EMBEDDED SYSTEMS LAB
MANUAL
(On ARM microcontroller and RTOS)

I Year M.Tech I - sem

Prepared
By
Ch. Madhuri Devi
Asso.Prof
Easari. Parushuramu
Asst.Proff

Under
Prof. K. Ashok Babu
HOD of ECE Dept.

SriIndu College of Engineering and Technology
(an Autonomous Institution under JNTUH)
SHERIGUDA (V), IBRAHIMPATNAM(M), R.R.DIST, A.P. – 501 510.
☎ : 08414-224166 Mobile No. : 9347054999, 9347363999 Fax No.: 08414-224177
LPC2148 (ARM) MICROCONTROLLER
Contents

1. Introduction to ARM Board (LPC2148)
   Features of the ARM Microcontroller
   Brief overview of ARM Architecture
   ARM operating modes
   System initialisation(Runtime environment)
   ARM Board Description

2. KIEL µVision 4 tool
   Cross compiler/Assembler
   ARM tools
   Writing programs
   Project structure
   Make Utility and Make file
   Building applications

3. Downloading the hex file onto Arm microcontroller
   Downloading Hex file onto target

   Issues in real time operating systems design
   Important characteristics of Real time operating systems
   uC/OS-II RTOS
   Comparison of various real time operating systems

5. Serial Communication Drivers for ARM Processors
   Study of serial communication architecture on LPC2148 ARM architecture
   Implementing serial communication drivers for ARM processors
   Development of echo program for Serial Communication

6. Software Development for DSP Applications
   Studying the basic DSP platform requirements
   Implementing the FIR filter
   Implementation of CORDIC DSP algorithm
Section - 1  

Introduction to ARM Board (LPC2148)

This section of the document introduces LPC2148 microcontroller board based on a 16-bit/32-bit ARM7TDMI-S CPU with real-time emulation and embedded trace support, that combine microcontrollers with embedded high-speed flash memory ranging from 32 kB to 512 kB. A 128-bit wide memory interface and unique accelerator architecture enable 32-bit code execution at the maximum clock rate. For critical code size applications, the alternative 16-bit Thumb mode reduces code by more than 30% with minimal performance penalty. The meaning of LPC is Low Power Low Cost microcontroller. This is 32 bit microcontroller manufactured by Philips semiconductors (NXP).

Due to their tiny size and low power consumption, LPC2148 is ideal for applications where miniaturization is a key requirement, such as access control and point-of-sale.

1.1 Features of ARM Microcontroller

- 16-bit/32-bit ARM7TDMI-S microcontroller in a tiny LQFP64 package.
- 8 kB to 40 kB of on-chip static RAM and 32 kB to 512 kB of on-chip flash memory; 128-bit wide interface/accelerator enables high-speed 60 MHz operation.
- In-System Programming/In-Application Programming (ISP/IAP) via on-chip boot loader software, single flash sector or full chip erase in 400 ms and programming of 256 Bytes in 1 ms. Embedded ICE RT and Embedded Trace interfaces offer real-time debugging with the on-chip Real Monitor software and high-speed tracing of instruction execution.
- USB 2.0 Full-speed compliant device controller with 2kB of endpoint RAM. In addition, the LPC2148 provides 8 kB of on-chip RAM accessible to USB by DMA.
- One or two (LPC2141/42 vs, LPC2144/46/48) 10-bit ADCs provide a total of 6/14 analog inputs, with conversion times as low as 2.44 ms per channel.
- Single 10-bit DAC provides variable analog output (LPC2148 only)
- Two 32-bit timers/external event counters (with four capture and four compare channels each), PWM unit (six outputs) and watchdog.
• Low power Real-Time Clock (RTC) with independent power and 32 kHz clock input.
• Multiple serial interfaces including two UARTs, two Fast I2C-bus (400 kbit/s), SPI and SSP with buffering and variable data length capabilities.
• Vectored Interrupt Controller (VIC) with configurable priorities and vector addresses.
• Up to 45 of 5 V tolerant fast general purpose I/O pins in a tiny LQFP64 package.
• Up to nine edge or level sensitive external interrupt pins available.
• 60 MHz maximum CPU clock available from programmable on-chip PLL with settling time of 100 ms.
• Power saving modes include Idle and Power-down
• Individual enable/disable of peripheral functions as well as peripheral clock scaling for additional power optimization.
• Processor wake-up from Power-down mode via external interrupt or BOD.
• Single power supply chip with POR and BOD circuits:
  - CPU operating voltage range of 3.0 V to 3.6 V (3.3 V ± 10 %) with 5 V tolerant I/O.

1.2 Brief overview of ARM7 Architecture

The ARM7TDMI core is a 32-bit embedded RISC processor delivered as a hard macrocell optimized to provide the best combination of performance, power and area characteristics. The ARM7TDMI core enables system designers to build embedded devices requiring small size, low power and high performance.

The ARM7 family also includes the ARM7TDMI processor, the ARM7TDMI-S processor, the ARM720T processor and the ARM7EJ-S processors, each of which has been developed to address different market requirements.

The market for microprocessors continues to diversify, based on the evolving demands of applications including wireless, home entertainment, automotive and microcontrollers. ARM core families sharing the ARMv7 architecture will cover the widening spectrum of embedded processing.
The ARM architecture is based on Reduced Instruction Set Computer (RISC) principles. The RISC instruction set and related decode mechanism are much simpler than those of Complex Instruction Set Computer (CISC) design. This simplicity gives:

- A high instruction throughput
- An excellence real-time interrupts response
- A small, cost-effective, processor macrocell

The ARM7TDMI core is the industry’s cost widely used 32-bit embedded RISC microprocessor solution. Optimized for cost and power-sensitive application, the ARM7TDMI solution provides low power consumption, small size, and high performance needed in portable, embedded application.

The ARM7DMI-S is synthesizable version of ARM7TDMI core. The ARM720T hard macrocell contain the ARM7DMI core, 8KB unified cache and MMU (Memory Management Unit) that allows the use of protected execution space and virtual memory.

The ARM7EJ-S processor is synthesizable core that provides all the benefit of ARM7DMI, while also incorporating ARM’s latest DSP extensions and jazelle technology, enabling acceleration of Java-based applications.

**Architecture:**

The ARM7 core is based on the von Neumann architecture with 32-bit data bus that carries both instruction and data. Data can be of 8 bits, 16 bits, 32 bits. It has following features:

- Instruction pipeline
- Memory format
- Operation modes
- Coprocessor
- Debugging feature

**Instruction pipeline:**

The ARMv7 core uses a three stage pipeline to increase the flow of instructions to the processor. This allows multiple simultaneous operations to take place and
continuous operations and memory systems. The instructions are executed in three stages:

· Fetch
· Decode
· Execute

During normal operation, while one instruction is being executed, its successor is being decoded, and third instruction is being fetched from the memory. The program counter (PC) value used in an executing instruction is always two instructions ahead of the address.

Memory Format:

The ARM7 memory interface is designed to allow optimum performance potential and minimize memory usage. Speed critical control signals are pipelined to allow system control function to exploit the fast burst access modes supported by many memory technologies. ARM7 has four basic types of cycle:

· Internal
· Non sequential
· Sequential
· Coprocessor transfer

The ARM7 can be configured to store the words as either in little-endian or big-endian format.

The ARM7 processor supports the following data types:

· Word, 32-bit
· Half word, 16-bit
· Byte, 8-bit

You must align this as follow:

· Word quantities must be aligned to four-byte boundaries.
· Half word quantities must be aligned to two-byte boundaries.
· Byte quantities can be placed on any boundary.

The ARM core supports two operating states and instruction sets

- ARM state for 32 bit word aligned instruction
- Thumb state for 16-bit half word aligned instruction
Operating modes:

The ARMv7 core has seven modes of operation:

- User mode – normal ARM program execution mode and used for executing most application programs.
- Fast Interrupt (FIQ) – mode supports data transfer or channel processes to allow very fast interrupt
- Interrupt (IRQ) – mode is used for general purpose interrupt handling.
- Supervisor (SVC) – is protected mode for operating system.
- Abort (ABT) – mode is entered after a data or instruction fetch is aborted.
- Undefined (UND) – mode is entered when an undefined instruction is executed.
- System (SYS) – is a privileged user mode for the operating system.

Modes other than user mode are collectively known as privileged modes. Privileged modes are used to service interrupts or exceptions, or to access protected resources.

The ARMv7 has 37 registers all are 32bit wide, not all the registers are available for a given modes. R15 is program counter. R14 is link register. R13 stack pointer.

CPSR – current program status register.
SPSR – saved program status register.

Coprocessor:

Up to 16 coprocessors can be connected to an ARMv7 system. Coprocessors are separate processing unit that tightly coupled to the ARM processor. Typical coprocessor contains:

- An instruction pipeline
- Instruction decode logic
- Handshake logic
- A register bank
- Special processing logic with its own data path

Debugging Feature:

Internal state of the ARM core can be examined using a JTAG interface to allow the insertion of instructions into core pipeline and avoid using external data bus.
ARM7TDMI core includes an internal functional unit known as the Embedded ICE logic. The embedded ICE logic is configured to monitor the ARM7TDMI core actively for specific instruction fetches and data accesses.

Applications:
Using the ARMv7 architecture, ARM can strengthen its position as a low-power/performance leader while conquering new markets to carry its cores up in high performance and down in the low-cost high-volume domain of the microcontroller ARM designs the technology that lies at the heart of advanced digital products, from wireless, networking and consumer entertainment solutions to imaging, automotive, security and storage devices. ARM's comprehensive product offering includes 16/32-bit RISC microprocessors, data engines, 3D processors, digital libraries, embedded memories, peripherals, software and development tools, as well as analog functions and high-speed connectivity products.

