DEVELOPMENT OF REAL TIME OPERATING SYSTEM FOR PIC18F MICROCONTROLLERS FOR EDUCATIONAL PURPOSES

By

MOHAMED TAG ELSIR MOHAMED ELHUSSEIN

FINAL REPORT

Submitted to the Electrical and Electronics Engineering Programme
in Partial Fulfillment of the Requirements
for the Degree
Bachelor of Engineering (Hons)
(Electrical and Electronics Engineering)

Universiti Teknologi Petronas
Bandar Seri Iskandar
31750 Tronoh
Perak Darul Ridzuan

CERTIFICATION OF APPROVAL

DEVELOPMENT OF REAL TIME OPERATING SYSTEM FOR PIC18F MICROCONTROLLER FOR EDUCATIONAL PURPOSES

By

MOHAMED TAG ELSIR MOHAMED ELHUSSEIN

A project dissertation submitted to the
Electrical & Electronics Engineering Programme
Universiti Teknologi PETRONAS
in partial fulfilment of the requirement for the
Bachelor of Engineering (Hons)
(Electrical & Electronics Engineering)

Approved:

_____

Mr. Patrick Sebastian
Project Co-supervisor

UNIVERSITI TEKNOLOGI PETRONAS
TRONOH, PERAK

December 2009

CERTIFICATION OF ORIGINALITY

This is to certify that I am responsible for the work submitted in this project, that the original work is my own except as specified in the references and acknowledgements, and that the original work contained herein have not been undertaken or done by unspecified sources or persons.

_____

Mohamed Tag Elsir Mohamed Elhussein

ABSTRACT

Real **T**ime **O**perating **S**ystem (**RTOS**) is a small operating system designed to manage the peripherals of Microcontrollers and exhibit a low level layer to enhance the parallel execution of multiple programs. In addition to that, RTOSes are most of concern about guarantee the processing at real time.

This project aims to implement and develop **RTOS** on PIC18Fxxx family. This RTOS is to be developed under MPLAB IDE integrated development environment. The kernel of this RTOS is written in Assembly language while the users may use both assembly and C to develop their applications. A previous RTOS project called **PICos18** developed by Pragamtec inc. is being considered. The selection of this system is due to its free license and the availability of its documentations. PICos18 is based on **OSEK/VDX** (German/French industrial standards for operating systems).

The main contribution in this project is first, by developing RTOS to review and demonstrate the concept of RTOS and secondly, by developing drivers and application compatible with the developed RTOS and finally presenting the developed RTOS in educational form for future use as a teaching tool in microcontroller-based courses.

ACKNOWLEDGEMENTS

Foremost, my utmost gratitude is to ALLAH the All-Mighty for his uncountable graces upon me and for the successful completion of this project in due course of time.

Enormous thanks to my family members for their priceless support and continuous encouragement. Special gratitude is forwarded to my Mother for her continuous and unlimited support that kept me going. There is no words can fulfill her effort.

A respectful gratitude goes to my supervisor, AP. Dr. Yap Vooi Voon and my co-supervisor Mr. Patrick Sebastian for their full support in the completion of this project. Their constant guidance, helpful comments and suggestions have helped me not only to complete but also to enhance the expected results of the project. Their kindness, valuable advices, friendly approach and patience will always be appreciated.

I would like also to express my thanks for the FYP committee for their guidance and management in making all projects run smoothly. A special gratitude is conveyed to Siti Hawa Tahir for her effort on monitoring and checking the reports to match the university's standards.

Lastly, great appreciation is to my friends, who were a constant source of support during my work. To all UTP lecturers, students and staff and to all whose their names are not mentioned here but they provided help directly or indirectly.

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

LIST OF ABBREVIATIONS

A/D:	**A**nalogue to **D**igital
CAN:	**C**ontroller **A**rea **N**etwork
D/A:	**D**igital to **A**nalogue
OS:	**O**perating **S**ystem
RTOS:	**R**eal **T**ime **O**perating **S**ystem
USB:	**U**niversal **S**erial **B**us
USART:	**A**ddressable **U**niversal **A**synchronous **R**eceiver/**T**ransmitter.
PIC:	**P**eripheral **I**nterface **C**ontroller
PICmicro:	**PIC micro**controller

CHAPTER 1
INTRODUCTION

1.1.Background of Study

Microcontrollers are widely used in embedded systems to control and manage the operation of devices, and other peripherals. There are many types of microcontrollers available in the market, however, the PIC16 and PIC18 microcontrollers -manufactured by Microchip- are the famous ones. They are featured with very useful hardware modules and peripherals (e.g. A/D converters, Timers, Interrupts, serial communication (I^2C, SPI, USB, CAN ...) etc. Most of microcontroller-based applications are programmed using Assembly and C languages. The conventional approach of programming microcontrollers is by using the round-robin programming methodology. In round robin programming methodology, programmers write all the required instructions and tasks inside an infinite loop which is continuously executing. So, round-robin programming approach does not provide convenient programming environment and does not help in reducing development time when the applications get complex. Additionally, round robin programming does not guarantee real time processing for tasks because it is probable that the microcontroller gets busy with checking other less important tasks while another time-critical task needs microcontroller attention. For the reasons mentioned and others, **R**eal **T**ime **O**perating **S**ystem (**RTOS**) is developed. However, the implementation of RTOS in such tight environments requires additional care for the overhead processing time and the utilization of the memory within the microcontroller.

1.2. Problem Statement

In this project, a RTOS is expected to be implemented on PIC18Fxxx environment. The proposed system should exhibit the concept of RTOS and

should demonstrate multitasking processing, tasks scheduling, time and events management in multiple tasks processing environment.

It is envisaged that software drivers would be developed to run other peripherals which interface with the microcontroller.

1.3. Objective and Scope of Study

The objectives of the project are:

- To develop RTOS for Microchip Peripheral interface controllers (**PIC**)
- To run the developed RTOS on one of the PIC devices (PIC18F452/8 is proposed).
- To develop drivers compatible with the developed RTOS to utilize the PIC's peripherals.
- To create an application relies on the new implemented system and demonstrate its functionality.

