Identification Register					Interrupt Type	Interrupt Source Description	Interrupt Reset Method
Bit3	Bit2	Bit1	Bit0	Priority Level			
0	0	0	1	None	None	None	None
0	1	1	0	1	Receiver line status	Overrun errors, parity errors, framing errors, or line breaks	Read the Line Status Register (LSR)
0	1	0	0	2	Received data available	If FIFOE is disabled, there is one received data available in the RBR. If FIFOE is enabled, the numbers of received data available reach the trigger level (RFIFOT). The interrupt signal will stay active until the number of data available becoming smaller than the trigger level.	Read the Receiver Buffer Register (RBR)
1	1	0	0	2	Character timeout	When FIFOE is enabled and no character have been removed from or input to receive FIFO and there is at least one character in receive FIFO during the last four character times.	Read the Receiver Buffer Register (RBR)
0	0	1	0	3	Transmitter Holding Register empty	If FIFOE is disabled, the 1-byte THR is empty. If FIFOE is enabled, the whole 16-byte transmit FIFO is empty.	Write the Transmitter Holding Register (THR) or Read the Interrupt Identification Register (IIR).
0	0	0	0	4	Modem status	The Modem Status Register (MSR) bit[3:0] is not 0. One of the following events occurred: Clear To Send (CTS), Data Set Ready (DSR), Ring Indicator (RI), or Data Carrier Detect (DCD)	Read the Modem Status Register (MSR)

6.2.10 FIFO Control Register:

Table 6-11 FIFO Control Register

Name	Bit	Type	Description	Reset
-	31:8	-	Reserved	0x0
RFIFOT	7:6	WO	Receiver FIFO trigger level Please refer to Table 15	0x0
TFIFOT	5:4	WO	Transmitter FIFO trigger level Please refer to Table 16	0x0
DMAE	3	WO	DMA enable 0: Disable 1: Enable	0x0
TFIFORST	2	WO	Transmitter FIFO reset Write 1 to clear all bytes in the TXFIFO and resets its counter. The Transmitter Shift Register is not cleared. This bit will automatically be cleared.	0x0
RFIFORST	1	WO	Receiver FIFO reset Write 1 to clear all bytes in the RXFIFO and resets its counter. The Receiver Shift Register is not cleared. This bit will automatically be cleared.	0x0
FIFOE	0	WO	FIFO enable Write 1 to enable both the transmitter and receiver FIFOs. The FIFOs are reset when the value of this bit toggles.	0x0

Table 6-12 Receiver FIFO trigger level

RFIFOT Value	RXFIFO Trigger Level			
	16-byte RXFIFO	32-byte RXFIFO	64-byte RXFIFO	128-byte RXFIFO
0	Not empty	Not empty	Not empty	Not empty
1	More than 3	More than 7	More than 15	More than 31
2	More than 7	More than 15	More than 31	More than 63
3	More than 13	More than 27	More than 55	More than 111

Table 6-13 Transmit FIFO trigger level

TFIFOT Value	TXFIFO Trigger Level			
	16-byte TXFIFO	32-byte TXFIFO	64-byte TXFIFO	128-byte TXFIFO
0	Not full	Not full	Not full	Not full
1	Less than 12	Less than 24	Less than 48	Less than 96
2	Less than 8	Less than 16	Less than 32	Less than 64
3	Less than 4	Less than 8	Less than 16	Less than 32

6.2.11 Line control register:

Chief register in regards to the data being transferred on the UART lines. Various parameters like word length, parity bits, DLAB, etc., can be controlled using this register. SPS and EPS are only used when the data bytes have parity bits. Table 6-14 shows various bits used in Line Control Register. Table 6-15 shows what different values in PEN, EPS and SPS bits signify in the terms of parity bits.

Table 6-14 Line Control Register

Name	Bit	Type	Description	Reset
-	31:8	-	Reserved	0x0
DLAB	7	R/W	Divisor latch access bit	0x0
BC	6	R/W	Break control	0x0
SPS	5	R/W	Stick parity 1: parity bit is constant 0 or 1, depending on bit4 (EPS). 0: disable the sticky bit parity. Please refer to Table 18.	0x0
EPS	4	R/W	Even parity select 1: even parity (an even number of logic-1 is in the data and parity bits) 0: old parity. Please refer to Table 18.	0x0
PEN	3	R/W	Parity enable When this bit is set, a parity bit is generated in transmitted data before the first STOP bit and the parity bit would be checked for the received data.	0x0

STB	2	R/W	Number of STOP bits 0: 1 bits 1: the number of STOP bit is based on the WLS setting When WLS = 0, STOP bit is 1.5 bits When WLS = 1, 2, 3, STOP bit is 2 bits	0x0
WLS	1:0	R/W	Word length setting 0: 5 bits 1: 6 bits 2: 7 bits 3: 8 bits	0x0

Table 6-15 Parity bit selection field

PEN (bit3)	SPS (bit5)	EPS (bit4)	Parity Bit
0	X	X	No parity bit
1	0	0	Parity is odd
1	0	1	Parity is even
1	1	0	Parity bit is always 1
1	1	1	Parity bit is always 0

6.2.12 Line status register:

This register reports the status of transmitter and receiver.

Table 6-16 Line Status Register

Name	Bit	Type	Description	Reset
-	31:8	-	Reserved	0x0
ERRF	7	RO	Error in RXFIFO In the FIFO mode, this bit is set when there is at least one parity error, framing error, or line break associated with data in the RXFIFO. It is cleared when this register is read and there is no more error for the rest of data in the RXFIFO.	0x0
TEMT	6	RO	Transmitter empty This bit is 1 when the THR (TXFIFO in the FIFO mode) and the Transmitter Shift Register (TSR) are both empty. Otherwise, it is zero.	0x1

THRE	5	RO	Transmitter Holding Register empty This bit is 1 when the THR (TXFIFO in the FIFO mode) is empty. Otherwise, it is zero. If the THRE interrupt is enabled, an interrupt is triggered when THRE becomes 1.	0x1
LBreak	4	RO	Line break This bit is set when the uart_sin input signal was held LOW for longer than the time for a full-word transmission. A full-word transmission is the transmission of the START, data, parity, and STOP bits. It is cleared when this register is read. In the FIFO mode, this bit indicates the line break for the received data at the top of the RXFIFO.	0x0
FE	3	RO	Framing error This bit is set when the received STOP bit is not HIGH. It is cleared when this register is read. In the FIFO mode, this bit indicates the framing error for the received data at the top of the RXFIFO.	0x0
PE	2	RO	Parity error This bit is set when the received parity does not match with the parity selected in the LCR[5:4]. It is cleared when this register is read. In the FIFO mode, this bit indicates the parity error for the received data at the top of the RXFIFO.	0x0
OE	1	RO	Overrun error This bit indicates that data in the Receiver Buffer Register (RBR) is overrun.	0x0
DR	0	RO	Data ready. This bit is set when there are incoming received data in the Receiver Buffer Register (RBR). It is cleared when all of the received data are read.	0x0

6.3 UART Virtual Window:

- Target window manager to show all the simulator supported devices. (*There are some exceptions and all devices shown here might not work, this will depend on the version of software*). From target manager, choose UART1 or UART2 based on the peripheral used in the program.

