Chapter 4

Enhancing ARM7 architecture by embedding RTOS

4.1 ARM7 architecture
4.2 ARM7TDMI processor core
4.3 Embedding RTOS on ARM7TDMI architecture
4.4 Block diagram of the Design
4.5 Hardware Design of the System
4.6 Software Design of the System
4.7 Programming LPC2138 using Flash Magic
4.8 Simulation Results
4.9 Summary & Discussion
In this chapter I have described about the enhancement of ARM7/ ARM7 TDMI Architecture[2][3] by embedding multiple tasks[16] running together thereby utilizing the power of real time operating system on ARM7 TDMI Architecture. By doing so, one can realize the process level parallelism. ARM7 architecture is the most popular architecture in version 3 and later on version 4T with ARM7 TDMI is highly popular.

4.1 ARM7 Architecture:

ARM cores use a 32-bit, Load-Store RISC architecture[2][3]. That meaning that the core cannot directly manipulate the memory. All data manipulation must be done by loading registers with information located in memory, performing the data operation and then storing the value back to memory. There are 37 total registers in the processor. However, that number is split among seven different processor modes. The seven processor modes are used to run user tasks, an operating system, and to efficiently handle exceptions such as interrupts. Some of the registers within each mode are reserved for specific use by the core, while most are available for general use. The reserved registers that are used by the core for specific functions are r13 is commonly used as the stack pointer (SP), r14 as a link register (LR), r15 as a program counter (PC), the Current Program Status Register (CPSR), and the Saved Program Status Register (SPSR). The SPSR and the CPSR contain the status and control bits specific to the properties the processor core is operating under. These properties define the operating mode, ALU status flags; interrupt disable/enable flags and whether the core is operating in 32-bit ARM or 16-bit Thumb state.

There are 37 total registers divided among seven different processor modes. Figure 4.1 shows the bank of registers[2][3] visible in each mode. User mode, the only non-privileged mode, has the least number of total registers visible. It has no SPSR and limited access to the CPSR. FIQ and IRQ are the two interrupt modes of the CPU. Supervisor mode is the default mode of the processor on start up or reset. Undefined mode traps unknown or illegal instructions when they are passed through the pipeline. Abort mode traps illegal memory accesses as a result of fetching instructions or accessing data.
Finally, system mode, which uses the user mode bank of registers, was introduced to provide an additional privileged mode when dealing with nested interrupts. Each additional mode offers unique registers that are available for use by exception handling routines. These additional registers are the minimum number of registers required to preserve the state of the processor, save the location in code, and switch between modes. FIQ mode, however, has an additional five banked registers to provide more flexibility and higher performance when handling critical interrupts. When the ARM core is in Thumb state, the registers banks are split into low and high register domains.

The majority of instructions in Thumb state have a 3-bit register specifier. As a result, these instructions can only access the low registers in Thumb, R0 through R7. The high registers, R8 through R15, have more restricted use. Only a few instructions have access to these registers.