The scope of this project can be partitioned into 3 different complementing parts. The first part is about the understanding of the architecture and the operation of the high performance PIC microcontrollers (PIC18Fxxx series). The second part is about understanding the operational and structural concept of RTOS and the typical services which RTOS provides. The third part is about developing the RTOS on one of the PIC18F series. At the early phases of this project, the concern was held on studying the PIC18F452 microcontroller. The availability of this microcontroller in UTP store made it a good choice. An overview look has been made on the architecture of this PIC including: device hardware features, flash and data memories, timers, interrupt, I/O ports, instruction set and assembly programming procedure. For practical and economical considerations; MPLAB IDE and C18 C compiler were the choice as programming development environment. A project called PICos18 developed by Pragmatic Corporation has been chosen as a typical example for understanding RTOS. The first semester of final year will be dedicated for research and study about the programming practice of PIC18Fxxx and familiarity with MPLAB IDE under MPASM assembler and C18 compiler. At the end of the first semester, the operational concept of RTOS

on PIC18Fxxx device is to be demonstrated. The second semester is dedicated for codes development and programs testing so that at the end of the semester, the full functioning RTOS on PIC18Fxxx with other interfaced peripherals are to be presented.

CHAPTER 2
LITERATURE REVIEW

2.1. Operating System (OS)

An operating system is a program that controls the execution of application programs and acts as an interface between applications and the computer (or microprocessor/microcontroller system) hardware. It can be thought of having **3** objectives:

- **Convenience**: An operating system makes a computer more convenient to use.
- **Efficiency**: An operating system allows the computer system resources to be used in an efficient manner.
- **Ability to evolve**: An operating system should be constructed in such a way as to permit the effective development, testing, and introduction of new functions without interfering with service [4].

2.2. RTOS Concept

Real time operating systems are operating systems specially made to be used in time-critical environment where data must be processed extremely quickly [4].

An RTOS facilitates the creation of a real-time system, but does not guarantee the final result will be real-time; this requires correct development of the software. An RTOS does not necessarily have high throughput; rather, an RTOS provides facilities which, if used properly, guarantee deadlines can be met generally or deterministically (known as soft or hard real-time, respectively). An RTOS will typically use **specialized scheduling algorithms** in order to provide the real-time developer with the tools necessary to produce **deterministic behavior** in the final system. An RTOS is valued more for how quickly and/or predictably it can

respond to a particular event than for the amount of work it can perform over a given period of time. Key factors in an RTOS are therefore **minimal interrupt latency** and a **minimal task switching latency** [12].

2.3. Multitasking Environment

Multitasking is the processing of multiple tasks in a way that they are seemingly executed simultaneously on the same microcontroller CPU. This is achieved by sharing the time of CPU so that it executes one task per CPU time and switching the processing to cover all the tasks.

2.4. RTOS Kernel

RTOS kernel is the lowest-level and the core of the software layer which adapts the microcontroller to the real-time and multitasking processing environment. The functions provided by the kernel can be further divided in broader terms as follows:

1. The ability to switch from one task to another based on interrupt or software driven events (i.e. Task Switcher). This is the core of multitasking.
2. Usually provides some way of determining which tasks should be running based on priority (i.e. Task Scheduler)
3. Provides other services for the convenience of development (e.g. timing and alarming functions to entertain RTOS with delays and time management facilities (i.e. Alarm manager), events-based functions: SetEvent and WaitEvent functions (i.e. event manager), and internal communication between tasks (i.e. message manager).

In the following subsection, some the main kernel services will be discussed [13].

2.4.1. Task switcher

Task switcher is a part of kernel code which provides the RTOS with the mechanism to switch the execution of tasks on interrupt bases. Task switcher is the sole of kernel to achieve the multi-tasking processing.

2.4.2. Task scheduler

The Task Scheduler controls the execution of tasks, and can make them run in a very timely and responsive fashion based on their priorities and readiness.

Most RTOSs do their scheduling of tasks using a scheme called "priority-based preemptive scheduling." Each task in a software application must be assigned a priority, with higher priority values representing the need for quicker responsiveness. Very quick responsiveness is made possible by the "preemptive" nature of the task scheduling. "Preemptive" means that the scheduler is allowed to stop any task at any point in its execution, if it determines that another task needs to run immediately.

The basic rule that governs priority-based preemptive scheduling is that at every moment in time, "The Highest Priority Task that is ready to run will be the Task that must be running." In other words, if both a low-priority task and a higher-priority task are ready to run, the scheduler will allow the higher-priority task to run first. The low-priority task will only get to run after the higher-priority task has finished with its current work.

2.4.3. Other RTOS services manager

For the RTOS to be more convenient for complex tasks development, other timing and communication services have to be available. For this reason, some RTOSes have events, alarms, and tasks' communication functions served by their kernels. However, implementing this feature adds overhead processing and increase kernel interrupt latency time. Moreover, these services increase the RAM usage by RTOS. So, it is not always optimal to have them.

2.5. RTOS Scheduling Algorithms

As mentioned before, the "real-time" term in the acronym "RTOS" indicates the essential role of time in these particular systems. Typically, one or more physical devices external to the microcontroller generate stimuli, and the microcontroller must react appropriately to them within a **fixed short amount of time** [5].

In more broad details, Real-time systems can generally be categorized as **hard** real time, meaning there are absolute deadlines that must be met, or else, and **soft** real time, meaning that missing an occasional deadline is tolerable. In both cases, real-time behaviour is achieved by dividing the program into a number of processes, each of whose behaviour is predictable and known in advance. These processes are supposingly short lived and can run to completion in under a second. When an external event is detected, it is the job of the scheduler to schedule the processes in such a way as that all deadlines are met.

The events that a real-time system may have to respond to can be further categorized as periodic (occurring at regular intervals) or aperiodic (occurring unpredictably). A system may have to respond to multiple periodic event steams. Depending on how much time each event requires for processing, it may not even be possible to handle them all. For example, if there are m periodic events and event i occurs with period Pi and requires Ci seconds of CPU time to handle each event, then the load can only be handled if:

$$\sum_{i=1}^{m} \frac{C_i}{P_i} \leq 1$$

A real-time system that meets this criterion is said to be schedulable.