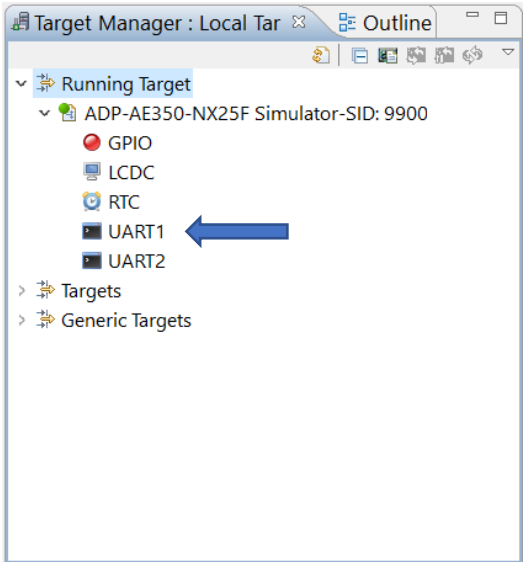


Fig. 6.2.1 Target Manager

- UART virtual window support in simulator to debug programs.
- FIGURE: A sequence of numbers transmitted through UART lines is displayed on this virtual window.

Fig. 6.2.2 Transmitted data on UART1

6.4 Programming:

6.4.1 Transmitter program:

This program simulates byte transfer on the UART lines, virtual window is used to show the data being transmitted.

```
#include<stdio.h>
#include<ae350.h>

void baud_initialize()
{
	/* Internal Clock = 50MHz
	 * OSCR = 16 (by default and dependent on Manufacturer)
	 * DLL & DLM Value = Internal Clock / (Required Baud Rate * OSCR)
	 * for 9600bps baud rate
	 * following are the values for DLL & DLM
	 * DLL = 0X46, DLM=0X01
	 */
```

```
    AE350_UART1->LCR=0x83;       //DLAB=1 to access DLL and DLM and 8 bit
                                 //transmission length
    AE350_UART1->DLL=0x46;
    AE350_UART1->DLM=0x01;       //define baud rate=9600
    AE350_UART1->LCR &=0x7F;
    //ANDing with 7F to make DLAB=0

}

void put_data(unsigned char a)
{
    AE350_UART1->THR = a;
    while(!(AE350_UART1->LSR & 0x20));     //wait until THR is empty
    while(!(AE350_UART1->LSR & 0x40));     //wait until transmitter is empty
}


int main(void)
{
    baud_initialize();
    for(int i=0;i<=9;i++)
    {
            put_data(0x30+i); //this will print numbers 0 to 9 on the
virtual window
    }
    return 0;
}
```

Output:

Fig.6.3 output of transmitter program

6.4.2 Receiver program:

This program simulates a byte being received in the buffer and the virtual window is used to input data into the buffer.

```
#include<stdio.h>
#include<ae350.h>

void baud_initialize()
{
            /* Internal Clock = 50MHz
             * OSCR = 16 (by default and dependent on Manufacturer)
             * DLL & DLM Value = Internal Clock / (Required Baud Rate * OSCR)
             * for 9600bps baud rate
             * following are the values for DLL & DLM
             * DLL = 0X46, DLM = 0X01
```

```
            */
    AE350_UART1->LCR=0x83;     //DLAB=1 to access DLL and DLM and 8 bit
                               //transmission length
    AE350_UART1->DLL=0x46;
    AE350_UART1->DLM=0x01;     //define baud rate=9600bps
    AE350_UART1->LCR &=0x7F;   //DLAB=0 for UART working
}

int main(void)
{
    baud_initialize();
    printf("%c \n",AE350_UART1->RBR);
    while(!(AE350_UART1->LSR&(0X01))); //checking receiver buffer
    printf("%c \n",AE350_UART1->RBR);
    return 0;
}
```

Procedure:

- Execute the program step-by-step to the point where it goes inside while loop.
- In the virtual window type a single letter; there will be no direct feedback on the window of what you have written.
- Upon typing a character, the program will come out of the while loop and execute the printf() statement.
- Observe the console because the character that was typed will be shown here

Output:

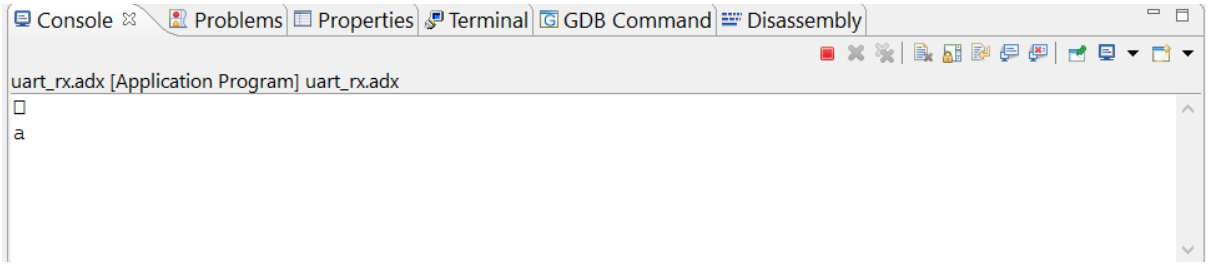


Fig. 6.4 'a' is printed on the console

6.4.3 Transmitter and receiver working together:

In this program, whatever will be typed on the virtual window will be printed back on it. This is done using the transmitter buffer and just a simple internal data transfer from receiver buffer. This program also helps in understanding the receiver functionality in a more realistic sense as there are no consoles in actual hardware but UART lines are.

```
#include<stdio.h>
#include<ae350.h>

void baud_initialize()
{

      /* Internal Clock = 50MHz
```

```
        * OSCR = 16 (by default and dependent on Manufacturer)
        * DLL & DLM Value = Internal Clock / (Required Baud Rate * OSCR)
        * for 9600bps baud rate
        * following are the values for DLL & DLM
        *
        */

    AE350_UART1->LCR=0x83;
    //DLAB=1 to access DLL and DLM and 8 bit word length
    AE350_UART1->DLL=0x46;
    AE350_UART1->DLM=0x01;     //define baud rate=9600bps
    AE350_UART1->LCR &=0x7F;     //DLAB=0 for UART working
}

void put_data(unsigned char a)
{
    AE350_UART1->THR = a;
    while(!(AE350_UART1->LSR & 0x20));     //wait until THR is empty
    while(!(AE350_UART1->LSR & 0x40));     //wait until transmitter is empty
}

int main(void)
{
      //baud rate set function
    baud_initialize();

    //stop transmitting after receiving enter key
    while(!(AE350_UART1->RBR == '\r'))
    {
      while(!(AE350_UART1->LSR&(0X01))); //checking receiver buffer for
data ready
      put_data(AE350_UART1->RBR); //sending the same data as output on the
transmitter
    }
    return 0;
}
```

Procedure:

Same as receiver program, the only difference being that the virtual window itself will be observed for output instead of the console.