<table>
<thead>
<tr>
<th>System &amp; User</th>
<th>FIQ</th>
<th>Supervisor</th>
<th>Abort</th>
<th>IRQ</th>
<th>Undefined</th>
</tr>
</thead>
<tbody>
<tr>
<td>R0</td>
<td>R0</td>
<td>R0</td>
<td>R0</td>
<td>R0</td>
<td>R0</td>
</tr>
<tr>
<td>R1</td>
<td>R1</td>
<td>R1</td>
<td>R1</td>
<td>R1</td>
<td>R1</td>
</tr>
<tr>
<td>R2</td>
<td>R2</td>
<td>R2</td>
<td>R2</td>
<td>R2</td>
<td>R2</td>
</tr>
<tr>
<td>R3</td>
<td>R3</td>
<td>R3</td>
<td>R3</td>
<td>R3</td>
<td>R3</td>
</tr>
<tr>
<td>R4</td>
<td>R4</td>
<td>R4</td>
<td>R4</td>
<td>R4</td>
<td>R4</td>
</tr>
<tr>
<td>R5</td>
<td>R5</td>
<td>R5</td>
<td>R5</td>
<td>R5</td>
<td>R5</td>
</tr>
<tr>
<td>R6</td>
<td>R6</td>
<td>R6</td>
<td>R6</td>
<td>R6</td>
<td>R6</td>
</tr>
<tr>
<td>R7</td>
<td>R7</td>
<td>R7</td>
<td>R7</td>
<td>R7</td>
<td>R7</td>
</tr>
<tr>
<td>R8</td>
<td>R8 fiq</td>
<td>R8</td>
<td>R8</td>
<td>R8</td>
<td>R8</td>
</tr>
<tr>
<td>R9</td>
<td>R9 fiq</td>
<td>R9</td>
<td>R9</td>
<td>R9</td>
<td>R9</td>
</tr>
<tr>
<td>R10</td>
<td>R10 fiq</td>
<td>R10</td>
<td>R10</td>
<td>R10</td>
<td>R10</td>
</tr>
<tr>
<td>R11</td>
<td>R11 fiq</td>
<td>R11</td>
<td>R11</td>
<td>R11</td>
<td>R11</td>
</tr>
<tr>
<td>R12</td>
<td>R12 fiq</td>
<td>R12</td>
<td>R12</td>
<td>R12</td>
<td>R12</td>
</tr>
<tr>
<td>R13</td>
<td>R13 fiq</td>
<td>R13 svc</td>
<td>R13 abt</td>
<td>R13 irq</td>
<td>R13 und</td>
</tr>
<tr>
<td>R14</td>
<td>R14 fiq</td>
<td>R14 svc</td>
<td>R14 abt</td>
<td>R14 irq</td>
<td>R14 und</td>
</tr>
<tr>
<td>CPSR</td>
<td>CPSR</td>
<td>CPSR</td>
<td>CPSR</td>
<td>CPSR</td>
<td>CPSR</td>
</tr>
<tr>
<td>SPSR fiq</td>
<td>SPSR svc</td>
<td>SPSR abt</td>
<td>SPSR irq</td>
<td>SPSR und</td>
<td></td>
</tr>
</tbody>
</table>

**Figure 4.1 Registers Available In Different Modes In ARM7**

### 4.1.1 Current Program Status Register (CPSR):

This is a Program Status Register which is used to monitor the current status of the conditional codes along with that it also consists of various control bits;
this can be used in user level programs. Once status of conditional codes bits are available in CPSR it can be utilized for decision making process based on conditional codes testing. The usual approach is to check for set or reset condition and then branch to particular location based on the result. Figure 4.2 shows the format of CPSR[2][3].

![Current Program Status Register (CPSR)](image)

Total it consists of four conditional code bits.

- **N: Negative:**
  Bit sets when after performing ALU operation if result is negative.

- **Z: Zero:**
  Bit sets when all (32) the bits are zero after ALU operation.

- **C: Carry:**
  Bit sets when there is end carry occurs after ALU operation.

- **V: overflow:**
  Bit sets when overflow occur in signed arithmetic after ALU operation.

CPSR also consist of mode bits by which one can change the modes by these bits. It has got three more control bits which are T, F and I which are used for changing the state, enabling & disabling of FIQ and IRQ respectively.

### 4.2 ARM7TDMI Processor Core:

The ARM7TDMI has a core based on the fourth version of the ARM architecture[2][3]. This implementation uses a three stage pipeline - a standard fetch-decode-execute organization. It features a unified cache, as
well as the Thumb extension permitting 32-bit and 16-bit operation. It is completely forward compatible, meaning that any code written for this core will be compatible with any new core releases, such as ARM9 or ARM10. This core also includes the on-chip debug extension discussed in the previous training module. The core is successful mainly because of the extremely small but high performance processor - slightly more than 70,000 transistors in all and with extremely low power consumption.