1.3 ARM operating modes
The processor mode determines which registers are active and the access rights to the cpsr register itself. Each processor mode is either privileged or non privileged; A privileged mode allows full read-write access to the cpsr. Conversely, a non privileged mode only allows read access to the control field in the cpsr but still allows read-write access to the condition flags.

There are seven processor/operating modes in total: six privileged modes

Abort mode

Fast interrupt request mode

Interrupt request mode

Supervisor mode

System mode

Undefined mode

User mode Nonprivileged mode

The processor enters

Abort mode : when there is a failed attempt to access memory.

Fast interrupt : Two interrupt levels

Interrupt : available on the ARM processor
Supervisor : The processor is in after reset and is generally the mode that an operating system kernel operates in.

System mode : A special version of user mode that allows full read-write access to the cpsr.

Undefined mode : When the processor encounters an instruction that is undefined or not supported by the implementation.

User mode : The mode is used for programs and applications.

1.4 System initialisation (Runtime Environment)

Most ARM applications begin by executing an assembly start up file. This file could be linked to the bottom of the on-chip memory (Flash (0x0) or SRAM (0x400 0000)) depending from where the application is targeted to run.

The following should be covered in this startup file:

1. Interrupt Vector table
2. Stack pointers
3. Branch to Main

After the above basic assembly initialization code is executed, a branch is done to C main().

The following steps could be carried out in C code:

1. Enable the Memory Accelerator Module (MAM) if the application is run from on-chip Flash. It provides accelerated execution at higher frequencies and also helps in reducing power consumption. The MAM is only available in devices with on-chip Flash.

2. Set the System clock and peripheral clock. The system clock can be boosted using the PLL to 60 MHz or 75 MHz depending upon the input frequency. The peripheral clock can be set using the VPB Divider register. Please refer to Application note AN10331 to get detailed information about the PLL.

3. Set the Memory Mapping Control register (MEMMAP at address 0xE01F C040) accordingly. The MEMMAP register gives the application the
flexibility of executing interrupts from different memory regions. For instance, if MEMAP is set to 0x2, the
Interrupt vectors would be mapped to 0x4000 0000 (bottom of on-chip SRAM).

4. Disable unused peripherals using the Power Control for Peripherals register (PCONP at address 0xE01F C0C4).

5. Configure GPIO’s using the respective IODIR, IOSET and IOCLR registers. On the LPC2000, there are certain pins that should not be held low on reset. For instance, by driving P0.14 low on reset would make the on-chip bootloader to take control of the part after reset.

6. Depending upon the peripherals being used set the port functions accordingly using the appropriate Pin Function Select register (PINSELx).

7. Initialize the peripherals which are still enabled in PCONP and enable interrupts for them if needed.

8. Configure the Vectored Interrupt Controller (VIC) to handle the different interrupt sources and classify them as IRQ and FIQ. It is recommended that only one interrupt source should be classified as an FIQ.

9. It is always safe to program the Default Vector Address Register (VICDefVectAddr) with a dummy ISR address wherein the VIC would be updated (by performing a write operation on the VIC Vector Address register (VICVectAddr) to avoid any spurious interrupts.
1.5 Arm Board description

Power:

DC 6.5V with power LED. On-board linear regulators generate +3.3V/500mA and +5V/500mA from power supply.

USB connector (as alternate power source).

Connectors:

- Extension headers for all microcontroller pins.
- RS232 connectors (2).
- VGA connector.
- PS/2 connector.
- JTAG connector.
- SD/MMC connector.
- USB B-type connector with Link-LED.

All peripheral configurable via jumpers.

Other Peripherals:

- 256Kb I2C based EEPROM
- Audio power amplifier.
- 2 line X 16 character LCD with back light control.
- Configurable for manual and automatic program download(ISP) via serial port.
- 8 controllable LEDs on SPI using 74HC595.
## Modules and Jumpers Relationship

<table>
<thead>
<tr>
<th>Jumper</th>
<th>Related</th>
<th>Usage</th>
</tr>
</thead>
<tbody>
<tr>
<td>J6</td>
<td>UART0 &amp; UART1</td>
<td>Connecting all pins enables both UART0 and UART1 and pins 5 and 7 enable UART0.</td>
</tr>
<tr>
<td>J8</td>
<td>VREF voltage</td>
<td>Connecting this will set the VTEF voltage to 0.3V</td>
</tr>
<tr>
<td>J9</td>
<td>Test LED’s</td>
<td>Connecting all pins enables test LED’s Pins 3 to 9 are connected to SPI0 lines of LPC2148.</td>
</tr>
<tr>
<td>J10</td>
<td>ADC</td>
<td>This will enable the ADC interface</td>
</tr>
<tr>
<td>J11</td>
<td>JTAG</td>
<td>This will enable the debug mode on the microcontroller.</td>
</tr>
<tr>
<td>J12</td>
<td>Keyboard(PS/2)</td>
<td>This will enable the PS/2 connector.</td>
</tr>
<tr>
<td>J13</td>
<td>Keyboard(PS/2)</td>
<td>This will provide 5V supply to PS/2</td>
</tr>
<tr>
<td>J18</td>
<td>LCD</td>
<td>Connecting all pins enabled LCD. Pins 1 to 7 are data lines, 9 to 13 are control lines and pin 15 is 5V power pin.</td>
</tr>
<tr>
<td>J19</td>
<td>LCD Backlight</td>
<td>If pins 1 are 2 are connected the LCD back light will always stay PN and if pins 2 and 3 are connected the back can be controlled by firmware.</td>
</tr>
<tr>
<td>J22</td>
<td>Power supply to board</td>
<td>Connecting this will provide 3.3V supply to board.</td>
</tr>
<tr>
<td>J25</td>
<td>I2C</td>
<td>By connecting all pins it enables I2C interface and its status is displayed on LCD.</td>
</tr>
<tr>
<td>J26</td>
<td>Bootloader select</td>
<td>If pins 1 and 2 are connected, manual bootloader mode is selected and If pins 2 and 3 are connected auto bootloader mode is selected. UART0 to be used for this purpose.</td>
</tr>
<tr>
<td>J27</td>
<td>RTC</td>
<td>Connect a battery to use RTC.</td>
</tr>
</tbody>
</table>
Section 2

KEIL µVision 4 Tool

2.1 How to work with keil?

2.1.1 How to create a new µProject?

Step 1: Give a double click on µvision 4 icon on the desk top, it will generate a window as shown below.
Step 2: To create new project go to project select new micro vision project.
Step 3: select a drive where you would like to create your project.
Step 4: Create a new folder and name it with your project name.

Step 5: Open that project folder and give a name of your project executable file and save it.
Step 6: After saving it will show some window there you select your microcontroller company i.e NXP from Phillips.

Step 7: Select your chip as LPC2148
Step 8: After selecting chip click on OK then it will display some window asking to add STARTUP file. Select YES.

Step 9: A target is created and startup file is added to your project target and is shown below.
Step 10: To write your project code select a new file from FILE menu bar.

Step 11: It will display some text editor, to save that file select SAVE option from FILE menu bar.
Step 12: By giving a file name with extension .C for c files and save it.

Step 13: Write the code of your project and save it.
Step 14: To add our c file to target give a right click on Source Group, choose “ADD files to Group” option.

Step 15: It will displays some window there select the file you have to add and click on ADD option.
Step 16: The file will be added to our target and it shows in the project window.

Step 17: Now give a right click on target in the project window and select “Options for Target”.
Step 18: It will show some window, in that go to output option and choose Create Hex file option by selecting that box.

Step 19: In the same window go to Linker option and choose Use Memory Layout from Target Dialog by selecting the box, and click OK.
Step 20: Now to Compile your project go to Project select Build Target option or press F7.

Step 21: In the build OUT PUT window you can see the errors and warnings if there in your code. And here Your project Hex file will be created.

Section 3
3.1 Flash Magic Tool

To program the Microcontroller, Flash Magic tool is used. Generally, the microcontroller is in one of the two modes. One is RUN mode and the other is PROGRAMMING mode. In RUN mode microcontroller executes the application present in the microcontroller flash memory. In PROGRAMMING mode, microcontroller programs its flash memory in synchronisation with Flash Magic.

To enter into the programming mode, Hold down SW2(isp) and SW3(reset), then release SW3 first and finally SW2. To enter into Run Mode, press the SW3(reset) after programming is over.

![Snapshot of the Flash Magic Tool.](image)

3.2 Downloading Hex file onto microcontroller
To program the flash memory, first keep the microcontroller in PROGRAMMING mode. Launch the Flash Magic Tool. Select the COM1, Baud rate as 19200, device as LPC2148; Oscillator Freq (MHz) as 12, in Communication block. Select the box erase all Flash + Code Rd Prot in Erase block. Select the box Verify after programming in Options Block. Select the hex file in Hex File block. Hold down SW2 (isp) and SW3 (reset), then release SW3 first and finally SW2. Then click Start Button in Start Block.
Section 4

Study of Real Time operating system

4.1 Issues in Real-time System Design

Designing Real time systems is a challenging task. Most of the challenge comes from the fact that Real time systems have to interact with real world entities. These interactions can get fairly complex. A typical Real time system might be interacting with thousands of such entities at the same time. For example, a telephone switching system routinely handles calls from tens of thousands of subscribers. The system has to connect each call differently. Also, the exact sequence of events in the call might vary a lot. Main issues in real time system design are

Real time Response
Recovering from Failures
Working with Distributed Architectures
Asynchronous Communication
Race Conditions and Timing

Real time Response

Real time systems have to respond to external interactions in a predetermined amount of time. Successful completion of an operation depends upon the correct and timely operation of the system. Design the hardware and the software in the system to meet the Real time requirements. For example, a telephone switching system must feed dial tone to thousands of subscribers within a recommended limit of one second. To meet these requirements, the off hook detection mechanism and the software message communication involved have to work within the limited time budget. The system has to meet these requirements for all the calls being set up at any given time. The designers have to focus very early on the Real time response requirements. During the architecture design phase, the hardware and software engineers work together to select the right system architecture that will meet the requirements. This involves deciding inter connectivity of the processors, link speeds, processor speeds, etc. some points to remember at this stage are:
Is the architecture suitable? If message communication involves too many nodes, it is likely that the system may not be able to meet the Real time requirement due to even mild congestion. Thus a simpler architecture has a better chance of meeting the Real time requirements.