Output:

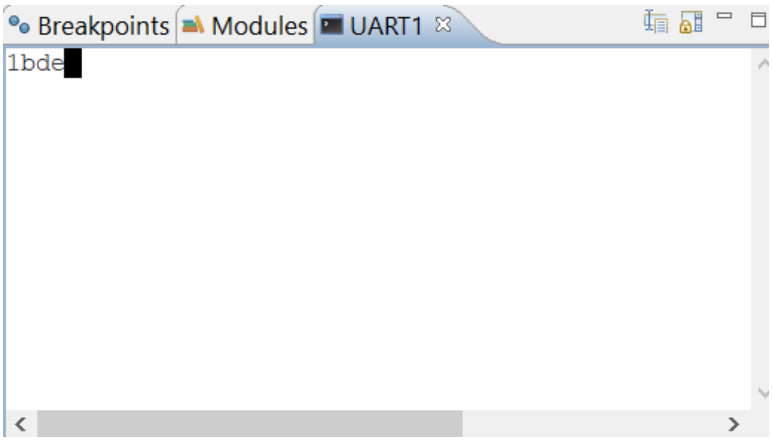


Fig. 6.5 Transmission of character that was typed

Chapter 7
Timer

7.1 Introduction

ATCPIT100 supports up to 4 PIT channels. Each PIT channel is a multi-function timer and provides the following 6 usage scenarios:

- one 32-bit timer
- two 16-bit timers
- four 8-bit timers
- one 16-bit PWM
- one 16-bit timer and one 8-bit PWM
- two 8-bit timers and one 8-bit PWM

An n-bit timer means the timer contains an n-bit counter to generate periodic interrupts.

Effective Devices of Channel Modes shows the effective devices of the corresponding channel modes.

Table 7-1 Timers available

Channel Mode	32-bit Timer	16-bit Timers	8-bit Timers	PWM	Mixed PWM/16-bit Timer	Mixed PWM/8-bit Timers
	32-bit Timer 0	16-bit Timer 0	8-bit Timer 0		16-bit Timer 0	8-bit Timer 0
		16-bit Timer 1	8-bit Timer 1			8-bit Timer 1
			8-bit Timer 2			
			8-bit Timer 3			
				16-bit PWM	8-bit PWM	8-bit PWM

7.2 Special Register Description

The following sections describe ATCPIT100 registers in detail.

Description of registers is given that were have used in sample program i.e., for 16-bit Timer of channel 0.

7.2.1 Channel Control Register

This register defines the functionality of the channel used along with the clock source for the channel

Table 7-2 Channel Control Register

Name	Bit	Type	Description	Reset
ChClk	3	R/W	Channel clock source: 0: External clock 1: APB clock	0x0
ChMode	2:0	R/W	Channel mode: 0: Reserved 1: 32-bit timer 2: 16-bit timers 3: 8-bit timers 4: PWM 5: Reserved 6: Mixed PWM/16-bit timer 7: Mixed PWM/8-bit timers	0x0

7.2.2 Channel Reload Register

The Reload Register keeps the initial/reload value(s) for timer counter(s). The definition of the Reload Register varies according to the channel mode. The period of timer is equal to the reload value plus one. For example, in the 16-bit Timer mode, a timer interrupt will be generated every (TMR16_1 + 1) cycles.

Table 7-3 Channel Reload Register for 16-bit Timers Mode (ChMode=2)

Name	Bit	Type	Description	Reset
TMR16_1	31:16	R/W	Reload value for 16-bit Timer 1	0x0
TMR16_0	15:0	R/W	Reload value for 16-bit Timer 0	0x0

7.2.3 Channel Enable Register

This registers enables the required channel and timer.

Table 7-4 Channel Enable Register

Name	Bit	Type	Description	Reset
-	31:16	-	Reserved	-
Ch3TMR3En/ CH3PWMEn	15	R/W	ChMode = 1, 2, 3 Channel 3 Timer 3 enable ChMode = 4, 6, 7 Channel 3 PWM enable	0x0
Ch3TMR2En	14	R/W	Channel 3 Timer 2 enable	0x0
Ch3TMR1En	13	R/W	Channel 3 Timer 1 enable	0x0
Ch3TMR0En	12	R/W	Channel 3 Timer 0 enable	0x0
Ch2TMR3En/ CH2PWMEn	11	R/W	ChMode = 1, 2, 3 Channel 2 Timer 3 enable ChMode = 4, 6, 7 Channel 2 PWM enable	0x0
Ch2TMR2En	10	R/W	Channel 2 Timer 2 enable	0x0
Ch2TMR1En	9	R/W	Channel 2 Timer 1 enable	0x0
Ch2TMR0En	8	R/W	Channel 2 Timer 0 enable	0x0
Ch1TMR3En/ CH1PWMEn	7	R/W	ChMode = 1, 2, 3 Channel 1 Timer 3 enable ChMode = 4, 6, 7 Channel 1 PWM enable	0x0
Ch1TMR2En	6	R/W	Channel 1 Timer 2 enable	0x0
Ch1TMR1En	5	R/W	Channel 1 Timer 1 enable	0x0
Ch1TMR0En	4	R/W	Channel 1 Timer 0 enable	0x0
Ch0TMR3En/ Ch0PWMEn	3	R/W	ChMode = 1, 2, 3 Channel 0 Timer 3 enable ChMode = 4, 6, 7 Channel 0 PWM enable	0x0
Ch0TMR2En	2	R/W	Channel 0 Timer 2 enable	0x0
Ch0TMR1En	1	R/W	Channel 0 Timer 1 enable	0x0
Ch0TMR0En	0	R/W	Channel 0 Timer 0 enable 0: Disable 1: Enable	0x0

7.2.4 Interrupt Enable Register

This register enables the interrupts on the corresponding channels.