Figure 4.3 ARM7TDMI Block Diagram

Figure 4.3 shows[3] the register bank in the center of the diagram, plus the required address bus and data bus. The multiplier, in-line barrel shifter, and ALU are also shown. In addition, the diagram illustrates the in-line decompression process of Thumb instructions while in the decode stage of the pipeline. This process creates a 32-bit ARM equivalent[2][3] instruction from the 16-bit Thumb instruction, decodes the instruction, and passes it on to the execute stage.
T-D-M-I stands for:
- **Thumb**, which is a 16-bit instruction set extension to the 32-bit ARM architecture, referred as states of the processor.
- "D" and "I" together comprise the on-chip debug facilities offered on all ARM cores. These stand for the Debug signals and EmbeddedICE logic, respectively.
- The M signifies the support for 64-bit results and an enhanced multiplier, resulting in higher performance. This multiplier is now standard on all ARMv4 architectures and above.

### 4.2.1 Thumb 16-bit Instructions:

<table>
<thead>
<tr>
<th>THUMB state</th>
<th>ARM state</th>
</tr>
</thead>
<tbody>
<tr>
<td>R0</td>
<td>R0</td>
</tr>
<tr>
<td>R1</td>
<td>R1</td>
</tr>
<tr>
<td>R2</td>
<td>R2</td>
</tr>
<tr>
<td>R3</td>
<td>R3</td>
</tr>
<tr>
<td>R4</td>
<td>R4</td>
</tr>
<tr>
<td>R5</td>
<td>R5</td>
</tr>
<tr>
<td>R6</td>
<td>R6</td>
</tr>
<tr>
<td>R7</td>
<td>R7</td>
</tr>
<tr>
<td>Stack Pointer (SP)</td>
<td>Stack Pointer (R13)</td>
</tr>
<tr>
<td>Link Register (LR)</td>
<td>Link Register (R14)</td>
</tr>
<tr>
<td>Program Counter (PC)</td>
<td>Program Counter (R15)</td>
</tr>
<tr>
<td>CPSR</td>
<td>CPSR</td>
</tr>
<tr>
<td>SPSR</td>
<td>SPSR</td>
</tr>
</tbody>
</table>

**Figure 4.4 Differentiation of ARM & THUMB State**

With growing code and data size, memory contributes to the system cost. The need to reduce memory cost leads to smaller code size and the use of narrower memory. Therefore ARM developed a modified instruction set to give market-leading code density for compiled standard C language. There is also the problem of performance loss due to using a narrow memory path,
such as a 16-bit memory path with a 32-bit processor. The processor must take two memory access cycles to fetch an instruction or read and write data. To address this issue, ARM introduced another set of reduced 16-bit instructions labeled Thumb, based on the standard ARM 32-bit instruction set. For Thumb to be used, the processor must go through a change of state from ARM to Thumb in order to begin executing 16-bit code. This is because the default state of the core is ARM. Therefore, every application must have code at boot up that is written in ARM. If the application code is to be compiled entirely for Thumb, then the segment of ARM boot code must change the state of the processor. Once this is done, 16-bit instructions are fetched seamlessly into the pipeline without any result. It is important to note that the architecture remains the same but the operating register states are different in ARM and THUMB as shown[3] in Figure 4.4. The instruction set is actually a reduced set of the ARM instruction set and only the instructions are 16-bit; everything else in the core still operates as 32-bit. An application code compiled in Thumb is 30% smaller on average than the same code compiled in ARM and normally 30% faster when using narrow 16-bit memory systems.

4.2.2 Debug Extensions:
The Debug extensions[2][3] to the core add scan chains to monitor what is occurring on the data path of the CPU. Signals were also added to the core so that processor control can be handed to the debugger when a breakpoint or watch point has been reached. This stops the processor enabling the user to view such characteristics as register contents, memory regions, and processor status.