Are the link speeds adequate? Generally, loading a link more than 40-50% is a bad idea. Higher link utilization causes the queues to build up on different nodes, thus causing variable amounts of delays in message communication.

Are the processing components powerful enough? A CPU with really high utilization will lead to unpredictable Real time behaviour. Also, it is possible that the high priority tasks in the system will starve the low priority tasks of any CPU time. This can cause the low priority tasks to misbehave. As with link, keep the peak CPU utilization below 50 %.

Is the Operating System suitable? Assign high priority to tasks that are involved in processing Real time critical events. Consider preemptive scheduling if Real time requirements are stringent. When choosing the operating system, the interrupt latency and scheduling variance should be verified.

Scheduling variance refers to the predictability in task scheduling times. For example, a telephone switching system is expected to feed dial tone in less than 500 ms. This would typically involve scheduling three to five tasks within the stipulated time. Most operating systems would easily meet these numbers as far as the mean dial tone delay is concerned. But general purpose operating systems would have much higher standard deviation in the dial tone numbers.

Interrupt Latency refers to the delay with which the operating system can handle interrupts and schedule tasks to respond to the interrupt. Again, real-time operating systems would have much lower interrupt latency.

**Recovering from Failures**

Real time systems must function reliably in event of failures. These failures can be internal as well as external. The following sections discuss the issues involved in handling these failures.
Internal Failures

Internal failures can be due to hardware and software failures in the system. The different types of failures you would typically expect are:

Software Failures in a Task: Unlike desktop applications, Real time applications do not have the luxury of popping a dialog box and exiting on detecting a failure. Design the tasks to safeguard against error conditions. This becomes even more important in a Real time system because sequence of events can result in a large number of scenarios. It may not be possible to test all the cases in the laboratory environment. Thus apply defensive checks to recover from error conditions. Also, some software error conditions might lead to a task hitting a processor exception. In such cases, it might sometimes be possible to just rollback the task to its previous saved state.

Processor Restart: Most Real time systems are made up of multiple nodes. It is not possible to bring down the complete system on failure of a single node thus design the software to handle independent failure of any of the nodes. This involves two activities:

Handling Processor Failure: When a processor fails, other processors have to be notified about the failure. These processors will then abort any interactions with the failed processor node. For example, if a control processor fails, the telephone switch clears all calls involving that processor.

Recovering Context for the Failed Processor: When the failed processor comes back up, it will have to recover all its lost context from other processors in the system. There is always a chance of inconsistencies between different processors in the system. In such cases, the system runs audits to resolve any inconsistencies. Taking our switch example, once the control processor comes up it will recover the status of subscriber ports from other processors. To avoid any inconsistencies, the system initiates audits to crosscheck data-structures on the different control processors.

Board Failure: Real time systems are expected to recover from hardware failures. The system should be able to detect and recover from board failures. When a board fails, the system notifies the operator about it. Also, the system should be able to switch
In a spare for the failed board. (If the board has a spare).

Link Failure: Most of the communication in Real-time systems takes place over links connecting the different processing nodes in the system. Again, the system isolates a link failure and reroutes messages so that link failure does not disturb the message communication.

External Failures
Real time systems have to perform in the real world. Thus they should recover from failures in the external environment. Different types of failures that can take place in the environment are:
Invalid Behaviour of External Entities: When a Realtime system interacts with external entities, it should be able to handle all possible failure conditions from these entities. A good example of this is the way a telephone switching systems handle calls from subscribers. In this case, the system is interacting with humans, so it should handle all kinds of failures, like:
Subscriber goes off hook but does not dial
Toddler playing with the phone!
Subscriber hangs up before completing dialing.
Inter Connectivity Failure: Many times a Realtime system is distributed across several locations. External links might connect these locations. Handling of these conditions is similar to handling of internal link failures. The major difference is that such failures might be for an extended duration and many times it might not be possible to reroute the messages.

**Working with Distributed Architectures**

Most Realtime systems involve processing on several different nodes. The system itself distributes the processing load among several processors. This introduces several challenges in design:

Maintaining Consistency: Maintaining data-structure consistency is a challenge when multiple processors are involved in feature execution. Consistency is generally maintained by running data-structure audits.
Initializing the System: Initializing a system with multiple processors is far more complicated than bringing up a single machine. In most systems the software release is resident on the OMC. The node that is directly connected to the OMC will initialize first. When this node finishes initialization, it will initiate software downloads for the child nodes directly connected to it. This process goes on in an hierarchical fashion till the complete system is initialized.

Inter-Processor Interfaces: One of the biggest headache in Real time systems is defining and maintaining message interfaces. Defining of interfaces is complicated by different byte ordering and padding rules in processors. Maintenance of interfaces is complicated by backward compatibility issues. For example if a cellular system changes the air interface protocol for a new breed of phones, it will still have to support interfaces with older phones.

Load Distribution: When multiple processors and links are involved in message interactions distributing the load evenly can be a daunting task. If the system has evenly balanced load, the capacity of the system can be increased by adding more processors. Such systems are said to scale linearly with increasing processing power. But often designers find themselves in a position where a single processor or link becomes a bottleneck. This leads to costly redesign of the features to improve system scalability.

Centralized Resource Allocation: Distributed systems may be running on multiple processors, but they have to allocate resources from a shared pool. Shared pool allocation is typically managed by a single processor allocating resources from the shared pool. If the system is not designed carefully, the shared resource allocate can become a bottleneck in achieving full system capacity.

Asynchronous Communication

Remote procedure calls (RPC) are used in computer systems to simplify software design. RPC allows a programmer to call procedures on a remote machine with the same semantics as local procedure calls. RPCs really simplify the design and development of conventional systems, but they are of very limited use in Realtime systems. The main reason is that most communication in the real world is asynchronous in nature, i.e. very few message interactions can be classified into the query response paradigm that works so well using RPCs.

Thus most Realtime systems support state machine based design where multiple messages can be received in a single state. The next state is determined by the contents of
the received message. State machines provide a very flexible mechanism to handle asynchronous message interactions. The flexibility comes with its own complexities.

**Race Conditions and Timing**

A race condition occurs when the state of a resource depends on timing factors that are not predictable. This is best explained with an example. Telephone exchanges have two way trunks which can be used by any of the two exchanges connected by the trunk. The problem is that both ends can allocate the trunk at more or less the same time, thus resulting in a race condition. Here the same trunk has been allocated for a incoming and an outgoing call. This race condition can be easily resolved by defining rules on who gets to keep the resource when such a clash occurs. The race condition can be avoided by requiring the two exchanges to work from different ends of the pool. Thus there will be no clashes under low load. Under high load race conditions will be hit which will be resolved by the pre-defined rules.

A more conservative design would partition the two way trunk pool into two one way pools. This would avoid the race condition but would fragment the resource pool. The main issue here is identifying race conditions. Most race conditions are not as simple as this one. Some of them are subtle and can only be identified by careful examination of the design.

**4.2 Introduction and characteristics of Real time operating systems**

The “kernel” of a real-time operating system (“RTOS”) provides an “abstraction layer” that hides from application software the hardware details of the processor (or set of processors) upon which the application software will run. This is shown in below figure.

![Figure: RTOS kernel provides abstraction between application software and Hardware](image)

In providing this “abstraction layer” the RTOS kernel supplies five main categories of basic services to application software, as seen in below Figure.
Each time the priority-based pre-emptive scheduler is alerted by an external world trigger (such as a switch closing) or a software trigger (such as a message arrival), it must go through the following 5 steps:

* Determine whether the currently running task should continue to run. If not …
* Determine which task should run next.
* Save the environment of the task that was stopped (so it can continue later).
* Set up the running environment of the task that will run next.
* Allow this task to run.

These 5 steps together are called “task switching”.

**Fixed Time Task switching**

The time it takes to do task switching is of interest when evaluating an operating system. A simple general-computing (non-pre-emptive) operating system might do task
switching only at timer tick times, which might for example be ten milliseconds apart. Then if the need for a task switch arises anywhere within a 10-millisecond timeframe, the actual task switch would occur only at the end of the current 10-millisecond period. Such a delay would be unacceptable in most real-time embedded systems.

These two types of timing behavior for task switching can be seen in below figure.

In this figure, for a general-computing (non-real-time) operating system, the task switching time generally rises as a software system includes more tasks that can be scheduled. However, the actual time for a task switch is not the time shown by the dashed red line. Instead, in any given task switch instance, it might be well above or well below the time shown by the dashed red line. The shaded regions surrounding the dashed red line simply show the likelihood of the actual task switch time being that far above or below the dashed red line.

On the other hand, the horizontal solid green line shows the task switching time characteristic of a real-time operating system. It is constant, independent of any load factor such as the number of tasks in a software system.
Please note that in some instances, such as the leftmost area of the graph, the task switching time might in special cases be quicker for a general-computing non-real-time operating system, than for a real-time operating system. This does not detract from the appropriateness of a real-time operating system for real-time embedded applications. For, in fact, the term “real-time” does not mean “as fast as possible”; but rather “real-time” demands consistent, repeatable, known timing performance. Although a non-real-time operating system might do some faster task switching for small numbers of tasks, it might equally well introduce a long time delay the next time it does the same task switch. The strength of a real-time operating system is in its known, repeatable timing performance, which is also typically faster than that of a non-deterministic task scheduler in situations of large numbers of tasks in a software system. Most often, the real-time operating system will exhibit task-switching times much faster than its non-real-time competitor when the number of tasks grows above 5 or 10.