Table 7-5 Interrupt Enable Register

Name	Bit	Type	Description	Reset
-	31:16	-	Reserved	-
Ch3Int3En	15	R/W	Channel 3 Timer 3 interrupt enable	0x0
CH3Int2En	14	R/W	Channel 3 Timer 2 interrupt enable	0x0
Ch3Int1En	13	R/W	Channel 3 Timer 1 interrupt enable	0x0
CH3Int0En	12	R/W	Channel 3 Timer 0 interrupt enable	0x0
Ch2Int3En	11	R/W	Channel 2 Timer 3 interrupt enable	0x0
Ch2Int2En	10	R/W	Channel 2 Timer 2 interrupt enable	0x0
Ch2Int1En	9	R/W	Channel 2 Timer 1 interrupt enable	0x0
Ch2Int0En	8	R/W	Channel 2 Timer 0 interrupt enable	0x0
Ch1Int3En	7	R/W	Channel 1 Timer 3 interrupt enable	0x0
Ch1Int2En	6	R/W	Channel 1 Timer 2 interrupt enable	0x0
Ch1Int1En	5	R/W	Channel 1 Timer 1 interrupt enable	0x0
Ch1Int0En	4	R/W	Channel 1 Timer 0 interrupt enable	0x0
Ch0Int3En	3	R/W	Channel 0 Timer 3 interrupt enable	0x0
Ch0Int2En	2	R/W	Channel 0 Timer 2 interrupt enable	0x0
Ch0Int1En	1	R/W	Channel 0 Timer 1 interrupt enable	0x0
Ch0Int0En	0	R/W	Channel 0 Timer 0 interrupt enable 0: Disable 1: Enable	0x0

7.2.5 Interrupt Status Register

This register tells whether the interrupt occurred due to time up of a given timer. Interrupt of only those timers is considered whose interrupt was enabled from interrupt enable register.

Table 7-6 Interrupt Status Register

Name	Bit	Type	Description	Reset
-	31:16	-	Reserved	-
Ch3Int3	15	W1C	Channel 3 Timer 3 interrupt status	0x0
Ch3Int2	14	W1C	Channel 3 Timer 2 interrupt status	0x0
Ch3Int1	13	W1C	Channel 3 Timer 1 interrupt status	0x0
Ch3Int0	12	W1C	Channel 3 Timer 0 interrupt status	0x0
Ch2Int3	11	W1C	Channel 2 Timer 3 interrupt status	0x0
Ch2Int2	10	W1C	Channel 2 Timer 2 interrupt status	0x0
Ch2Int1	9	W1C	Channel 2 Timer 1 interrupt status	0x0
Ch2Int0	8	W1C	Channel 2 Timer 0 interrupt status	0x0
Ch1Int3	7	W1C	Channel 1 Timer 3 interrupt status	0x0
Ch1Int2	6	W1C	Channel 1 Timer 2 interrupt status	0x0
Ch1Int1	5	W1C	Channel 1 Timer 1 interrupt status	0x0
Ch1Int0	4	W1C	Channel 1 Timer 0 interrupt status	0x0
Ch0Int3	3	W1C	Channel 0 Timer 3 interrupt status 0: No effect 1: Timer 3 time up	0x0
Ch0Int2	2	W1C	Channel 0 Timer 2 interrupt status 0: No effect 1: Timer 2 time up	0x0
Ch0Int1	1	W1C	Channel 0 Timer 1 interrupt status 0: No effect 1: Timer 1 time up	0x0
Ch0Int0	0	W1C	Channel 0 Timer 0 interrupt status 0: No effect 1: Timer 0 time up	0x0

7.3 Programming

Using channel 0 in channel mode=2 (16-bit Timer-1) to generate an interrupt after every 1000 APB cycles.

```
#include <stdio.h>
#include <stdlib.h>
#include <ae350.h>

int main(void)
{
	int i=0;
	AE350_PIT->CHANNEL->CTRL=0x0A;
	//channel mode 2:16-bit timer, chclk-ABP clock
	AE350_PIT->CHANNEL->RELOAD=0x03E70000;
	//Interrupt will occur after every 999+1=1000 ABP Clk cycles

	AE350_PIT->INTEN=0x02;//Channel-0 Timer-1 interrupt enable
	AE350_PIT->CHNEN=0X02;//Channel-0 Timer-1 enable
	while(1)
	{
		i++;
		if(AE350_PIT->INTST & 0x02)//Monitor Channel-0 Timer-1 interrupt status
		{
			printf("timer 1 times up ---- %d \n",i);
			AE350_PIT->INTST=0x02;
			//Clear interrupt status register by writing 1 because
			//this register is W1C type.
		}
		else
			printf("timer on ---- %d \n",i);
	}
}
```

Output:

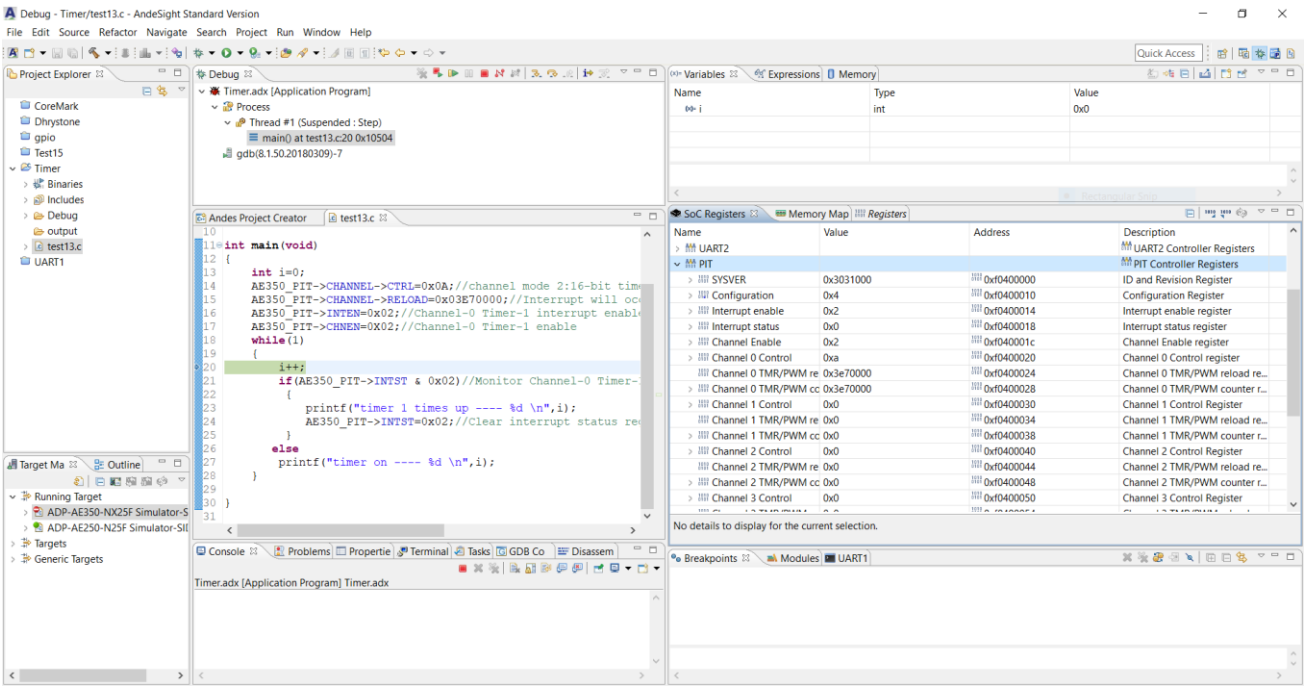


Fig 7.1 Debugging the program and observing the PIT Register values.