Additionally, based on the time when scheduling decisions are taken, scheduling algorithms can be further divided in two categories: dynamic and static. The former makes its scheduling decisions at run time; the latter makes them before the system starts running.

This section will consider a few common dynamic real-time scheduling algorithms. The classic algorithm is the rate monotonic algorithm (Liu and Layland, 1973). In advance, it assigns to each process a priority proportional to

the frequency of occurrence of its triggering event. For example, a process to run every 20 msec gets priority 50 and a process to run every 100 msec gets priority 10. At run time, the scheduler always runs the highest priority ready process, pre-empting the running process if need be. Liu and Layland proved that this algorithm is optimal.

Another popular real-time scheduling algorithm is earliest deadline first. Whenever an event is detected, its process is added to the list of ready processes. The list is kept sorted by deadline, which for a periodic event is the next occurrence of the event. The algorithm runs the first process on the list, the one with the closest deadline.

A third algorithm first computes for each process the amount of time it has to spare, called its laxity. If a process requires 200 ms and must be finished in 250 millisecond, its laxity is 50 msec. The algorithm, called least laxity, chooses the process with the smallest amount of time to spare [5].

While in theory it is possible to turn a general-purpose operating system into a real-time system by using one of these scheduling algorithms, in practice the context-switching overhead of general-purpose systems is so large that real-time performance can only be achieved for applications with easy time constraints. As a consequence, most real-time work uses special real-time operating systems that have certain important properties. Typically these include a small size, fast interrupt time, rapid context switch, and short interval during which interrupts are disabled, and the ability to manage multiple timers in the millisecond or microsecond range [5].

2.6. PIC18Fxxx microcontroller System

Microcontroller is a small computer system on a single chip consisting of a relatively simple CPU combined with support functions such as interrupts, timers, watchdog timer, serial and analog I/O etc.

PIC18Fxxx are high performance microcontrollers built with enhanced flash memory technology with 16 bits instruction word length and can run at 40MHz oscillator frequency.

PIC18Fxxx microcontrollers have several devices (e.g. 18F452, 18F458, 18F4550) each has its own special features (CAN module, USB module, etc...) but the set of instructions used are still the same.

In this project, the general purpose microcontroller 18F452 and 18F458 (microcontroller with CAN module) will be used.

The following figure shows the pins alignment of 18F458 microcontroller:

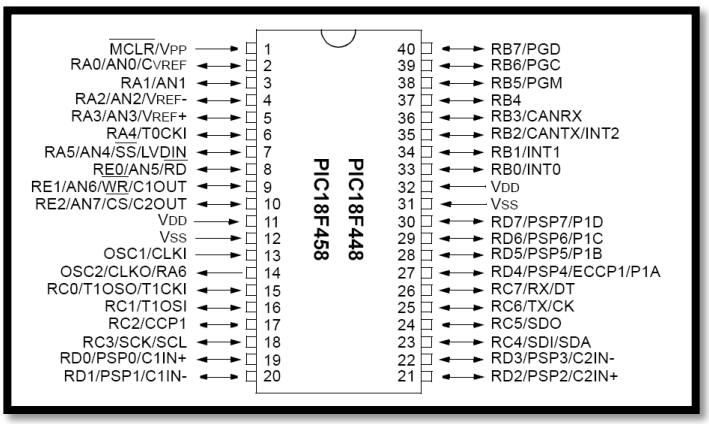


Figure 1: PIC18F458 microcontroller pins alignment [3]

This microcontroller is characterized by several features which can be summarized in the following table:

Table 1: PIC18Fxx8 devices' features [3]

Features		PIC18F248	PIC18F258	PIC18F448	PIC18F458
Operating Frequency		DC – 40 MHz	DC – 40 MHz	DC – 40 MHz	DC – 40 MHz
Internal Program Memory	Bytes	16K	32K	16K	32K
	# of Single-Word Instructions	8192	16384	8192	16384
Data Memory (Bytes)		768	1536	768	1536
Data EEPROM Memory (Bytes)		256	256	256	256
Interrupt Sources		17	17	21	21
I/O Ports		Ports A, B, C	Ports A, B, C	Ports A, B, C, D, E	Ports A, B, C, D, E
Timers		4	4	4	4
Capture/Compare/PWM Modules		1	1	1	1
Enhanced Capture/Compare/ PWM Modules		—	—	1	1
Serial Communications		MSSP, CAN, Addressable USART	MSSP, CAN, Addressable USART	MSSP, CAN, Addressable USART	MSSP, CAN, Addressable USART
Parallel Communications (PSP)		No	No	Yes	Yes
10-bit Analog-to-Digital Converter		5 input channels	5 input channels	8 input channels	8 input channels
Analog Comparators		No	No	2	2
Analog Comparators VREF Output		N/A	N/A	Yes	Yes
Resets (and Delays)		POR, BOR, RESET Instruction, Stack Full, Stack Underflow (PWRT, OST)	POR, BOR, RESET Instruction, Stack Full, Stack Underflow (PWRT, OST)	POR, BOR, RESET Instruction, Stack Full, Stack Underflow (PWRT, OST)	POR, BOR, RESET Instruction, Stack Full, Stack Underflow (PWRT, OST)
Programmable Low-Voltage Detect		Yes	Yes	Yes	Yes
Programmable Brown-out Reset		Yes	Yes	Yes	Yes
CAN Module		Yes	Yes	Yes	Yes
In-Circuit Serial Programming™ (ICSP™)		Yes	Yes	Yes	Yes
Instruction Set		75 Instructions	75 Instructions	75 Instructions	75 Instructions
Packages		28-pin SPDIP 28-pin SOIC	28-pin SPDIP 28-pin SOIC	40-pin PDIP 44-pin PLCC 44-pin TQFP	40-pin PDIP 44-pin PLCC 44-pin TQFP