**Inter Task Communication and Synchronization**

Most operating systems, including RTOSs, offer a variety of mechanisms for communication and synchronization between tasks. These mechanisms are necessary in a preemptive environment of many tasks, because without them the tasks might well communicate corrupted information or otherwise interfere with each other.

For instance, a task might be preempted when it is in the middle of updating a table of data. If a second task that preempts it reads from that table, it will read a combination of some areas of newly-updated data plus some areas of data that have not yet been updated. These updated and old data areas together may be incorrect in combination, or may not even make sense. An example is a data table containing temperature measurements that begins with the contents “10 C”. A task begins updating this table with the new value “99 F”, writing into the table character-by-character. If that task is preempted in the middle of the update, a second task that preempts it could possibly read a value like “90 C” or “99 C.” or “99 F”, depending on precisely when thepreemption took place. The partially updated values are clearly incorrect, and are caused by delicate timing coincidences that are very hard to debug or reproduce consistently.

An RTOS’s mechanisms for communication and synchronization between tasks are provided to avoid these kinds of errors. Most RTOSs provide several mechanisms, with each mechanism
optimized for reliably passing a different kind of information from task to task.

Probably the most popular kind of communication between tasks in embedded systems is the passing of data from one task to another. Most RTOSs offer a message passing mechanism for doing this, as seen in below Figure. Each message can contain an array or buffer of data.

![Intertask Message Communication](image)

**Figure:** Intertask Message Communication

If messages can be sent more quickly than they can be handled, the RTOS will provide message queues for holding the messages until they can be processed. Another kind of communication between tasks in embedded systems is the passing of what might be called “synchronization information” from one task to another. “Synchronization information” is like a command, where some commands could be positive, and some negative. For example, a negative command to a task would be something like “Please don’t print right now, because my task is using the printer”. Or more generally, “I want to lock the … for my own use only”. A positive command would be something like “I’ve detected a cardiac emergency, and I want you to help me handle it”. Or more generally, “Please join me in handling ….”.

Most RTOSs offer a semaphore or mutex mechanism for handling negative synchronization (sometimes called “mutual exclusion”). These mechanisms allow tasks to lock certain embedded system resources for their use only, and subsequently to unlock the resource when they’re done.

For positive synchronization, different RTOSs offer different mechanisms. Some RTOSs offer event-flags, while others offer signals. And yet others rely on message passing for positive synchronization as well as data passing duties.

**Determinisim and High Speed message passing**
Intertask message communication is another area where different operating systems show different timing characteristics. Most operating systems actually copy messages twice as they transfer them from task to task via a message queue. The first copying is from the message-sender task to an operating system-owned “secret” area of RAM memory (implementing the "message queue"); and the second copying is from the operating system’s "secret" RAM area to the message-receiver task. Clearly this is non-deterministic in its timing, as these copying activities take longer as message length increases.

![Figure: Message Transfer via Message Queue](image)

An approach that avoids this non-determinism and also accelerates performance is to have the operating system copy a pointer to the message and deliver that pointer to the message-receiver task without moving the message contents at all. In order to avoid access collisions, the operating system then needs to go back to the message-sender task and obliterate its copy of the pointer to the message. For large messages, this eliminates the need for lengthy copying and eliminates non-determinism.

**Dynamic Memory Allocation**

Determinism of service times is also an issue in the area of dynamic allocation of RAM memory. Many general-computing non-real-time operating systems offer memory allocation services from what is termed a “Heap”. The famous “malloc” and “free” services known to C-language programmer’s work from a heap. Tasks can temporarily borrow some memory from the operating system’s heap by calling “malloc”, and specifying the size of memory buffer needed. When this task (or another task) is finished with this memory buffer it can return the buffer to the operating system by calling “free”. The operating system will then return the buffer to the heap, where its memory might be used again, perhaps as part of a larger buffer. Or perhaps it may in the future be broken into several smaller buffers.

Heaps suffer from a phenomenon called “External Memory Fragmentation” that may
cause the heap services to degrade. This fragmentation is caused by the fact that when a buffer is returned to the heap, it may in the future be broken into smaller buffers when “malloc” requests for smaller buffer sizes occur. After a heap undergoes many cycles of “malloc”’s and “free”’s, small slivers of memory may appear between memory buffers that are being used by tasks. These slivers are so small that they are useless to tasks. But they are trapped between buffers that are being used by tasks, so they can’t be coagulated (“glued”) together into bigger, useful buffer sizes. Over time, a heap will have more and more of these slivers. This will eventually result in situations where tasks will ask for memory buffers (“malloc”) of a certain size, and they will be refused by the operating system --- even though the operating system has enough available memory in its heap.

The problem: That memory is scattered in small slivers distributed in various separate parts of the heap. In operating system terminology, the slivers are called “fragments”, and this problem is called “external memory fragmentation”.

This fragmentation problem can be solved by so-called “garbage collection” (defragmentation) software. Unfortunately, “garbage collection” algorithms are often wildly non-deterministic – injecting randomly-appearing random-duration delays into heap services. These are often seen in the memory allocation services of general-computing non-real-time operating systems.

This puts the embedded system developer who wants to use a general-computing non-real-time operating system into a quandry: Should the embedded system be allowed to suffer occasional randomly-appearing random-duration delays if / when "garbage collection" kicks in? ... Or, alternatively, should the embedded system be allowed to fragment its memory until application software "malloc" requests to the heap are refused even though a sufficient total amount of free memory is still available? Neither alternative is acceptable for embedded systems that need to provide service continually for long periods of time.

Real-time operating systems, on the other hand, solve this quandry by altogether avoiding both memory fragmentation and “garbage collection”, and their consequences. RTOSs offer non-fragmenting memory allocation techniques instead of heaps. They do this by limiting the variety of memory chunk sizes they make available to application software. While this approach is less flexible than the approach
taken by memory heaps, they do avoid external memory fragmentation and avoid the need for defragmentation. For example, the “Pools” memory allocation mechanism allows application software to allocate chunks of memory of perhaps 4 or 8 different buffer sizes per pool. Pools totally avoid external memory fragmentation, by not permitting a buffer that is returned to the pool to be broken into smaller buffers in the future. Instead, when a buffer is returned the pool, it is put onto a “free buffer list” of buffers of its own size that are available for future re-use at their original buffer size. This is shown in below Figure.

**Figure: A Memory Pool's Free Buffer Lists**

Memory is allocated and de-allocated from a pool with deterministic, often constant, timing.

### 4.3 UCOS-II

uC/OS-II is a priority-based preemptive multi-tasking real-time operating system, including a real-time kernel, task management, time management, inter-task communication synchronization (semaphores, mailboxes, message queues), and memory management functions.

It enables all tasks to work independently, and mutual non interference, it is easy to achieve timely and accurate implementation of real-time applications to the design and expansion easier, the application of the design process greatly
reduced. And its kernel source code open, portability strong, and the programming staff provided a good one software platform.

**uC/OS-II core structure:**

Multi-tasking system, the kernel is responsible for all management tasks, or tasks assigned for each CPU time, and be responsible for communications between tasks. Core provision of basic services task switching. **uC/OS-II** can manage up to 64 tasks. Since it occupied and the author retains the eight tasks, so users for the applications of up to 56 tasks. Given the priority of each task must not be the same.

This means that **μC/OS-II** does not support round-robin scheduling. **uC/OS-II** for each of its mandate independent stack space, the task can rapidly switch. **uC/OS-II** to approximate the time always place the highest priority task in functioning condition, in order to ensure this, it calls system API functions, interrupt the end, time is always interrupted at the end of the implementation of scheduling, **μC/OS-II** in advance of good data through simplified computation, through careful design ready table structure makes delay predictable.

**μC/OS-II features:**

**Source Code:**

Source code is open for educational purposes. Completely written in c language.

**Portable:**

Most of uC/OS-II is written in highly portable ANSI C, with target microprocessor specific code written in assembly language. Assembly language is kept to a minimum to make μC/OS -II easy to port to other processors. Like μC/OS, μC/OS-II can be ported to a large number of microprocessors as long as the microprocessor provides a stack pointer and the CPU registers can be pushed onto and popped from the stack. Also, the C compiler should either provide in-line assembly or language extensions that allow you to enable and disable interrupts from C. μC/OS-II can run on most 8-bit, 16-bit, 32-bit or even 64-bit microprocessors or micro-controllers and, DSPs.

All the ports that currently exist for μC/OS can be easily converted to μC/OS-II in about an hour. Also, because μC/OS-II is upward compatible with μC/OS, your μC/OS applications should run on μC/OS-II with few or no changes.
Check for the availability of ports on the μC/OS-II Web site at ‘www.uCOS-II.com’.

ROMable:

μC/OS-II was designed for embedded applications. This means that if you have the proper tool chain (i.e. C compiler, assembler and linker/locator), you can embed μC/OS-II as part of a product.

Scalable:

μC/OS-II is designed that you can use only the services that you need in your application. This means that a product can have just a few of μC/OS-II’s services while another product can have the full set of features. This allows you to reduce the amount of memory (both RAM and ROM) needed by μC/OS-II on a product per product basis. Scalability is accomplished with the use of conditional compilation. You simply specify (through #define constants) which features you need for your application/product. I did everything I could to reduce both the code and data space required by μC/OS-II.

Preemptive:

μC/OS-II is a fully-preemptive real-time kernel. This means that μC/OS-II always runs the highest priority task that is ready. Most commercial kernels are preemptive and μC/OS-II is comparable in performance with many of them.