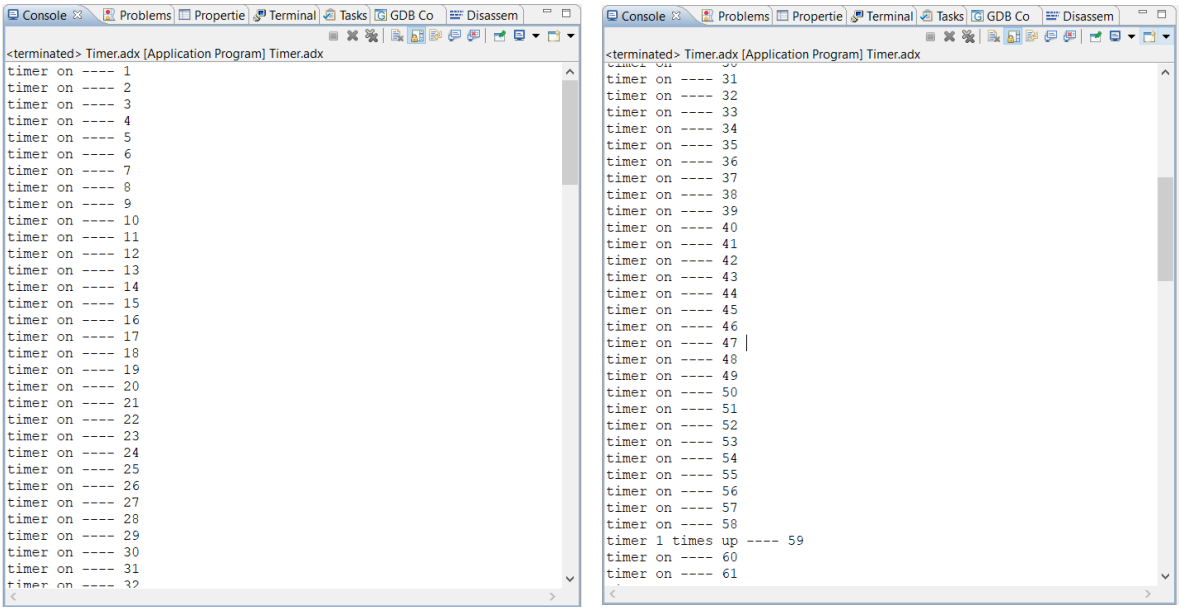


Fig 7.2 Output of Sample Program

Chapter 8
Case Study

8.1 Overview

Previous chapters build the fundamentals and covered programming of GPIO, UART and PIT. This chapter covers an application developed using the three peripherals.

The program is a basic “command line” type GUI. It uses UART1 to provide the necessary instructions that the processor needs to perform and the output is observed on UART2. The tasks have been embedded and the user just needs to press 1 through 4 to perform a specific task. Pressing 0 ends the execution.

The modes are described as follows:

a) **Mode 1** – Generates *m* digit *n* random numbers. The m and n variables are to be provided by the user.
b) **Mode 2** – Enters equipment controller mode for GPIO. This mode is used to toggle ON or OFF the connected devices to GPIO. The program uses bit[0] and bit[1] of GPIO to toggle bit[13] and bit[14] respectively. Bit[2] exits the controller mode and returns user to console mode on UART1 awaiting for input.
c) **Mode 3** – Sets a timer to remind the user after a fixed interval of time. This mode always asks the user for the input time interval on entry. A message “REMINDER” is print on UART2.
d) **Mode 4** – Generates a predefined LED pattern on GPIO port bit[15] through bit[31].