4.2.3 EmbeddedICE Logic:
In order to provide a powerful debugging environment for ARM-based applications the EmbeddedICE logic[2][3] was developed and integrated into the ARM core architecture. It is a set of registers providing the ability to set hardware breakpoints or watch points on code or data. The EmbeddedICE logic monitors the ARM core signals every cycle to check if a breakpoint or watch point has been hit. Lastly, an additional scan chain is used to establish contact between the user and the EmbeddedICE logic. Communication with
the EmbeddedICE logic from the external world is provided via the test access port, or TAP, controller and a standard IEEE 1149.1 JTAG connection. The advantage of on-chip debug solutions is the ability to rapidly debug software, especially when the software resides in ROM. This is critical in shortening the development cycle. The use of Multi-ICE and EmbeddedICE provides full debug capabilities for a processor integrated deep inside an ASIC, even in a production version of a consumer product.

4.2.4 Three-stage Pipeline ARM Organization:
The major components of an ARM organization with 3-stage pipeline[2][3] are:

- **Register banks** – stores the processor state. It has two read ports and one write port which can each be used to access any register, plus an additional read port and an additional write port that give special access to r15, the program counter (PC).
- **Barrel shifter** – can shift or rotate one operand by any number of bits.
- **ALU** – performs the arithmetic and logic functions required by the instruction set.
- **Address register and incremental** – select and hold all memory addresses and generate sequential addresses when required.
- **Data register** – hold data passing to and from memory.
- **Instruction decoder and associated control logic.**

Two registers operands are accessed, in a single-cycle data processing instruction. The value on the B bus is shifted and combined with the value on the A bus in the ALU, then the result is written back into the register bank and also into the address register to be used as the address for the next instruction fetch if needed.

ARM7 comprises of 3-stage pipeline having following pipeline stages:

- **Fetch**
  The instruction is fetched from memory and placed in the instruction pipeline.
- **Decode**
  The instruction is decoded and the data path control signals prepared for the next cycle. In this stage, the instruction owns the decode logic but not
Execute

The instruction owns the data path: The register bank is read. An operand is shifted. The ALU result is generated, and written back into a destination register.

Figure 4.5 ARM Single-cycle Instruction 3-stage Pipeline Operation

Figure 4.5 shows[3] at any point of time, three different instructions may occupy each of these stages, so the hardware in each stage has to be capable of independent operation. When the processor is executing simple data processing instructions the pipeline enables one instruction to be completed every clock cycle (that is, the throughput is one instruction per cycle), although an individual instruction takes three clock cycles to complete, that is, it has a three-cycle latency (latency pipeline).

4.2.5 ARM Exceptions:

The ARM architecture defines the following types of exceptions[2][3] (listed in the order of decreasing priority):

- **Reset** starts the processor from a known state and renders all other pending exceptions irrelevant.
- **Data abort** exception is raised by memory management hardware when a load or store instruction violates memory access permissions.
- **Fast interrupt** exception is raised whenever the processor receives an interrupt signal from the designated fast interrupt source.
- **Normal interrupt** exception is raised whenever the processor receives an interrupt signal from any non-fast interrupt source.
- **Prefetch abort** exception is raised by memory management hardware when memory access permissions are violated during instruction fetch.
- **Software interrupt** exception is raised by a special instruction, typically to request an operating system service.
- **Undefined** instruction exception is generated when trying to decode an instruction that is supported neither by the main integer core nor by one of the coprocessors.

<table>
<thead>
<tr>
<th>Exception</th>
<th>Mode</th>
<th>Priority</th>
<th>IV Address</th>
</tr>
</thead>
<tbody>
<tr>
<td>Reset</td>
<td>Supervisor</td>
<td>1</td>
<td>0x00000000</td>
</tr>
<tr>
<td>Undefined instruction</td>
<td>Undefined</td>
<td>6</td>
<td>0x00000004</td>
</tr>
<tr>
<td>Software interrupt</td>
<td>Supervisor</td>
<td>6</td>
<td>0x00000008</td>
</tr>
<tr>
<td>Prefetch Abort</td>
<td>Abort</td>
<td>5</td>
<td>0x0000000C</td>
</tr>
<tr>
<td>Data Abort</td>
<td>Abort</td>
<td>2</td>
<td>0x00000010</td>
</tr>
<tr>
<td>Interrupt</td>
<td>IRQ</td>
<td>4</td>
<td>0x00000018</td>
</tr>
<tr>
<td>Fast interrupt</td>
<td>FIQ</td>
<td>3</td>
<td>0x0000001C</td>
</tr>
</tbody>
</table>

Table 4.1 ARM List of Exception

Except for the reset exception, all exceptions are handled in a similar way: the processor switches to the corresponding execution mode, saves the address of the instruction following the exception entry instruction in R14 of the new mode, saves the old value of CPSR to SPSR of the new mode, disables IRQ (in case of a fast interrupt, FIQ is also disabled), and starts execution from the relevant exception vector.