18F458 and 18F452 microcontrollers have relatively large data and program memories. This is shown in the following table:

Table 2: Program and data memory for PIC18F458 [3]

Device	Program Memory		Data Memory		I/O	10-bit A/D (ch)	Comparators	CCP/ ECCP (PWM)	MSSP		USART	Timers 8/16-bit
	Flash (bytes)	# Single-Word Instructions	SRAM (bytes)	EEPROM (bytes)					SPI™	Master I^2C™		
PIC18F248	16K	8192	768	256	22	5	—	1/0	Y	Y	Y	1/3
PIC18F258	32K	16384	1536	256	22	5	—	1/0	Y	Y	Y	1/3
PIC18F448	16K	8192	768	256	33	8	2	1/1	Y	Y	Y	1/3
PIC18F458	32K	16384	1536	256	33	8	2	1/1	Y	Y	Y	1/3

In the following sections, more details about PIC18F devices will be discussed.

2.6.1. PIC18F oscillator

The device system clock is required for the device to execute instructions and for the peripherals to function. Four device system clock periods (T_{SCLK}) generate one internal instruction clock cycle (T_{CY}).

The device system clock (T_{SCLK}) is derived from an external system clock. This external system clock can be generated in one of eight different oscillator modes. The device configuration bits select the oscillator mode. Device configuration bits are non-volatile memory locations and the operating mode is determined by the value written during device programming.

The oscillator modes are:

- EC : External Clock
- ECIO : External Clock with I/O pin enabled
- LP : Low Frequency (Power) Crystal XT Crystal/Resonator HS High Speed Crystal/Resonator
- RC : External Resistor/Capacitor
- RCIO : External Resistor/Capacitor with I/O pin enabled
- HS4 : High Speed Crystal/Resonator with 4x frequency PLL multiplier enabled, figure 2.2 shows device clock source schematic

Multiple oscillator circuits can be implemented on an Enhanced Architecture device. There is the default oscillator (OSC1), and additional oscillators may be available, such as the Timer1 oscillator.

Software may allow these auxiliary oscillators to be switched in as the device oscillator. The Timer1 oscillator is a low frequency (low power) oscillator that is designed to be operated at 32 kHz. Figure2-1 shows a block diagram of the oscillator options. The output signal of the Timer1 oscillator circuitry is a low frequency (power) clock source (T_{T1P}).

The source for the device system clock can be switched from the default clock (T_{SCLK}) to the 32 kHz-clock low power clock source (T_{T1P}) under software control. Switching to the 32kHz low frequency (power) clock source from any of the eight default clock sources may allow power saving.

These oscillator options are made available to allow a single device type the flexibility to fit applications with different oscillator requirements. The RC oscillator option saves system cost, while the LP crystal option saves power. The HS4 option allows frequency of incoming crystal oscillator signal to be multiplied by four for higher internal clock frequency. This is useful for customers who are concerned with EMI due to high frequency crystals. The device configuration bits are used to select these various options.

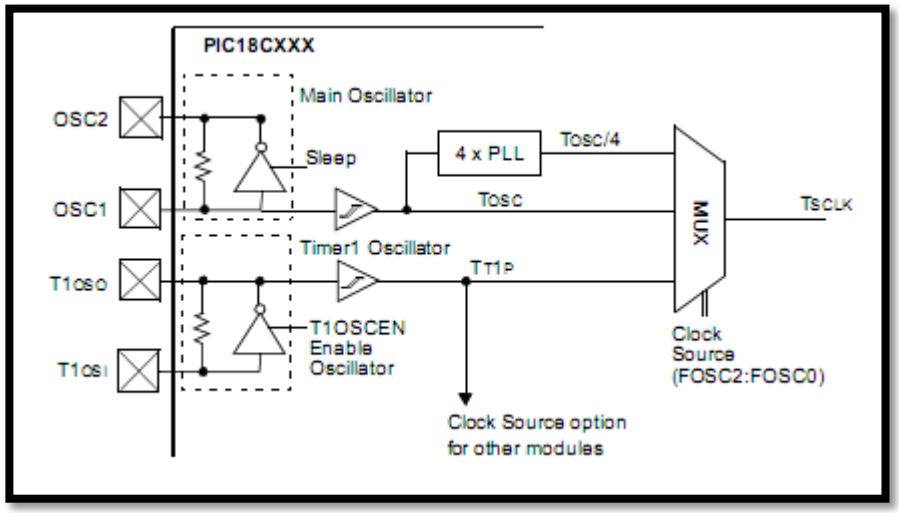


Figure 2: Device Clock source

2.6.2. PIC18F System Architecture

The high performance of the PIC18CXXX devices can be attributed to a number of architectural features commonly found in RISC microprocessors. These include:

- Harvard architecture
- Long Word Instructions
- Single Word Instructions
- Single Cycle Instructions
- Instruction Pipelining
- Reduced Instruction Set
- Register File Architecture
- Orthogonal (Symmetric) Instructions. Figure 4.3 shows a general block diagram for PIC18CXXX devices.

a) Harvard Architecture:

Harvard architecture has the program memory and data memory as separate memories which are accessed from separate buses. This improves bandwidth over

traditional von Neumann architecture in which program and data are fetched from the same memory using the same bus.

To execute an instruction, a von Neumann machine must make one or more (generally more) accesses across the 8-bit bus to fetch the instruction. Then data may need to be fetched, operated on and possibly written. As can be seen from this description, the bus can become extremely congested. In Harvard architecture, the instructions fetched in a single instruction cycle (all 16 bits). While the program memory is being accessed, the data memory is on an independent bus and can be read and written. These separated busses allow one instruction to execute, while the next instruction is fetched. A comparison of Harvard and von Neumann architectures is shown in figure below.