Multi-tasking:

μC/OS-II can manage up to 64 tasks, however, the current version of the software reserves eight (8) of these tasks for system use. This leaves your application with up to 56 tasks. Each task has a unique priority assigned to it which means that μC/OS-II cannot do round robin scheduling. There are thus 64 priority levels.

Deterministic:

Execution time of all μC/OS-II functions and services are deterministic. This means that you can always know how much time μC/OS-II will take to execute a function or a service. Furthermore, except for one service, execution time of all μC/OS-II services do not depend on the number of tasks running in your application.

Task stacks:
Each task requires its own stack, however, μC/OS-II allows each task to have a different stack size. This allows you to reduce the amount of RAM needed in your application. With μC/OS-II’s stack checking feature, you can determine exactly how much stack space each task actually requires.

**Services:**

μC/OS-II provides a number of system services such as mailboxes, queues, semaphores, fixed-sized memory partitions, time related functions, etc.

**Interrupt Management:**

Interrupts can suspend the execution of a task and, if a higher priority task is awakened as a result of the interrupt, the highest priority task will run as soon as all nested interrupts complete. Interrupts can be nested up to 255 levels deep.

**Robust and reliable:**

μC/OS-II is based on μC/OS which has been used in hundreds of commercial applications since 1992. μC/OS-II uses the same core and most of the same functions as μC/OS yet offers more features.

**μC/OS-II porting:**

Porting is to make μC/OS-II to run on LPC2148. This porting involves development of software in c and assembly language. In fact, μC/OS-II completely written in c language for to achieve portability across different architectures, so development of porting files is relatively easier.

**μC/OS-II source files:**

As shown in Fig. A, the architecture of software and hardware of μC/OS-II Real time operating system can be classified into application software, processor independent code, application specific μC/OS-II configuration and μC/OS-II porting blocks. μC/OS-II porting covers development of os_cpu.h, os_cpu_a.s and os_cpu_c.c.
**uC/OS-II Hardware/Software architecture**

**Application software (your code)**

**uC/OS-II**
- Os_core.c
- Os_flag.c
- Os_mbox.c
- Os_mem.c
- Os_mbox.c
- Os_q.c
- Os_sem.c
- Os_task.c
- Os_time.c
- Ucos_i.c
- Ucos_i.h

**uC/OS-II Configuration**
(Application specific)
- Os_cfg.h
- Lincludes.h

**uC/OS-II port**
- os_cpu.h
- os_cpu_a.s
- os_cpu_c.c

**CPU**

**Timer**

**Port header:** os_cpu.h

**Critical section protection:**

The ARM-GCC port uses OS critical section protection method #3; it defines a function for saving the processor status while disabling FIQ and IRQ interrupts, and another to restore the processor status. The function declarations and critical section macros are located in os_cpu.h;

```c
OS_CPU_SR OS_CPU_SR_Save(void);
void OS_CPU_SR_Restore(OS_CPU_SR cpu_sr);
#define OS_ENTER_CRITICAL() { cpu_sr = OS_CPU_SR_Save();}
#define OS_EXIT_CRITICAL() {OS_CPU_SR_Restore(cpu_sr);}
```

And the function implementations are in os_cpu_a.s.

**Task-level context switch:**

The task-level context switch macro, OS_TASK_SW(), is defined as a call to OSCtxSw() (see the OS_CPU_a.s section).

**Port C-functions:** os_cpu.c.c

The only C function the port needed to define was OSTaskStkInit() to initialize the stack.

**Port assembler-functions:** os_cpu.a.s
A port requires the implementation of four assembler routines; OSStartHighRdy (start multi-tasking), OSCtxSw (task-level context switch), OSIntCtxSw (interrupt-level context switch), and OSTickISR (time-tick ISR).

Start multi-tasking

OSStartHighRdy() is called at the end of OSStart() (in μCOS-II source file OS_CORE.C), and is the exit point from main()’s context into the RTOS. OSStartHighRdy() implements the context restore of the registers. The function starts by ensuring that the processor is in user/system mode with FIQ and IRQ interrupts disabled (although having the interrupts disabled is not critical, as there should be no interrupt generating sources setup at this point). The OSTaskSwHook() function is then called, and the OSRunning flag set to true. The user/system mode stack pointer is then changed to that of the highest-priority (and only) task. The task CPSR is then copied into the CPSR register (which happens to enable FIQ/IRQ interrupts), and the task register context is restored.

Task-level context switch

OS_Sched() (OS_CORE.C) calls OS_TASK_SW() to implement a task-level context switch from inside a critical section (so both FIQ and IRQ are disabled when this function is called). The macro OS_TASK_SW() is a call to OSCtxSw() in this port, so on entry to the context switch function, the link register will contain the task return address. The job of OSCtxSw() is to save the current task context, switch over to the higher-priority task, and then restore context. The code saves the current tasks registers onto its stack; the contents of link register are saved to both the link register and the program counter locations on the stack. The task stack-pointer is then saved to its task control block, the OSTaskSwHook() function is called, the higher-priority task stack is loaded, and the context of the higher-priority task is restored.

Interrupt-level context switch:

The FIQ and IRQ ISRs start by saving the processor context, incrementing the OSIntNesting counter (and saving the current value of the stack pointer if required), and the IRQ ISR then reenables IRQ interrupts. The ISR then calls handler code (written in C). When the handler returns, the ISR calls OSIntExit(), and then restores the processor state. OSIntExit() (OS_CORE.C) checks to see if interrupt nesting is over, and then if a higher-priority task is ready. If interrupts are still nested, or the same task has the highest priority, then OSIntExit() returns, and
the ISR runs to completion (i.e., performs the context restore of the task or interrupt it interrupted). If however, interrupt nesting is over, and a higher-priority task has been made ready, then a switch to the new task is required; that is the job of OSIntCtxSw(). OSIntExit() calls OSIntCtxSw() inside a critical section, so interrupts are disabled when this function is called. The interrupt-level context switch code is similar to the task-level context switch code, except that the ISR has already done the work of saving the processor context to the task stack. OSIntCtxSw() starts by calling the OSTaskSwHook(), the higher-priority task stack is then loaded, and the context of the higher-priority task is restored.

Interrupt service routines (ISRs)

The FIQ and IRQ ISRs are setup to call C-coded handlers. It is up to the board-support package to decide where to call the OS function OSTimeTick(). For example, timer 0 can be setup to generate clock ticks and the VIC can be setup to generate an FIQ (for testing), or as an IRQ (a vectored interrupt would be recommended). When an FIQ interrupt occurs, the ISR performs a partial context save (since the stack pointer is currently that of the FIQ, not the system mode task stack), and the processor is placed into system mode with interrupts disabled. The task context is then saved to the task stack. The interrupt nesting counter is then incremented, and if this is the first layer of nesting, the current value of the stack-pointer is saved to the task control block. The processor is then changed back to FIQ mode with interrupts disabled, and the FIQ handler function is called. After the handler returns, the processor is moved back to system mode, OSIntExit() is called, and the task context is restored. FIQ interrupts are not nested. When an IRQ interrupt occurs, the ISR performs a partial context save (since the stack pointer is currently that of the IRQ, not the system mode task stack), and the processor is placed into system mode with interrupts disabled. The task context is then saved to the task stack. The interrupt nesting counter is then incremented, and if this is the first layer of nesting, the current value of the stack-pointer is saved to the task control block. The processor is then changed back to system mode, OSIntExit() is called, and the task context is restored. FIQ interrupts are not nested. When an IRQ interrupt occurs, the ISR performs a partial context save (since the stack pointer is currently that of the IRQ, not the system mode task stack), and the processor is placed into system mode with interrupts disabled. The task context is then saved to the task stack. The interrupt nesting counter is then incremented, and if this is the first layer of nesting, the current value of the stack-pointer is saved to the task control block. The processor is then changed back to system mode. The VIC vector address register is then read. The VIC vector address register returns the address of the IRQ handler, and triggers the VIC priority logic to only allow IRQ interrupts of higher-priority to interrupt the processor core. FIQ and IRQ interrupts are then enabled (with the processor left in system mode), and the handler function read from the VIC is
called. After the handler returns, FIQ and IRQ interrupts are disabled, and the VIC is acknowledged by writing to the VIC vector address register. OSIntExit() is called, and the task context is restored.

4.4. Comparison of various real time operating system

The selected quantitative parameters to be analyzed in each system are:

1. Latency: Latency is analyzed externally taking the RTOS under test in conjunction with the hardware as a black box. The latency consists of the time difference between the moment that an interrupt is generated and the moment that the associated interrupt handler generates an external response. The latency was measured in a scenario with low CPU use and with the CPU overloaded.

2. Jitter: Jitter is indirect information obtained from several latency measures, Consisting of a random variation between each latency value. In a RTOS, the jitter impact could be notorious, as it is analyzed by Proctor when trying to control step motors. For example, the pulses duration controls the motor rotation, but the jitter induce the torque to vary, causing step losses in the motor. To compute jitter, the time difference between two consecutive interrupt latency measures is calculated. Finally, the greatest encountered difference is selected as the worst jitter of this system;

3. Worst Case Response Time: Worst Case Response Time is obtained using the method proposed by ISA that was discussed above analyzing the maximum interrupts frequency that is handled by the RTOS with reliability. The worst case response time is the inverse of the maximum frequency obtained. The test was made in a low CPU usage scenario and in an overloaded CPU scenario.

| Table 1. Worst times measured during the experiments. A: Response Time (1/maximum sustained frequency), B: Latency, C: Latency Jitter |
|-----------------|-----------------|-----------------|-----------------|-----------------|-----------------|-----------------|-----------------|
|                 | Win XP          | Win CE          | Neutrino        | µC/OS-II        | Linux           | RTA1            | VxWorks         |
| A                | 200µs           | 20µs            | 20µs            | 1,92µs          | 13,89µs         | 5µs             | 3,85µs          |
| B                | 848µs           | 99µs            | 35,2µs          | 3,2µs           | 98µs            | 11,4µs          | 13,4µs          |
| C                | 700µs           | 88,8µs          | 32µs            | 2,32µs          | 77,6µs          | 7,01µs          | 10,4µs          |
Section 5
Serial Communication Drivers for ARM Processors

5.1 Study of serial communication architecture on LPC2148 ARM architecture

Introduction: LPC2148 microcontroller has two UARTs. The microcontroller has to communicate to external world through the UARTs only. There is an importance for developing serial port driver.