Table 4.1 shows the list of exceptions with appropriate mode along with its interrupt vector Address.

Table 4.2 shows the list of exceptions in descending order along with F bit and I bit.
### Table 4.2 Exception List With Priority

<table>
<thead>
<tr>
<th>Exceptions</th>
<th>Priority</th>
<th>F bit</th>
<th>I bit</th>
</tr>
</thead>
<tbody>
<tr>
<td>RESET</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>DATA ABORT</td>
<td>2</td>
<td>-</td>
<td>1</td>
</tr>
<tr>
<td>FIQ</td>
<td>3</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>IRQ</td>
<td>4</td>
<td>-</td>
<td>1</td>
</tr>
<tr>
<td>Pre-fetch abort</td>
<td>5</td>
<td>-</td>
<td>1</td>
</tr>
<tr>
<td>SWI</td>
<td>6</td>
<td>-</td>
<td>1</td>
</tr>
<tr>
<td>Undefined Instructions</td>
<td>6</td>
<td>-</td>
<td>1</td>
</tr>
</tbody>
</table>

### 4.3 Embedding RTOS On ARM7TDMI Architecture:

For enhancing the ARM 7/ARM7 TDMI architecture I have prepared the design which uses Soft Real Time Operating System that would be using pre-emptive scheduling for the tasks and thereby showing multitasking of different peripherals connected to the ARM7 controller[16]. I have used the evaluation board of the ARM 7 and connected peripherals externally. RTOS is developed in KEIL simulator. USB loader is used to load the program in the controller.

### 4.3.1 Introduction Of Operating System:

Operating systems[18] are software environments that provide a buffer between the user and the low level interfaces to the hardware within a system. Key points in OS are kernel, task list, scheduler, time slicing etc.

These operating systems provide additional facilities allowing applications that would normally interface directly with the microprocessor architecture to use interrupts and drive peripherals to do so without the operating system blocking such activities. A characteristic of a real-time operating system is its defined
response time to external stimuli. If a peripheral generates an interrupt, a real-time system will acknowledge and start to service it within a maximum defined time.

Writing a small to medium size embedded application can be nearly identical from a code function standpoint whether an infinite main loop or RTOS is used. In many cases when a RTOS is not used, a programmer must write the equivalent of a limited scheduler to handle events. The advantage of a RTOS is that a programmer can use a tested and reliable solution to achieve the same goal. In addition, the RTOS[18] supplies features such as priority, semaphores and queues that makes the structure of the application much easier than if a RTOS was not used.

4.3.2 Types Of Real Time Operating System (RTOS):
Real Time Operating Systems are mainly divided into 3 types[18] and are as follows:

- **Hard Real Time Operating System:**
  - Hard Real Time Operating Systems are those which on failure to meet the time Constraints lead to system failure and major disaster.
  - Example: - Missile Launching, Medical devices etc.

- **Firm Real Time Operating System:**
  - Firm Real Time Operating Systems can miss some deadlines, but eventually performance will degrade if too many are missed.
  - Example: computer system, weather forecasting, seismic sensors.

- **Soft Real Time Operating Systems:**
  - Soft Real Time Operating Systems are those which on failure to meet the time constraints may lead to degradation of the performance of the system. Their failure doesn’t lead to major disaster.
  - Example: - Mobile phones, Set top box, consumer electronics, etc.
4.4 Block Diagram Of The Design:

**LPC2138 Board** - LPC2138[17] is a main controller and it is the heart of the system. The evaluation board includes inbuilt peripherals like 8 LEDs, UART, all the 45 GPIO pins, potentiometer for ADC, JTAG standard port etc. LPC2138 is responsible for the control of the whole system and coordination work of each module. RTOS with pre-emptive scheduling i.e. it will give the control to the task with higher priority and complete it first and then carry on the regular scheduling between the peripherals that will show the multitasking between the peripherals, multitasking is done on following Man-machine interaction peripherals.