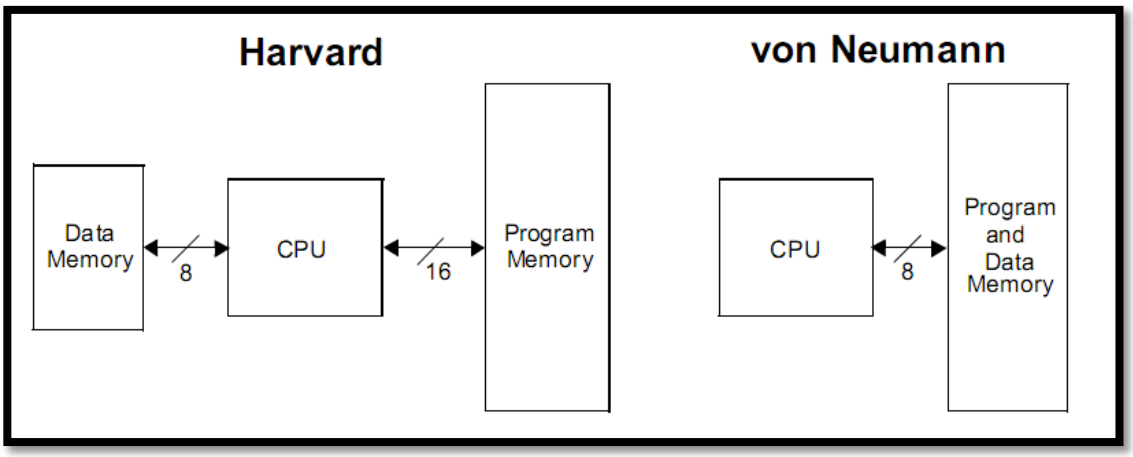


Figure 3: Comparison of Harvard and von Neumann architectures

b) **Long Word Instructions:**

Long word instructions have a wider (more bits) instruction bus than the 8-bit data memory bus. This is possible because the two buses are separate. This allows instructions to be sized differently than the 8-bit wide data word and allows a more efficient use of the program memory, since the program memory width is optimized to the architectural requirements.

c) **Single Word Instructions:**

Single word instruction op-codes are 16-bits wide making it possible to have all but a few instructions be single word instructions. A 16-bit wide program

memory access bus fetches a 16-bit instruction in a single cycle. With single word instructions, the number of words of program memory locations equals the number of instructions for the device. This means that all locations are valid instructions. Typically in the von Neumann architecture, most instructions are multi-byte. In general, a device with 4 Kbytes of program memory would allow approximately 2K of instructions. This 2:1 ratio is generalized and dependent on the application code. Since each instruction may take multiple bytes, there is no assurance that each location is a valid instruction.

d) **Double Word Instructions:**

Some operations require more information than what can be stored in the16 bits of a program memory location. These operations require a double word instruction, and are therefore 32-bits wide. Instructions that require this second instruction word are:

- Memory to memory move instruction (12 bits for each RAM address) - MOVFF SourceReg, DestReg
- Literal value to FSR move instruction (12 bits for data and 2 bits for FSR to load) - LFSR FSR#, Address
- Call and goto operations (20 bits for address)
    - CALL Address
    - GOTO Address

The first word indicates to the CPU that the next program memory location is the additional information for this instruction and not an instruction. If the CPU tries to execute the second word of an instruction (due to a software modified PC pointing to that location as an instruction), the fetched data is executed as a NOP. Double word instruction execution is not split between the two T_{CY} cycles by an interrupt request.

That is, when an interrupt request occurs during the execution of a double word instruction, the execution of the instruction is completed before the processor vectors to the interrupt address. The interrupt latency is preserved.

e) **Instruction Pipeline:**

The instruction pipeline is a two-stage pipeline that overlaps the fetch and execution of instructions. The fetch of the instruction takes one T_{CY}, while the

execution takes another T_{CY}. However, due to the overlap of the fetch of current instruction and execution of previous instruction, an instruction is fetched and another instruction is executed every T_{CY}.

f) **Single Cycle Instructions:**

With the program memory bus being 16-bits wide, the entire instruction is fetched in a single machine cycle (TCY), except for double word instructions. The instruction contains all the information required and is executed in a single cycle. There may be a one cycle delay in execution if the result of the instruction modified the contents of the program counter. This requires the pipeline to be flushed and a new instruction to be fetched.

g) **Two Cycle Instructions:**

Double word instructions require two cycles to execute, since all the required information is in the 32 bits.

h) **Reduced Instruction Set:**

When an instruction set is well designed and highly orthogonal (symmetric), fewer instructions are required to perform all needed tasks. With fewer instructions, the whole set can be more rapidly learned.

i) **Register File Architecture:**

The register files/data memory can be directly or indirectly addressed. All special function registers, including the program counter, are mapped in the data memory.

j) **Orthogonal (Symmetric) Instructions:**

Orthogonal instructions make it possible to carry out any operation on any register using any addressing mode. This symmetrical nature and lack of "special instructions" make programming simple yet efficient. In addition, the learning curve is reduced significantly. The Enhanced MCU instruction set uses only three non-register oriented instructions, which are used for two of the cores features. One is the SLEEP instruction, which places the device into the lowest power use mode. The second is the CLRWDT instruction, which verifies the chip is operating properly by preventing the on-chip Watchdog Timer (WDT) from

overflowing and resetting the device. The third is the RESET instruction, which resets the device.