- 16 byte Receive and Transmit FIFOs.
- Register locations conform to ‘550 industry standard.
- Receiver FIFO trigger points at 1, 4, 8, and 14 bytes.
- Built-in baud rate generator.
- LPC2148 contains mechanism that enables software flow control implementation.

The below fig. shows the register mapping. For complete description of individual register refer to user guide.
5.2 Implementing serial communication drivers for ARM processors

The major steps in carrying out the experiment:

i) write a c program.

ii) Build the application.

iii) Download the hex file on to the ARM board and check the result.

Writing the program:

Create a new folder as serial_driver. In the serial_driver create a folder called src.

To write source code, you have to open keil software and can write program in new file. While saving the file save it with .c extension. The program code can be seen in figure below.

Add UART0.c files in the project window.

After writing the code we to add all necessary files and save the code, after saving we have to compile and build the code. For building we can press F7 button or can click on either of this icons.
After that we can see the results or can find out any errors or warnings in the Build output window as shown in figure below.

We can see that there are 0 errors and few warning, we can ignore warning as we can get correct output with warnings also.

When there are no errors then we can download this hex file in ARM using FlashMagic.

To download the serial.hex file on to the microcontroller

To download the hex file into the microcontroller board we use a software called Flash magic tool.
Step 1 - Communications

- Set COM Port: COM1
- Baud Rate: 9600
- Device: LPC2148
- Oscillator Freq (MHz): 12

Step 2 - Erase

Select the box Erase all Flash + Code Rd Prot

Step 3 - Hex File

Click on browse to load the serial.hex file from the folder serial_driver.

Step 4 - Options

Select the box Verify after programming.

Step 5 - Start

Power up the microcontroller board using USB cable, make serial cable connection between PC and microcontroller’s UART0 db9 connector.

To make the board enter programming mode
Hold down SW2 (isp) and SW3 (reset), then release SW3 first and finally SW2.

**Step 5-Start**

Click the Start button

After finishing the programming, To check the result, open **hyper terminal** in PC using with same Flash Magic tool

Set baud rate: 9600  
Com port: com1  
Click on ok button.

Now, to put the microcontroller in RUN mode, press RESET

We can see “HELLO WORLD” displayed on hyper terminal.
Source code for Serial Communication Driver

**MAIN CODE**

```c
#include <LPC214X.H>
void main()
{
    PINSEL0=0x05;
    InitUart0();
    Tx_string("Hello world\n");
    while(1);
}
```

**UART 0 CODE**

```c
//include "LPC2148.h"
#include <LPC214X.H>

#define DESIRED_BAUDRATE 19200

#define CRYSTAL_FREQUENCY_IN_HZ 12000000
#define MAX_PCLK (CRYSTAL_FREQUENCY_IN_HZ*5)
#define PCLK (MAX_PCLK/4)/ (*1=60MHz,2=30MHz,4=15MHz*)
#define DIVISOR (PCLK/(16*DESIRED_BAUDRATE))

void InitUart0(void)
{

```
/* U0LCR: UART0 Line Control Register
0x83: enable Divisor Latch access, set 8-bit word length,
1 stop bit, no parity, disable break transmission  */
U0LCR=0x83;

/* VPBDIV: VPB bus clock divider
0x01: PCLK = processor clock */
VPBDIV=0x00;

/* U0DLL: UART0 Divisor Latch (LSB) */
U0DLL=DIVISOR&0xFF;

/* U0DLM: UART0 Divisor Latch (MSB) */
U0DLM=DIVISOR>>8;

/* U0LCR: UART0 Line Control Register
0x03: same as above, but disable Divisor Latch access */
U0LCR=0x03;

/* U0FCR: UART0 FIFO Control Register
0x05: Clear Tx FIFO and enable Rx and Tx FIFOs */
U0FCR=0x05;

}

void Tx_char(char ch)
{
    if (ch=='\n')
    {
        //wait until Transmit Holding Register is empty
        while (!(U0LSR&0x20)) {} 

        //then store to Transmit Holding Register
        U0THR='\r';
    }
    //wait until Transmit Holding Register is empty
    while (!(U0LSR&0x20)) {} 

    //then store to Transmit Holding Register
    U0THR=ch;
}

unsigned char Rx_char(void)
{
    char ch;

    //wait until there's a character to be read
    while (!(U0LSR&0x01)) {} 

    //then read from the Receiver Buffer Register
ch = U0RBR;
return ch;
}

int Tx_string(char *s)
{
    int i = 0;
    while(s[i] != '\0')
    {
        Tx_char(s[i]);
        i++;
    }
    return(i);
}
5.3 Development of echo program for Serial Communication

AIM:

To develop serial port driver for LPC2148 microcontroller

Experiment procedure:

Create a new folder as serial_echo_driver. In the serial_echo_driver create a folder called src.

To write source code, you have to open keil software and can write program in new file. While saving the file save it with .c extension. The program code can be seen in figure below.

Add UART0.c files in the project window.

After writing the code we to add all necessary files and save the code, after saving we have to compile and built the code. For building we can press F7 button or can click on either of this icons.
After that we can see the results or can find out any errors or warning in the Build output window as shown in figure below.

We can see that there are 0 errors and few warning, we can ignore warning as we can get correct output with warnings also.

When there are no errors then we can download this hex file in ARM using FlashMagic.

To download the `serial.hex` file on to the microcontroller

FOLLOW THE SAME STEPS GIVEN IN EXPERIMENT 3

RESULT:

To check the result open **hyper terminal** in PC using Flash Magic tool
Set baud rate: 9600
Com port: com1
Click on ok button.

Now, to put the microcontroller in RUN mode, press RESET.

We can see “HELLO WORLD” displayed on hyper terminal.

If we enter characters in the input window, the same characters are echoed back from the microcontroller and displayed on output window.
Source code of Experiment 4 (Echo program for Serial Communication)

MAIN CODE

#include <LPC214X.H>
void main()
{
    PINSEL0=0x05;
    InitUart0();

    Tx_string("This program will the echo the characters received\n");
    while(1)
        Tx_char(Rx_char());
}

UART 0 CODE

#include "LPC214X.H"
#define DESIRED_BAUDRATE 19200

#define CRYSTAL_FREQUENCY_IN_HZ 12000000
#define MAX_PCLK (CRYSTAL_FREQUENCY_IN_HZ*5)
#define PCLK (MAX_PCLK/4) /* 1=60MHz, 2=30MHz, 4=15MHz*/
#define DIVISOR (PCLK/(16*DESIRED_BAUDRATE))

void InitUart0(void)
{
    /* U0LCR: UART0 Line Control Register
       0x83: enable Divisor Latch access, set 8-bit word length,
       1 stop bit, no parity, disable break transmission */
    U0LCR=0x83;

    /* VPBDIV: VPB bus clock divider
       0x01: PCLK = processor clock */
    VPBDIV=0x00;

    /* U0DLL: UART0 Divisor Latch (LSB)
       U0DLL=DIVISOR&0xFF; */

    /* U0DLM: UART0 Divisor Latch (MSB)
       U0DLM=DIVISOR>>8; */

    /* U0LCR: UART0 Line Control Register
       0x03: same as above, but disable Divisor Latch access
       U0LCR=0x03; */

    /* U0FCR: UART0 FIFO Control Register
       0x05: Clear Tx FIFO and enable Rx and Tx FIFOs */
U0FCR=0x05;
}

void Tx_char(char ch)
{
    if (ch=='\n')
    {
        // wait until Transmit Holding Register is empty
        while (!(U0LSR&0x20)) {}

        // then store to Transmit Holding Register
        U0THR='\r';
    }
    // wait until Transmit Holding Register is empty
    while (!(U0LSR&0x20)) {}

    // then store to Transmit Holding Register
    U0THR=ch;
}

unsigned char Rx_char(void)
{
    char ch;

    // wait until there's a character to be read
    while (!(U0LSR&0x01)) {}

    // then read from the Receiver Buffer Register
    ch=U0RBR;
    return ch;
}

int Tx_string(char *s)
{
    int i=0;
    while(s[i]!='\0')
    {
        Tx_char(s[i]);
        i++;
    }
    return(i);
}
6.1 Studying the basic DSP platform requirements

The following block diagram shows the signal processing setup using microprocessors or microcontrollers. The ADC converts the input signal in digital form, which will be read by the processor. The digital signal processing application could be filtering, modulation, demodulation or any typical algorithm.

Any DSP algorithms finally need to be implemented by using arithmetic operations. Hence any Digital Signal Processing (DSP) platform requires arithmetic blocks. In addition higher memory space is required for handling larger programs. The following are the requirements for implementing DSP applications.