1. **LEDs:**
   8 LEDs are already inbuilt in the evaluation board of LPC2138 and all the LEDs form a series such that for a specific time 1 of the LED turns ON and after the completion of the time it turns OFF after that same process is carried with the next LED next to it. This process goes on continuously such that when the turn of last LED is finished again the 1st LED starts the process.

2. **UART:**
   The second task of the system is to send the data serially to the computer via DB-9 pin at the baudrate already set i.e. 9600. The data transmitted serially is ‘EMBEDDING RTOS ON ARM SERIES CONTROLLER’.

3. **16x2 LCD:**
   The third task of the system is to display data i.e. ‘BHARGAV’ on the LCD continuously

4. **Buzzer:**
   The final and 4th task of the controller is to ringing of the buzzer at every 10 seconds. Here for the count of 10 seconds a timer is used where it generates an interrupt at every 10 seconds. And the interrupt gives the
high signal at the pin where the buzzer is connected for few seconds and thus the buzzer rings at every 10 seconds

The general block diagram of the system is shown in Figure 4.6 which comprises of main controller LPC 2138, LED’s, LCD, UART and Buzzer.

Figure 4.6 General Block Diagram Of The System

4.5 Hardware Design Of The System:

System’s development cost was considered, as well as multitasking of various peripherals needs to take massive system resources when it is processing. Therefore, the system selects LPC2138 as the main controller. Figure 4.7 shows the evaluation board[20] of LPC 2138. LPC2138 is a 32-bit Microcontroller which is produced by NXP Corporation. LPC2138 is ARM7TDMI-S based high-performance 32-bit RISC Microcontroller with Thumb extensions 512KB on-chip Flash ROM with In-System Programming (ISP) and In-Application Programming (IAP), two 8 channel 10bit ADC32KB RAM, Vectored Interrupt Controller, two UARTs, one with full modem interface, two I2C serial interfaces.
Three 32-bit timers, Watchdog Timer, Real Time Clock with optional battery backup, Brown out detect circuit General purpose I/O pins. CPU clock up to 60 MHz, On-chip crystal oscillator and On-chip PLL.

4.6 Software Design Of The System:

Language used for programming is C - as the software platform, and adopting modularity design. So that it has good portability and expandability. Here RTOS is developed in the software KEIL 4 MDK version.

The system will initially load RTX_Config.c,RTL.c,LPC213x.c. The main body of the program consists of the following thing.

```c
int main (void) {
    /* program execution starts here */
    os_sys_init (job1); /* initialize and start task 1 */
}
```
Here initially in the main part of the program 1st task is initialized and then rests all the tasks are initialized in the task 1.

**RTX_Config.c** - This file keeps the details of how many tasks a user wants to Schedule at given time. Default count is 6 Tasks. It also keeps the details about timer, ticks, clocks, etc. all needed to calculate the amount of time used for time switching from 1 Task to another.

**rtl.h** - This file keeps the record of all the functions used to initialize the task, end the task, delay the task, etc;

For the programming related Tasks, I have made this system for the 4 Tasks[16] i.e. peripherals. The programming of the Tasks has been done in C language. Here every pin of ARM controller has capacity of doing different things i.e. for example pin P0.1 can be used as RXD0 or PWM3 or EINT0. So we need to assign specific values to the *PIN SELECT* command so that we can use our required pin as per our requirement.

In propose system I have programmed for 4 Tasks following are the 4 tasks:

1. Task 1 consists of 8 LEDs forming a specific pattern.
2. Task 2 consists of UART where *EMBEDDING RTOS ON ARM SERIES CONTROLLER* is continuously transmitted via DB9 pin.
3. Task 3 consists of LCD where data *BHARGAV* is continuously displayed.
4. Task 4 consists of Timer Interrupt where after every 10 seconds a buzzer sounds.