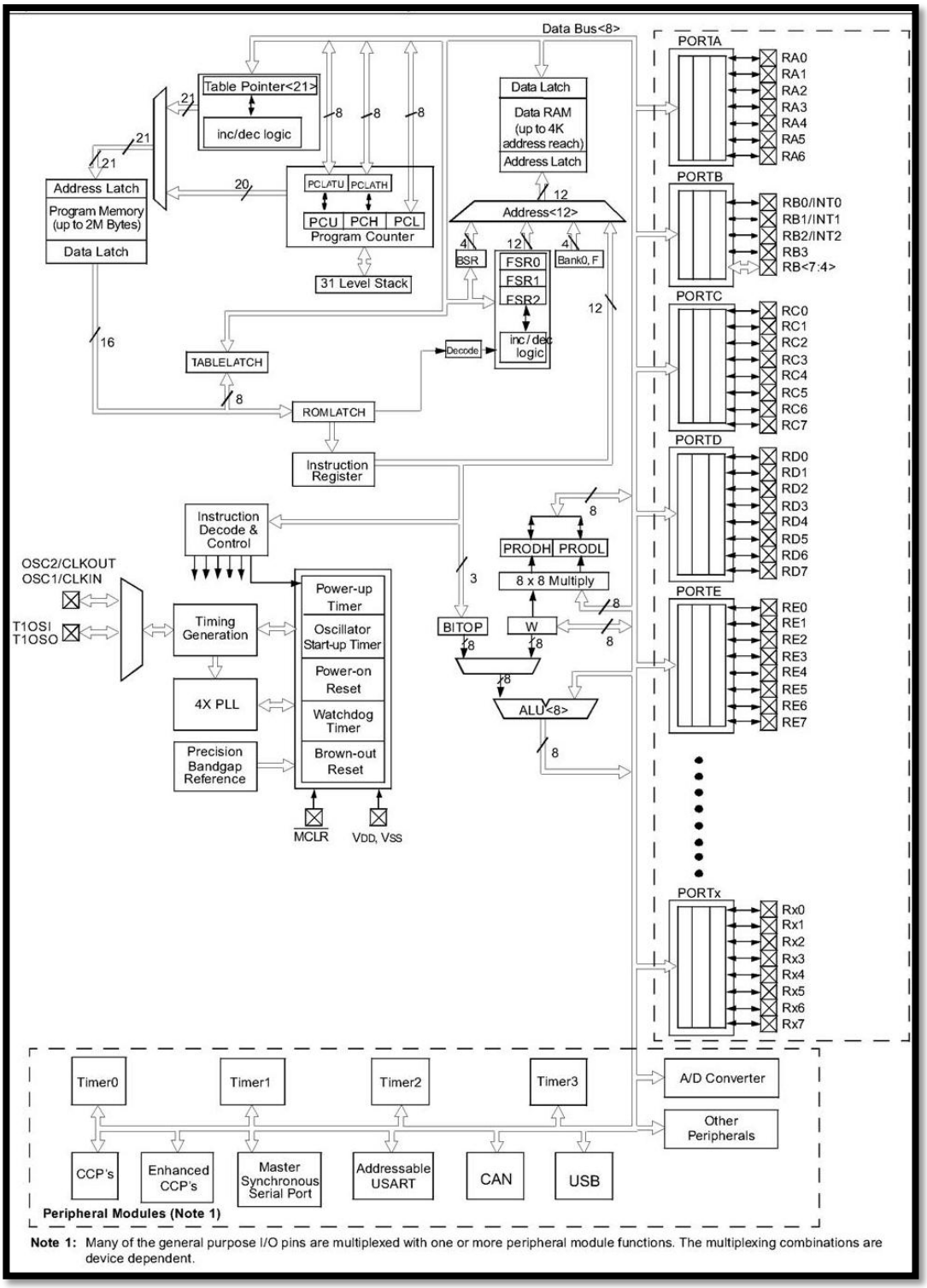


Figure 4: General Enhanced Microcontroller block diagram

2.6.3. CPU and ALU of PIC18

The Central Processing Unit (CPU) is responsible for using the information in the program memory (instructions) to control the operation of the device. Many of these instructions operate on data memory. To operate on data memory, the Arithmetic Logical Unit (ALU) is required. In addition to performing arithmetical and logical operations, the ALU controls the state of the status bits, which are found in the STATUS register. The result of some instructions forces status bits to a value depending on the state of the result.

a) CPU

The CPU can be thought of as the "brains" of the device. It is responsible for fetching the correct instruction for execution, decoding that instruction and then executing that instruction. The CPU sometimes works in conjunction with the ALU to complete the execution of the instruction (in arithmetic and logical operations). The CPU controls the program memory address bus, the data memory address bus and accesses to the stack.

b) ALU

18Fxxx devices contain an 8-bit ALU and an 8-bit working register (WREG). The ALU is a general purpose arithmetic and logical unit. It performs arithmetic and Boolean functions between the data in the working register and any register file. The WREG register is directly addressable and in the SFR memory map.