1. Enough Processing speed
2. Required ALU size (like 32 bit, 16 bit etc)
3. Sufficient code and data memories
4. Special functional units such as MAC (multiply and accumulate)

The ARM core is not a dedicated DSP. There is no single instruction that issues a multiply accumulate and data fetch in parallel. However by reusing loaded data we can achieve a respectable DSP performance.
6.2 Implementing the FIR filter

**AIM:** Implementing the FIR filter on ARM

**Introduction:** LPC 2148 has on chip 10 bit Digital to analog converter. The direct form of FIR filter is standard linear convolution, which described the output as convolution of input and impulse response of the filter.

\[ y[n] = x[n] * c[n] = \sum_{k} x[k]c[n-k] = \sum_{k} c[k]x[n-k]. \]

where \(c[n]\) values represent filter coefficients, and \(x[n]\) represents the input samples. The below figure shows the direct form FIR structure

![Direct form 6-tap FIR filter](image)

**Low pass filter specifications**

- Sampling frequency \(f_s\) = 50KHz
- Pass band frequency \(f_{pass}\) = 5khz
- Stop band frequency \(f_{stop}\) = 7khz
- The FDA tool designs : 9 tap FIR filter (equiripple)

Filter Design & Analysis Tool
FDATool in MATLAB provides access to most FIR and IIR filter design methods in the toolbox. Additionally, you can:

- Design filters by graphically placing poles and zeros in the z-plane
- Analyze filters by examining magnitude, phase, impulse, and step responses, group delay, and pole-zero plots.
- Generate C header files.
- Import designed filters and filter coefficients stored in the MATLAB workspace and export filter coefficients
- Access additional filter design methods and quantization features in the Filter Design Toolbox (available separately).

FDA TOOL SPECIFICATIONS:
To open the FDA window, Type fdatool in Command window of MATLAB.
The specifications of filter as shown in the figure below. The various specifications to be mentioned are:

1. Response type: To specify whether the desired filter is low pass, high pass, band pass or band stop filter. Select the Low pass filter in this case.
2. Design method: The filter to be designed is either FIR or IIR filter. Select FIR filter here.
3. Order of the filter: Specify the Order of the filter is 9 here.
4. Frequency specification: Specify the sampling, pass band stop band and frequencies 50kHz, 5 kHz and 7kHz respectively.
After giving the specifications for the required filter click on "design filter" which generates the coefficients for the fir filter. To export these coefficients, Go to File->Export then following window will open.
Click on “Export” and given a variable name as Num.
To convert the filter coefficients into required format, multiply Num with 128
Since Fractional part of coefficients is represented by 7 bit, 2 bits used to represents the
integer part and 1 bit used as signed bit.
Type Num in Command window in MATLAB, and then multiply with 128 by typing
Num*128.Consider the integer parts as co efficient of given filter.

The major steps in carrying out the experiment:

i) Write a c program.
ii) Calculate the filter co-efficient using FDA tool available in
MATLAB.
iii) Build the application.
iv) Download the hex file on to the ARM board and check the result.

Writing the program:

Create a new folder as firfilter_exp. In the firfilter_exp create a folder called src.

To write source code, you have to open keil software and can write program in new file.
While saving the file save it with .c extension. The program code can be seen in figure
below.
After writing the code we to add all necessary files and save the code, after saving we have to compile and built the code. For building we can press F7 button or can click on either of this icons.
After that we can see the results or can find out any errors or warning in the Build output window as shown in figure below

We can see that there are 0 errors and few warning, we can ignore warning as we can get correct output with warnings also.

When there are no errors then we can download this hex file in ARM using FlashMagic.

To download the filter.hex file on to the microcontroller

**FOLLOW THE SAME STEPS GIVEN IN EXPERIMENT 3**

**RESULT:**

Applying the sin wave with 1kHz (from function generator) as input to ADC0.3 pin of LPC2148. To see the output, connect CRO probe positive and Ground pins to the DAC output of LPC2148, ground pin of Board respectively.

Now, to put the microcontroller in RUN mode, press RESET. You can see the same sin wave as output, change the frequency observe the output.
Source code (Implementation of FIR filter Design)

```
// main.c

#include <LPC214X.H>

void small_delay()
{
    unsigned int i=0;
    //int j=0;

    //for(j=0;j<4;j++)
    for(i=0;i<1000;i++);
}

int main ()
{
    unsigned int temp=0,value=0;
    signed int x;
    signed long x_1,x_2,x_3,x_4,x_5,x_6,x_7,x_8,x_9,x_10,y=0,temp1;

    //PINSEL0 = 0x00000005 ; // enabling the UART0 pins
    PINSEL1 = 0x10080000;
    VPBDIV=0x00;
    AD0CR=0x00210408;
    //AD0CR |= 0x01000000; //start A/D conversion

    while(1)
    {
```
value=0;
//small_delay();

while(!(value & 0x80000000)) // till the done =0
value=AD0DR3;

// if((value & 0x07000000)==0x03000000)
// {
    // temp=(value& 0x0000FFFF) >> 6;
    temp=(value>>6)&0x03FF;
    x=temp;
    x_10=x_9;
    x_9=x_8;
    x_8=x_7;
    x_7=x_6;
    x_6=x_5;
    x_5=x_4;
    x_4=x_3;
    x_3=x_2;
    x_2=x_1;
    x_1=x;
    // y=(((x_1)*-10)+((x_2)*13)+((x_3)*19)+((x_4)*26)+((x_5)*31)+((x_6)*31)+((x_7)*26)+((x_8)*19)
    +((x_9)*13)+((x_10)*-10));
    y=(((x_1)*-21)+((x_2)*27)+((x_3)*39)+((x_4)*53)+((x_5)*62)+((x_6)*62)+((x_7)*53)+((x_8)*39)
    +((x_9)*27)+((x_10)*-21));
    y=y/256;
    temp1=y;
    DACR = temp1<<6;
// }
//
}
6.3 Implementation of CORDIC DSP algorithm

 AIM:

To implementing the CORDIC algorithm on LPC2148 microcontroller

Introduction: COordinate Rotation DIgital Computer (CORDIC) calculates the trigonometric functions of sine, cosine, magnitude and phase (arctangent) to any desired precision. CORDIC revolves around the idea of "rotating" the phase of a complex number, by multiplying it by a succession of constant values. However, the multiplies can all be powers of 2, so in binary arithmetic they can be done using just shifts and adds; no actual multiplier is needed.

Given a complex value: $C = Ic + jQc$
we will create a rotated value: $C' = Ic' + jQc'$
by multiplying by a rotation value: $R = Ir + jQr$

when you multiply a pair of complex numbers, their phases (angles) add and their magnitudes multiply. Similarly, when you multiply one complex number by the conjugate of the other, the phase of the conjugated one is subtracted (though the magnitudes still multiply).

Therefore:
To add $R$'s phase to $C$:
$C' = C \cdot R$
$Ic' = Ic \cdot Ir - Qc \cdot Qr$
$Qc' = Qc \cdot Ir + Ic \cdot Qr$

To subtract $R$'s phase from $C$:
$C' = C \cdot R^*$
$Ic' = Ic \cdot Ir + Qc \cdot Qr$
$Qc' = Qc \cdot Ir - Ic \cdot Qr$

To rotate by +90 degrees, multiply by $R = 0 + j1$. Similarly, to rotate by -90 degrees, multiply by $R = 0 - j1$. If you go through the Algebra above, the net effect is:

To add 90 degrees,
multiply by $R = 0 + j1$:
$Ic' = -Qc$
\( \text{Qc'} = \text{Ic} \)

(negate Q, then swap)

To subtract 90 degrees,

multiply by \( R = 0 - j1 \):

\( \text{Ic'} = \text{Qc} \)
\( \text{Qc'} = -\text{Ic} \)

(negate I, then swap)

To rotate by phases of less than 90 degrees, multiplying by numbers of the form "\( R = 1+/- jK \)". K will be decreasing powers of two, starting with \( 2^0 = 1.0 \). Therefore, \( K = 1.0, 0.5, 0.25, \text{etc.} \). (We use they symbol "L" to designate the power of two itself: 0, -1, -2, etc.)

Since the phase of a complex number "\( I + jQ \)" is \( \text{atan}(Q/I) \), the phase of "\( I + jK \)" is \( \text{atan}(K) \). Likewise, the phase of "\( I - jK \)" = \( \text{atan}(-K) = -\text{atan}(K) \).

To add phases we use "\( R = 1 + jK \)"; to subtract phases we use "\( R = 1 - jK \)". Since the real part of this, Ir, is equal to 1, we can simplify our table of equations to add and subtract phases for the special case of CORDIC multiplications to:

To add a phase,

multiply by \( R = 1 + jK \):

\( \text{Ic'} = \text{Ic} - K \cdot \text{Qc} = \text{Ic} - (2^{-L}) \cdot \text{Qc} = \text{Ic} - (\text{Qc} >> L) \)
\( \text{Qc'} = \text{Qc} + K \cdot \text{Ic} = \text{Qc} + (2^{-L}) \cdot \text{Ic} = \text{Qc} + (\text{Ic} >> L) \)

To subtract a phase,

multiply by \( R = 1 - jK \):

\( \text{Ic'} = \text{Ic} + K \cdot \text{Qc} = \text{Ic} + (2^{-L}) \cdot \text{Qc} = \text{Ic} + (\text{Qc} >> L) \)
\( \text{Qc'} = \text{Qc} - K \cdot \text{Ic} = \text{Qc} - (2^{-L}) \cdot \text{Ic} = \text{Qc} - (\text{Ic} >> L) \)

Let's look at the phases and magnitudes of each of these multiplier values to get more of a feel for it. The table below lists values of L, starting with 0, and shows the corresponding values of K, phase, magnitude, and CORDIC Gain (described below):
Since we're using powers of two for the $K$ values, we can just shift and add our binary numbers. That's why the CORDIC algorithm doesn't need any multiplies!