After completion of all the Tasks, these Tasks should be merged with the RTOS. And so we need to place all the 4 Tasks in the respective place in the RTOS, i.e. as explained earlier in the main body of the program consists of the initialization of the Task 1.
Flow Chart of the Software is shown below in Figure 4.8.

Program Start

First task initialized in the beginning

Task 1 initializes all other task

Sequentially all the task gets the control of the controller

Check for the interrupt?

No

Stops all the running tasks & execute the interrupt

Yes

Interrupt executed?

Not completed

Interrupt completed

End Program

Figure 4.8 Flow Chart Of The Program
After the initialization of the Task 1, the controls of the controller are given to the Task 1, in Task 1 all the other 3 Tasks are been initialized and so the functions related to all the 4 Tasks are outside the Main Body of the RTOS.

And because of the Round-Robin scheduling the controls are been shifted from 1 Task to the other Task. And for this system I have provided the time slice of 5msec. And this 5msec is initialized in the initializing file i.e. RTX_Config.c. For the system all the programming is done in KEIL MDK4 (Microcontroller development kit 4) version.

4.7 Programming The LPC2138 Using Flash Magic:

For loading the hex file generated in KEIL 4 MDK version[43], Flash Magic software is used. Figure 4.9 shows screenshot of the Flash Magic software.

![Figure 4.9 Screenshot Of The Software Flash Magic.](image-url)
Following are the steps to load the program in the controller:

1. Adjust the baudrate to 9600 bps (bits per second).
2. Oscillation frequency at 12 MHz.
3. Select the device LPC2138.
4. Select the COM port as per the requirement.
5. Select the HEX (Hexadecimal) file from the computer of the program that is to be loaded.
6. Click the Start button to start the loading of the program into the controller.

4.8 Simulation Results:

Below in the Figure 4.10 shows screen shot of the multitasking of two peripherals i.e. 8 LEDs are forming a series and data i.e. ‘BHARGAV’ is displayed on the LCD.

![Multitasking Of Two Tasks](image)

Figure 4.10 Multitasking Of Two Tasks

The serial communication is seen in the screenshot of HYPERTERMINAL SCREEN shown in Figure 4.11. Multitasking of Task4 (Buzzer) and Task2 (serial communication through UART) is shown in Figure 4.12, similarly multitasking of Task1 (LEDs) and Task3 (Character Display on LCD) is shown.
in Figure 4.13 in the Keil environment. Figure 4.14 shows the screen shot of the all four Tasks running in the Keil environment. The General purpose I/O pins are used to indicate the running of individual task. GPIO-0 is used to indicate the running of Task4 and Task2 while GPIO-1 is used to indicate the running of Task1 and Task3.

Figure 4.11 Screenshot Of Hyper-terminal (UART)

Task 2 and Task 4:

Figure 4.12 Screenshot of GPIO-0 Shows Task2 and Task4
Task 1 and Task 3:

Figure 4.13 Screenshot of GPIO-1 Shows Task1 and Task3

Figure 4.14 Screenshot Showing All Four Tasks Running
4.9 Summary And Discussion:

Embedding RTOS on ARM7 TDMI architecture offers great enhancement in the system. With simultaneous execution of multiple tasks, process level parallelism is achieved. Apart from that it offers following benefits

- Low cost for startups
- Easy to use
- Advanced function and high level computing possible for high end use
- Improves efficiency and transaction processing
- Less power consumption
- Compact Size
- Enables device automation at lower cost
- Easy to add peripherals and remove peripherals

On the other side, every system has its own positive and negative points, below are the points of future expansion

- Number of peripherals used here are 4 but can be increased up to more than 10.
- Programming of the RTOS is in the C platform but we can upgrade it to LINUX platform.
- We can change the scheduling of the system as per our requirement.
- If speed is the important criteria then we can upgrade our controller from present ARM 7 to other ARM versions like ARM 9 or ARM 11 or even to CORTEX Family.