8.2 Program

```
#include<stdio.h>
#include<ae350.h>
#include<time.h>
#include<string.h>
#include<math.h>

#define BACK_SPACE                  0x08
#define NEW_LINE                    0x0A
#define DOLLAR_SIGN                 0x24
#define PIT_CHNCTRL_TMR_32BIT       1
#define PIT_CHNCTRL_CLK_PCLK        (1 << 3)

void uart_init()
{
      /* Internal Clock = 50MHz
       * OSCR = 16 (by default and dependent on Manufacturer)
       * DLL & DLM Value = Internal Clock / (Required Baud Rate * OSCR)
       * for 9600bps baud rate
       * following are the values for DLL & DLM
       *
       */

    AE350_UART2->LCR=0x83;       //DLAB=1 to access DLL and DLM and 8 bit
                                 //transmission length
```

```
    AE350_UART2->DLL=0x46;
    AE350_UART2->DLM=0x01;      //define baud rate=9600
    AE350_UART2->LCR &=0x7F;
    //ANDing with 7F to make DLAB=0

    AE350_UART1->LCR=0x83;      //DLAB=1 to access DLL and DLM and 8 bit
                                    //transmission length
    AE350_UART1->DLL=0x46;
    AE350_UART1->DLM=0x01;      //define baud rate=9600
    AE350_UART1->LCR &=0x7F;

}

void print_str_uart2(const char *str)
{
      unsigned char c;
      int i;
      //printf("%d \n",strlen(str));
      put_char2(NEW_LINE);
      for(i=0;i<strlen(str);i++)
      {
            c = *(str+i);
            put_char2(c);
      }
      put_char2(NEW_LINE);

      put_bs_uart2(strlen(str));

}

void print_str_uart1(const char *str)
{
      unsigned char c;
      int i;
      //printf("%d \n",strlen(str));
      put_char1(NEW_LINE);
      for(i=0;i<strlen(str);i++)
      {
            c = *(str+i);
            put_char1(c);
      }
      put_char1(NEW_LINE);

      put_bs_uart1(strlen(str));

}

void put_char2(unsigned char a)
{
    AE350_UART2->THR = a;
    while(!(AE350_UART2->LSR & 0x20));    //wait until THR is empty
    while(!(AE350_UART2->LSR & 0x40));    //wait until transmitter is empty
}

void put_char1(unsigned char a)
{
    AE350_UART1->THR = a;
    while(!(AE350_UART1->LSR & 0x20));    //wait until THR is empty
    while(!(AE350_UART1->LSR & 0x40));    //wait until transmitter is empty
}
```

```
void put_bs_uart2(int n)
{
      int i;
      for(i=n;i>0;i--)
         put_char2(BACK_SPACE);
}

char take_char1()
{
      while(!(AE350_UART1->LSR&(0X01))); //checking receiver buffer
      put_char1(AE350_UART1->RBR);

      return (AE350_UART1->RBR);
}

void put_bs_uart1(int n)
{
      int i;
      for(i=n;i>0;i--)
         put_char1(BACK_SPACE);
}


void random_gen()
{
      int digits, num;
      //random seed will generate random numbers at each execution
      srand(time(0));
      put_char2(NEW_LINE);

      print_str_uart1("How many digits? (1 to 9)");
      put_char1(NEW_LINE);
      put_bs_uart1(25);

      digits=(take_char1()-0x30);
      put_bs_uart1(1);

      print_str_uart1("How many numbers? (1 to 9)");
      put_char1(NEW_LINE);
      put_bs_uart1(26);

      num=(take_char1()-0x30);
      put_bs_uart1(1);

      int i;
      for(int j=1;j<=num;j++)
      {
            for(i=1;i<=digits;i++)
            {
                  put_char2(0x30+(rand() % 10));
                  //this will print random numbers on the virtual window
                  //%10 limits the output from 0 to 9
                  //0x30 is for ASCII value adjustment
            }

      put_char2(NEW_LINE);

    put_bs_uart2(digits);

      }
}
```

```
void timer_init()
{

	//channel mode 1:32-bit timer
	//Clock source : APB clock
	AE350_PIT->CHANNEL[0].CTRL = (PIT_CHNCTRL_TMR_32BIT | PIT_CHNCTRL_CLK_PCLK);

	//disable all interrupts
	AE350_PIT->INTEN |= (0UL << 0);

	//interrupt enable for channel 0, timer 0
	AE350_PIT->INTEN |= (1UL << 0);

}


void remind_me()
{
	int input[4]={0};
	unsigned int a=0;
	int i,k;
	char tempc;
	print_str_uart1("Enter time");
	put_char1(NEW_LINE);
	put_bs_uart1(10);

	for(i=0;i<=3;i++)
	{
		tempc=take_char1();
		printf("%c",tempc);
		if(tempc == '$')
			break;
		input[i]=(tempc-0x30);
		printf("%d\t",input[i]);
	}

	put_char1(NEW_LINE);
	put_bs_uart1(i+1);
	print_str_uart1("Done!");
	put_char1(NEW_LINE);
	put_bs_uart1(5);

	for(k=0;k<i;k++)
	{
		a+=(int)(input[i-1-k]*pow(10,k));
	}
	printf("%d",a);

	//AE350_PIT->INTST = 0x01;

	//reload register value + 1 = initial timer value
	AE350_PIT->CHANNEL[0].RELOAD=a;

	//channel 0 timer 0 enable
	AE350_PIT->CHNEN |= (1UL << 0);


	while(1)
	{
		printf("timer value ---- 0x%x \n",AE350_PIT-=>CHANNEL[0].COUNTER);
```

```
		if(AE350_PIT->INTST & (1UL << 0))
		{
			print_str_uart2("REMINDER!");
			AE350_PIT->INTST |= (1<<0);
			put_char2(NEW_LINE);
			put_bs_uart2(9);

			//timer disable
			AE350_PIT->CHNEN |= (0UL << 0);
			//it won't be disabled by this statement for some reason

			break;
		}
	}

}

void gpio_input_init()
{
	AE350_GPIO->CHANNELDIR=0x0000FF00;
	//configure GPIO channels 0 to 7 as input and 8 to 15 as output
	AE350_GPIO->PULLEN=0xFF;
	//enable pull-type for channels 0 to 7
	AE350_GPIO->PULLTYPE=0x00;
	//the channels 0 to 7 has pull-up type
	AE350_GPIO->DEBOUNCEEN=0xFF;
	//enable de-bounce for channels 0 to 7
	AE350_GPIO->DEBOUNCECTRL=0x80000003;
	//pclk as de-bounce clock source and use filter to remove glitches
}

void gpio_interrupt_init()
{
	AE350_GPIO->INTREN=0x07;	//enable channels 0,1 and 2 interrupts
	AE350_GPIO->INTRMODE0=0x666;
	/* channel0 interrupt mode as positive edge
	 * channel1 interrupt mode as positive edge
	 * channel2 interrupt mode as positive edge
	 */
}

void interrupt()
{
	//initialization required each time this mode is called
	//as two different types of GPIO functionalities are present in the
	program
	gpio_input_init();
	gpio_interrupt_init();
	AE350_GPIO->INTRSTATUS=0x07;		//clear all the interrupts
	while(AE350_GPIO->INTRSTATUS==0);	//wait until interrupt occurs
	while(1)
	{
		if(AE350_GPIO->INTRSTATUS == 0x01)
		//if channel0 interrupt occurred
		{
			switch (AE350_GPIO->DATAOUT&(1<<13))
			{
				case (1<<13):
					AE350_GPIO->DOUTCLEAR=(1<<13);
					break;
				case (0<<13):
```

```
                        AE350_GPIO->DOUTSET=(1<<13);
                        break;
                  default:
                        break;
                }
                AE350_GPIO->INTRSTATUS=0x01;  //write 1 to clear the flag

            }
            if(AE350_GPIO->INTRSTATUS == 0x02)
            //if channel1 interrupt occurred
            {
                switch (AE350_GPIO->DATAOUT&(1<<14))
                {
                  case (1<<14):
                        AE350_GPIO->DOUTCLEAR=(1<<14);
                        break;
                  case (0<<14):
                        AE350_GPIO->DOUTSET=(1<<14);
                        break;
                  default:
                        break;
                }
                AE350_GPIO->INTRSTATUS=0x02;  //write 1 to clear the flag

            }
            if(AE350_GPIO->INTRSTATUS == 0x04)
            //if channel2 interrupt occurred
            {
                AE350_GPIO->INTRSTATUS=0x04;  //write 1 to clear the flag
                break;
            }
      }
}

void gpio_pattern()
{
      int i,j;
      AE350_GPIO->CHANNELDIR=0xFFFF0000;
      //configure all the GPIO channels as output

      //infinite loop will result into continuous blink of an LED
    for(int k=0;k<=2;k++)
    {
        for(j=16;j<=31;j++)
            {
                AE350_GPIO->DOUTSET|=(1<<j);   //creates the required pattern
                for(i=0;i<20000;i++);          //delay
            }
        for(j=31;j>=16;j--)
        {
            AE350_GPIO->DOUTCLEAR|=(1<<j);//creates the required pattern
            for(i=0;i<20000;i++);         //delay
        }
    }
}

int main(void)
{
      unsigned char MODE;
      uart_init();
      timer_init();
```

```
	exit_switch:

	print_str_uart1("Enter mode number");
	MODE=take_char1();
	put_char1(NEW_LINE);
	put_bs_uart1(1);

	switch(MODE)
	{
		case '1':
			random_gen();
			goto exit_switch;
		case '2':
			print_str_uart2("Equipment controller mode");
			interrupt();
			print_str_uart2("Equipment controller stopped");
			goto exit_switch;
		case '3':
			remind_me();
			goto exit_switch;
		case '4':
			gpio_pattern();
			goto exit_switch;
		case '0':
			break;
		default:
			print_str_uart1("not a valid number");
			goto exit_switch;
	}

    return 0;
}
```