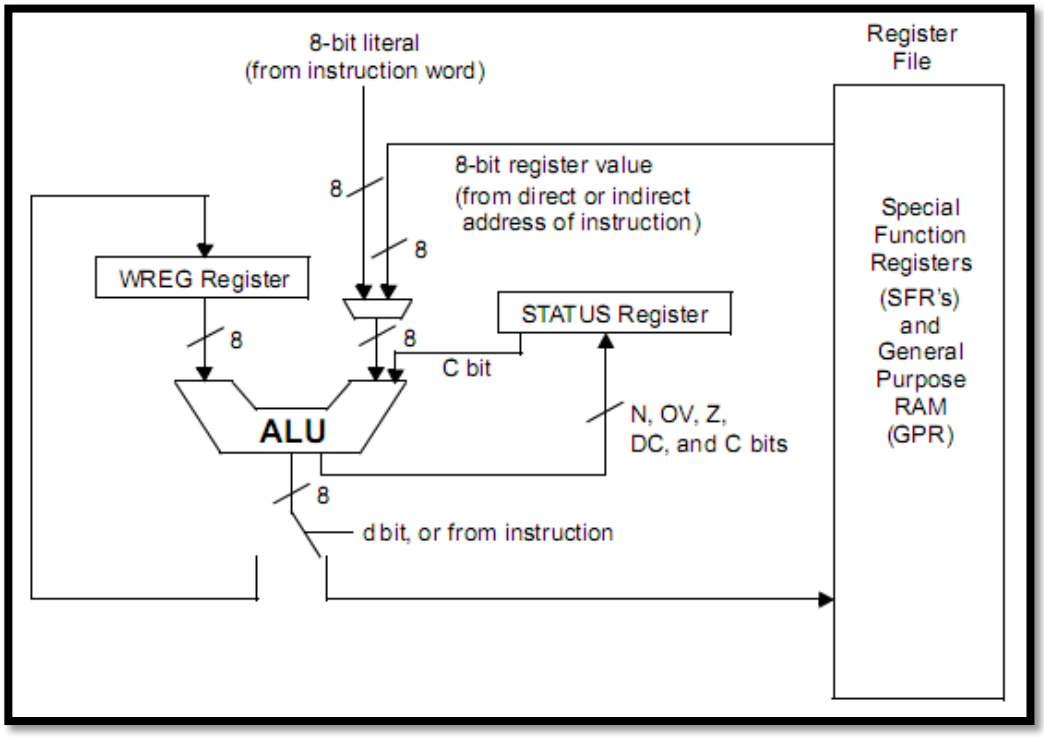
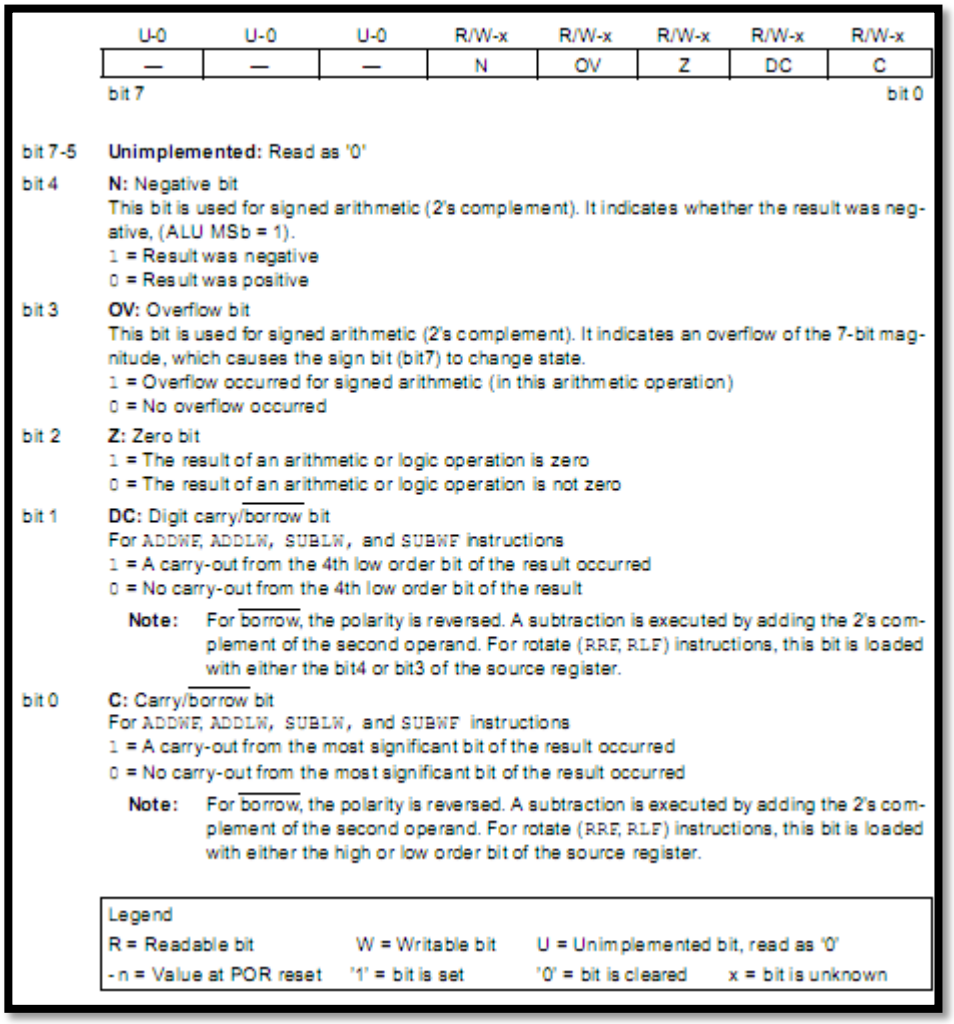


Figure 5: Operation of ALU and W register

c) STATUS Register

The STATUS register, shown in figure below, contains the arithmetic status of the ALU. The STATUS register can be the destination for any instruction, as with any other register. If the STATUS register is the destination for an instruction that affects the Z, DC, C, OV or N bits, then the write to these five bits is disabled. These bits are set or cleared according to the device logic. Therefore, the result of an instruction with the STATUS register as destination may be different than intended. For example, CLRF STATUS will clear the upper three bits and set the Z bit. This leaves the STATUS register as 000u u1uu (where u= unchanged). It is recommended, therefore, that only BCF, BSF, SWAPF, MOVFF, and MOVWF instructions are used to alter the STATUS register, because these instructions do not affect the Z, C, DC, OV or N bits of the STATUS register.



U-0	U-0	U-0	R/W-x	R/W-x	R/W-x	R/W-x	R/W-x
—	—	—	N	OV	Z	DC	C
bit 7							bit 0

bit 7-5 **Unimplemented:** Read as '0'

bit 4 **N:** Negative bit
This bit is used for signed arithmetic (2's complement). It indicates whether the result was negative, (ALU MSb = 1).
1 = Result was negative
0 = Result was positive

bit 3 **OV:** Overflow bit
This bit is used for signed arithmetic (2's complement). It indicates an overflow of the 7-bit magnitude, which causes the sign bit (bit7) to change state.
1 = Overflow occurred for signed arithmetic (in this arithmetic operation)
0 = No overflow occurred

bit 2 **Z:** Zero bit
1 = The result of an arithmetic or logic operation is zero
0 = The result of an arithmetic or logic operation is not zero

bit 1 **DC:** Digit carry/$\overline{\text{borrow}}$ bit
For ADDWF, ADDLW, SUBLW, and SUBWF instructions
1 = A carry-out from the 4th low order bit of the result occurred
0 = No carry-out from the 4th low order bit of the result

Note: For $\overline{\text{borrow}}$, the polarity is reversed. A subtraction is executed by adding the 2's complement of the second operand. For rotate (RRF, RLF) instructions, this bit is loaded with either the bit4 or bit3 of the source register.

bit 0 **C:** Carry/$\overline{\text{borrow}}$ bit
For ADDWF, ADDLW, SUBLW, and SUBWF instructions
1 = A carry-out from the most significant bit of the result occurred
0 = No carry-out from the most significant bit of the result occurred

Note: For $\overline{\text{borrow}}$, the polarity is reversed. A subtraction is executed by adding the 2's complement of the second operand. For rotate (RRF, RLF) instructions, this bit is loaded with either the high or low order bit of the source register.