You can see that starting with a phase of 45 degrees, the phase of each successive $R$ multiplier is a little over half of the phase of the previous $R$. That's the key to understanding CORDIC: we will be doing a "binary search" on phase by adding or subtracting successively smaller phases to reach some target phase. The sum of the phases in the table up to $L = 3$ exceeds 92 degrees, so we can rotate a complex number by +/- 90 degrees as long as we do four or more "$R = 1 +/- jK$" rotations. Put that together with the ability to rotate +/-90 degrees using "$R = 0 +/- j1$", and you can rotate a full +/-180 degrees. Each rotation has a magnitude greater than 1.0. That isn't desirable, but it's the price we pay for using rotations of the form $1 + jK$. The "CORDIC Gain" column in the table is simply a cumulative magnitude calculated by multiplying the current magnitude by the previous magnitude. Notice that it converges to about 1.647; however, the actual CORDIC Gain depends on how many iterations we do. (It doesn't depend on whether we add or subtract phases, because the magnitudes multiply either way.)

The major steps in carrying out the experiment:

i) Write a c program.

ii) Build the application.
iii) Download the hex file on to the ARM board and check the result.

Writing the program:

Create a new folder as cordic_exp. In the cordic_exp create a folder called src.

To write source code, you have to open keil software and can write program in new file. While saving the file save it with .c extension. The program code can be seen in figure below.

![Cordic Program Code](image)

After writing the code we to add all necessary files and save the code, after saving we have to compile and built the code. For building we can press F7 button or can click on either of this icons.

![Build Options](image)
After that we can see the results or can find out any errors or warning in the Build output window as shown in figure below.

We can see that there are 0 errors and few warning, we can ignore warning as we can get correct output with warnings also.

When there are no errors then we can download this hex file in ARM using FlashMagic.

To download the cordic.hex file on to the microcontroller

**FOLLOW THE SAME STEPS GIVEN IN EXPERIMENT 3**

**RESULT:**

Connect CRO probe positive and Ground pins to the DAC output of LPC2148, ground pin of Board respectively. Now, to put the microcontroller in RUN mode, press RESET.
Figure: sine wave

Source code (Implementation of CORDIC DSP algorithm)

MAIN CODE

```c
#include <LPC214X.H>
unsigned int count=0,count2=0,count_samples=0,count_samples_value = 0;

unsigned int sample[330]; // array stores the samples
unsigned int sampleIndex=0; // index for sample updation.
unsigned int DACIndex=0;    // index for writing into DAC.

//unsigned int phase_change_value_2 = 36,phase_change_value_1 = 1;

unsigned int phase=0;
unsigned int phaseIncValue = 9*65536; // corresponds to 50 khz at 500 ksp sample rate (36 * 65536)

unsigned int sampleIncrement=1; // number of samples to be

unsigned int amplitude=1;
unsigned char compute=1;

//void T0isr(void) __attribute__ ((interrupt));

//void def_isr(void) __attribute__ ((interrupt));

void processCommand(void);
void compute_samples(void);
void CORDIC(unsigned int,unsigned int*);
int calc_pow(int,int);

__irq void T0isr (void)
{
    ++count_samples;
}
```

www.sriindugroup.org
if(DACIndex == 20) 
DACIndex = 0;

DACIndex = DACIndex + 1;
DACR = sample[DACIndex];

T0IR |= 0x00000001;  // Clear match 0 interrupt

VICVectAddr = 0x00000000;  // Dummy
write to signal end of interrupt
if(count_samples == count_samples_value)
    count_samples = 0;
}

__irq void def_isr(void)
{
    Tx_string("Unknown interrupt");
    VICVectAddr = 0x00000000;  // Dummy
write to signal end of interrupt
}

int main()
{
    unsigned int i, j;

    PINSEL0 = 0x00000005;  // enabling the UART0 pins

    // PORT(0.2) configured as input by making bits 5,4 as 00, bits 4,3,2,1 0101 for UART0
    // bits 7,6, are zero for using p0.3

    PINSEL1 = 0x00008000;  // to use P0.25 as DAC output pin.

    // PLL configuration

    InitUart0();

    // IO0DIR = IO0DIR & PIN;  // only our output pin will be configured as output

    processCommand();
    compute_samples();

    T0TCR = T0TCR | (0x01);  // to set bit0 to 1 to enable timer counter 0
T0PR = 0x01;

T0MR0= 50 ;
T0MCR = T0MCR | (0x03); // to set the bit1 to 1 to enable the reset feature for Timer0 when MR0 matches
// bit0 =1 to generate interrupt

VICIntSelect = 0x00000000;
    VICDefVectAddr = (unsigned long) def_isr;
    VICVectAddr0 = (unsigned long) T0isr;
    VICVectCntl0= 0x00000024;
    VICIntEnable = 0x00000010;   // to enable TIMER0

    //temp_1=0;
    while(1)
    {
    
    }

void compute_samples(void)
{
    unsigned int i=0,temp;
    int flag=0;
    phase=0;
    for(i=0;i<320;i++)
    {
        phase += phaseIncValue;
        if(phase >= 0x01680000) // 0x01680000 corresponds to 360 degress
            phase=phase-0x01680000;   // to bring angle in 0 to 360 degress range;
        CORDIC(phase,&sample[i]);
        adc_out(sample[i]);
        Tx_char("n");
        sample[i] = (sample[i]) << 6 ;
    }
}

void processCommand(void)
{
    unsigned int i=0;
    // debugging purpose...we have not fixed frequency for
    modulation techniques
    count_samples_value = 20; // these are the values which indicate the no of samples
    required to make 360 degrees phase
void CORDIC(unsigned int phase_in,unsigned int *sin_out)
{
    int flag=0;  // 0 for less than 180 and 1 for more than 180
    int phase_val[14]={2949120,1740967,919879,466945,234378,117303,58666,29334,14667,7333,3666,1833,916,458};
    /*
     --actual value=45.00000000";
     --actual value=26.565051
     --actual value=14.036243
     --actual value=7.125016
     --actual value=3.576334
     --actual value=1.789911
     --actual value=0.895174
     --actual value=0.447614
     --actual value=0.223811
     --actual value=0.111906
     --actual value=0.055953
     --actual value=0.027976
     --actual value=0.013988
     --actual value=0.006994
    */

    int acc_angle,L;
    int I,Q,varI,varQ,tempI,tempQ,scaledI,scaledQ;
    int I_sign,Q_sign;
    int angle_90,zero_angle;
    while(phase_in >= 0x01680000)  // 0x01680000 corresponds to 360 degrees
        phase_in=phase_in - 0x01680000;   // to bring angle in 0 to 360 degrees range;

    if(phase_in > 0x00B40000)
    {
        phase_in = phase_in - 0x00B40000;
        flag=1;
    }

    if ((phase_in > zero_angle) && (phase_in < angle_90))    // -- when angle less than 90 degrees
    {
\begin{verbatim}
I = 256;
Q = 0;
acc_angle = 0;
I_sign=0; // cos positive
Q_sign =0; // sin positive
else         // means (temp_phase_in>=angle_90) and less than 180
{
    I = 0;
    Q = 256;
    acc_angle = angle_90;
    I_sign=1;  // cos negative
    Q_sign =0; // sin positive
}
templ=I;
tempQ=Q;
for(L=0;L<=8;L++)
{
    varI=templ;
    varQ=tempQ;
    scaledI = varI >> L;     // (2^-L) I
    scaledQ = varQ >> L;    // ((2^L) Q
    if (phase_in > acc_angle)
    {
        templ=varI-scaledQ;
        tempQ=varQ+scaledI;
        acc_angle=acc_angle+phase_val[L];
    }
    else
    {
        templ=varI+scaledQ;
        tempQ=varQ-scaledI;
        acc_angle=acc_angle-phase_val[L];
    }
}
if(Q_sign==1)
    *sin_out = -tempQ;
else
    *sin_out = tempQ;
if(flag==1) // if angle is greater than 180
{
    *sin_out = -(*sin_out);
}
*sin_out=*sin_out+512;
\end{verbatim}

UART 0 CODE
```c
#include <LPC214X.H>

#define DESIRED_BAUDRATE 19200
#define CRYSTAL_FREQUENCY_IN_HZ 12000000
#define PCLK CRYSTAL_FREQUENCY_IN_HZ*5   // since VPBDIV=0x01
#define DIVISOR (PCLK/(16*DESIRED_BAUDRATE))

void InitUart0(void)
{
    /* U0LCR: UART0 Line Control Register
       0x83: enable Divisor Latch access, set 8-bit word length,
       1 stop bit, no parity, disable break transmission */
    U0LCR=0x83;

    /* VPBDIV: VPB bus clock divider
       0x01: PCLK = processor clock */
    VPBDIV=0x01;

    /* U0DLL: UART0 Divisor Latch (LSB) */
    U0DLL=DIVISOR&0xFF;

    /* U0DLM: UART0 Divisor Latch (MSB) */
    U0DLM=DIVISOR>>8;

    /* U0LCR: UART0 Line Control Register
       0x03: same as above, but disable Divisor Latch access */
    U0LCR=0x03;

    /* U0FCR: UART0 FIFO Control Register
       0x05: Clear Tx FIFO and enable Rx and Tx FIFOs */
    U0FCR=0x05;
}

char Tx_char(char ch)
{
    if (ch=='\n')
    {
        //wait until Transmit Holding Register is empty
        while (!(U0LSR&0x20)) {}

        //then store to Transmit Holding Register
        U0THR='\r';
    }

    //wait until Transmit Holding Register is empty
    while (!(U0LSR&0x20)) {}

    //then store to Transmit Holding Register
    U0THR=ch;
}```
return ch;
}

char Rx_char(void)
{
    char ch;
    // wait until there's a character to be read
    while (!(U0LSR&0x01)) {}  

    // then read from the Receiver Buffer Register
    ch=U0RBR;
    return ch;
}

int IsUart0Ready ()
{
    // returns non-zero if a char is received on UART0
    return(U0LSR & 0x01) ;
}

int Tx_string(char *s)
{
    int i=0;
    while(s[i]!="\0")
    {
        Tx_char(s[i]);
        i++;
    }
    return(i);
}