Output:

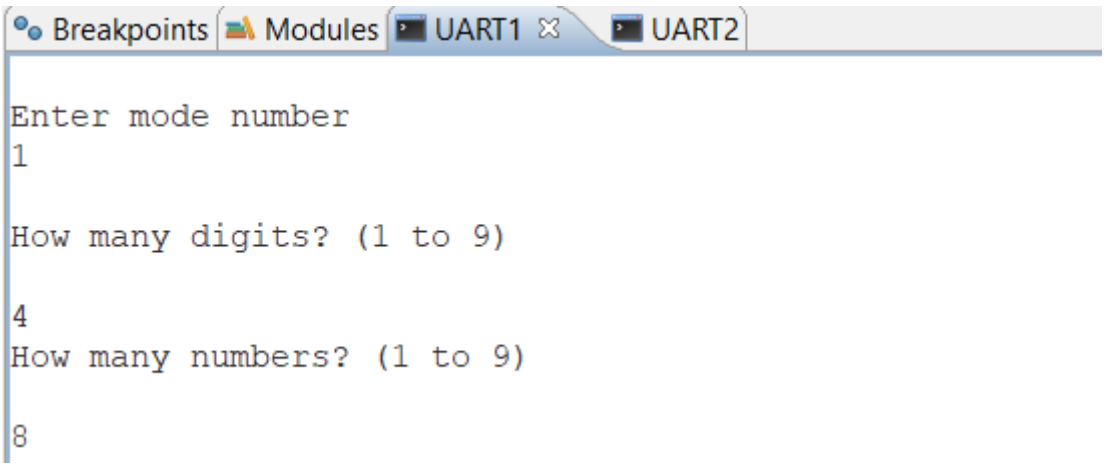


Fig. 8.1 Mode 1 on UART1

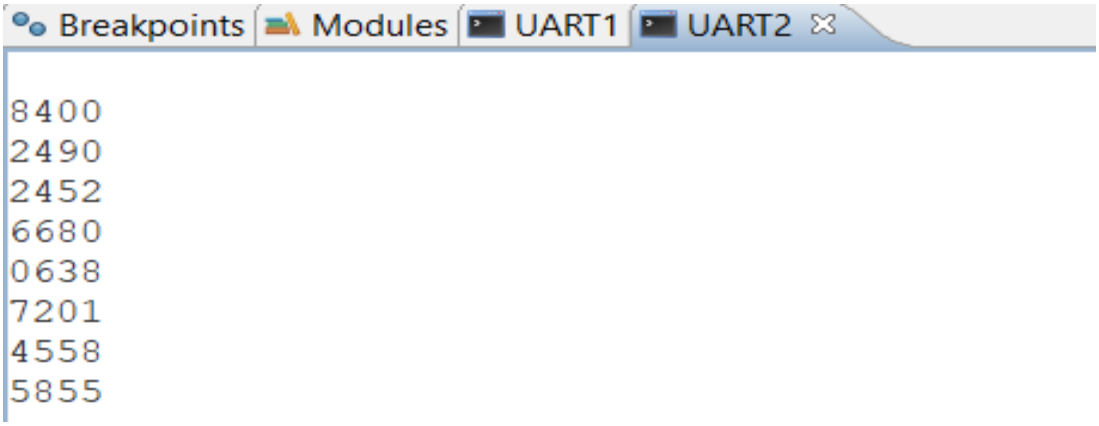


Fig. 8.2 Equivalent output of Mode 1 on UART2

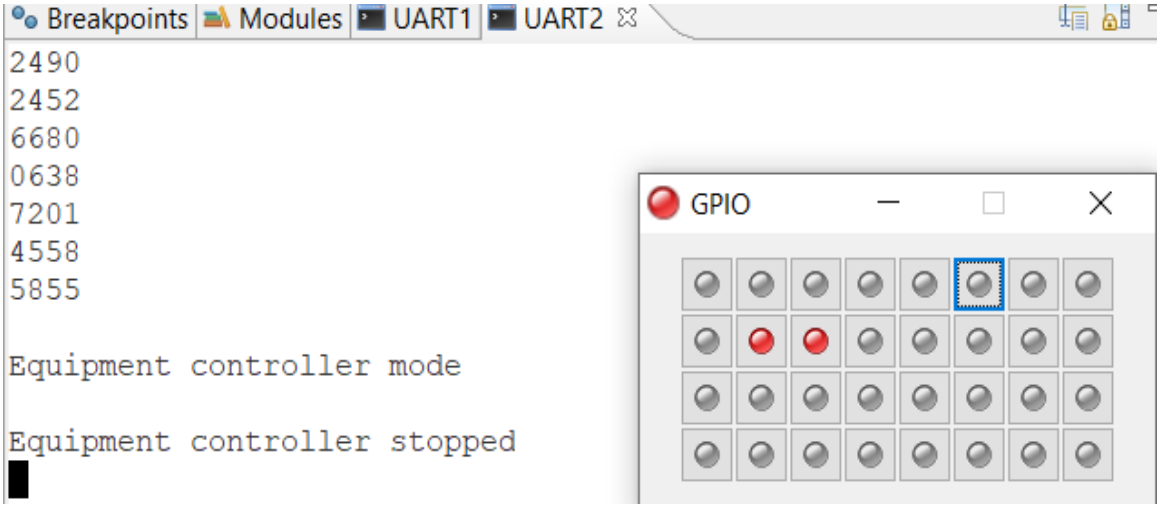


Fig. 8.3 Mode 2 output on UART2 and GPIO

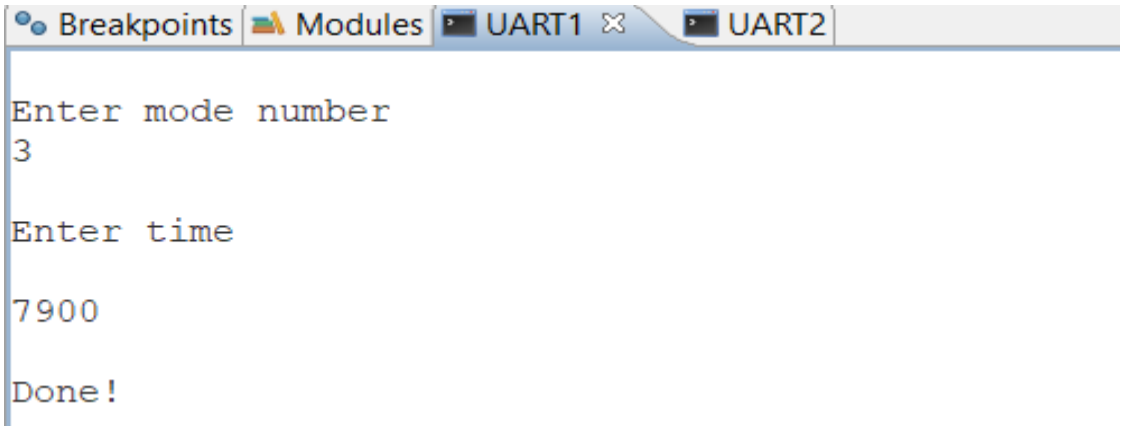


Fig. 8.4 Mode 3 on UART1

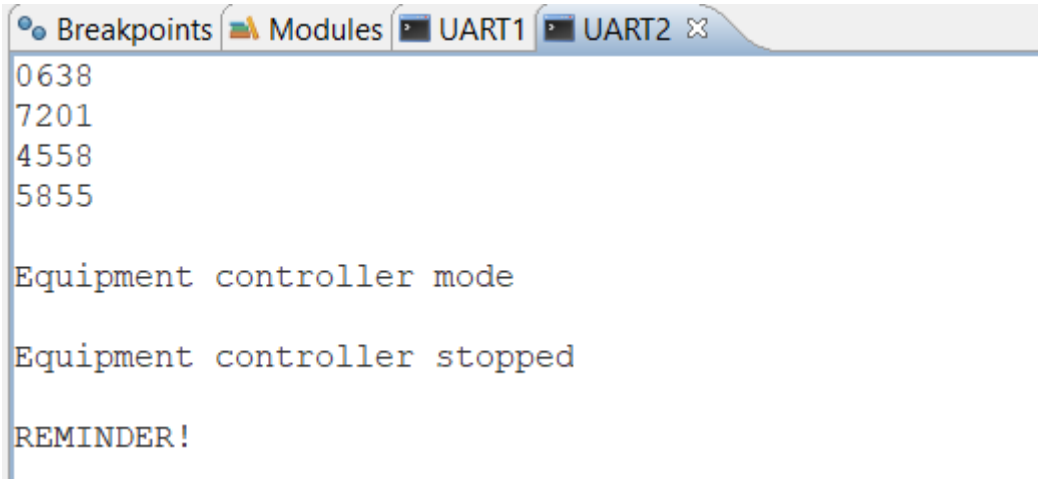


Fig. 8.5 Output of Mode 3 on UART2

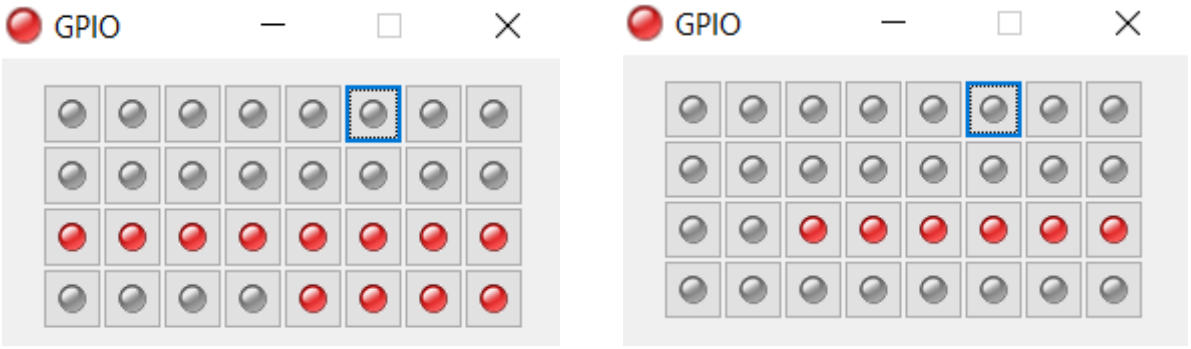


Fig. 8.6 Preset pattern of Mode 4 on GPIO

LIMITATIONS

The eclipse based IDE is at early stage of its development and hence has limitations associated with it. In the project, the following limitations were encountered:

- The GPIO GUI doesn't allow input at more than one ports at once.
- There is no GUI support for I2C and SPI in the simulations. This restricts us from using either of these protocols.
- The clock of RTC module needs to synchronize with the clock of the microcontroller. This is done when the "write bit" contains value 1. However, in the simulator, this bit never switched from 0 to 1 hence resulting in gibberish output at RTC. This rendered the RTC module useless for simulation.
- In PIT (when the value of counter is printed on the console) instead of printing the current value, the simulator prints the value that was reload into the timer. However, the timer is working as intended.
- In order to use the software interrupts or IRQs, the startup file needs to be modified. However, the software did not show the startup file anywhere in the project folder. This obstructed us from using any type of IRQs and we had to use the polling techniques to check for any interrupt request.
- The absence of any waveform viewer limited us from checking the PWM feature available in the PIT module.

Unavailability of the resources makes it difficult for a novice to write an application using RISC V on an eclipse based IDE. Because of the high skill floor, it is difficult for RISC V to attract new system designers.

CONCLUSION

With RISC V being in its early stages of development, the associated IDEs do not provide as many features as their competitive counterparts. Due to this the novice developers are unable to unlock the full potential of the architecture. Nevertheless, the IDE supports basic functionalities that allows the developers to get a gist of the architecture. RISC V has a promising future because of high accessibility, flexibility, and power & area efficiency but has a long way to go before it can compete with the likes of x86, ARM architectures.

Our project helps in solidifying the embedded concepts using RISC V and provides fundamental knowledge of the architecture. One can quickly pick up the IDE, get familiarized with it and start developing an application.

FUTURE WORK

As the eclipse IDE matures, it will incorporate all the features that Keil IDE offers for ARM. A better GUI support along with capability to modify startup file would allow better integration and easier development of applications using RISC V architecture.

Once the GUI support of various peripherals is available in the IDE, we can build an application around aforementioned protocols. In the current stage of IDE, the interrupts can only be identified with the polling technique. Only non-time critical applications can be built. As the IDE will allow implementation of IRQs and SWIs in a more transparent way, the real time applications will be easily built and simulated.

REFERENCES

[1] AndeSight_STD_V3.2_USER_MANUAL

[2] ATGCGPIO100_N25_REFERENCE_MANUAL

[3] RISC-V instruction RISC-V GitHub page (privileged and unprivileged sets)

[4] AndeSight_STD_v3.2_User_Manual_UM186_V1.5

[5] AndesCore_NX25(F)_DS131_V2.4