Legend			
R = Readable bit	W = Writable bit	U = Unimplemented bit, read as '0'	
-n = Value at POR reset	'1' = bit is set	'0' = bit is cleared	x = bit is unknown

Figure 6: Status Register

2.6.4. Memories of PIC18

There are two memory blocks in the memory map; program memory and data memory. Each block has its own bus, so that access to each block can occur during the same instruction cycle.

The data memory can further be broken down into General Purpose RAM and the Special Function Registers (SFRs). The SFRs used to control the peripheral modules in the microcontroller. In addition, there are other registers used that are neither part of the program nor data memory spaces.

These registers are not directly addressable and include:

- Return address stack
- Fast return stack

2.6.5. Interrupts

In PIC18 devices, interrupts can be generated from many sources such as timers, A/D conversion, USART receive/transmit etc…. Interrupts can be also prioritized as high or low level interrupt. There are several SFRs which control interrupts (e.g. INTCON, IPR, PIE, etc…)

2.6.6. Input /Output ports

General purpose I/O pins can be considered the simplest of peripherals. They allow the PICmicro to monitor and control other devices. To add flexibility and functionality to a device, some pins are multiplexed with an alternate function(s). These functions depend on which peripheral features are on the device. In general, when a peripheral is functioning, that pin may not be used as a general purpose I/O pin.

For most ports, the I/O pin's direction (input or output) is controlled by the data direction register, called the TRIS register. TRIS<x> controls the direction of PORT<x>. A '1' in the TRIS bit corresponds to that pin being an input, while a '0' corresponds to that pin being an output. An easy way to remember is that a '1' looks like an I (input) and a '0' looks like an O (output).

The PORT register is the latch for the data to be output. When the PORT is read, the device reads the levels present on the I/O pins (not the latch). This means that care should be taken with read-modify-write commands on the ports and changing the direction of a pin from an input to an output.

The following figure shows a schematic of a typical I/O port.

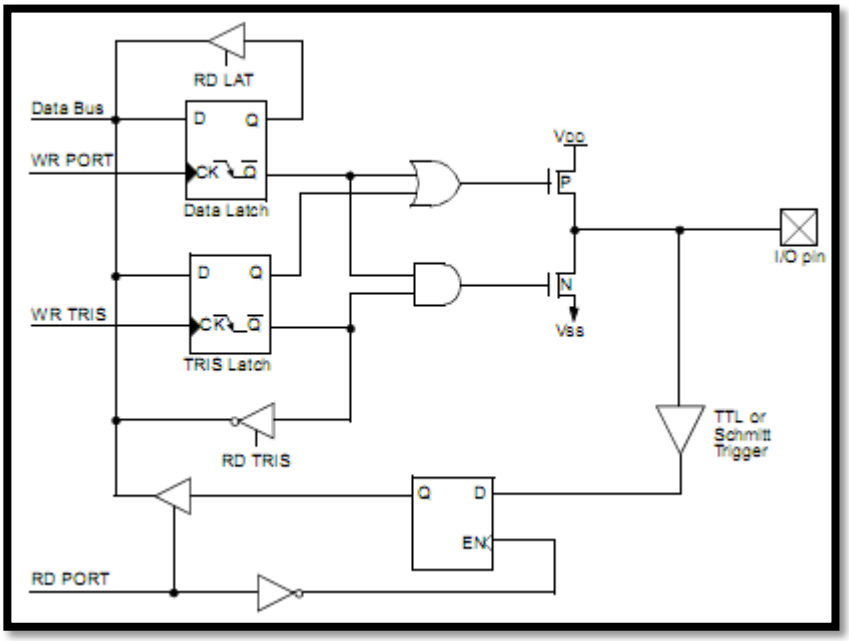


Figure 7: Typical I/O port

For more details about the modules of microcontrollers and related specs, please refer to:

PICmicro® 18C MCU Family Reference Manual

2.7. MPLAB C18 Compiler language suite

The MPLAB C18 compiler is a free-standing, optimizing ANSI C compiler for the PIC18 PICmicro microcontrollers (MCU).This compiler is fully compatible with Microchip's MPLAB IDE and MPLAB SIM simulator.

The MPLAB C18 compiler has the following features:

- Generation of relocatable object modules for enhanced code reuse.
- Compatibility with object modules generated by the MPASM assembler, allowing complete freedom in mixing assembly and C programming in a single project.
- Strong support for inline assembly when total control is absolutely necessary.

- Extensive library support, including PWM, SPI™, I2C™, UART, USART, string manipulation and math libraries.
- Full user-level control over data and code memory allocation.

In this project, The focus will be made for the following points:

i) C18 compiler managed resources:

   C18 uses some registers to do some of the intermediate and temporary operations. These registers are: FSR0, FSR1, FSR2, PRODH, PRODL, TABLAT, TBLPTRU, TBLPTRH and TBLPTRL

ii) Startup and Initialization

   C18 compiler generates a C function to initialize all the variables used in the main function. This function is directly called after reset.

   During the initialization, a software stack (used and managed by C18 compiler) is setup and initialized. FSR1 and FSR2 are used to manage the software stack.

iii) Software Stack:

   This is basically a memory section defined in the linker script file (e.g. 18f452.lkr). C18 uses this section to store arguments, return values and local variables of functions when they are called.

   The default size of this stack is 256 bytes. However this can be modified from the linker script by changing the following linker command:

```
STACK   SIZE = 0x100
```

   to the desired values.

CHAPTER 3
METHODOLOGY

3.1. Procedure Identification

The scope of this project and flow of tasks are envisaged to be carried out as shown in the following chart:

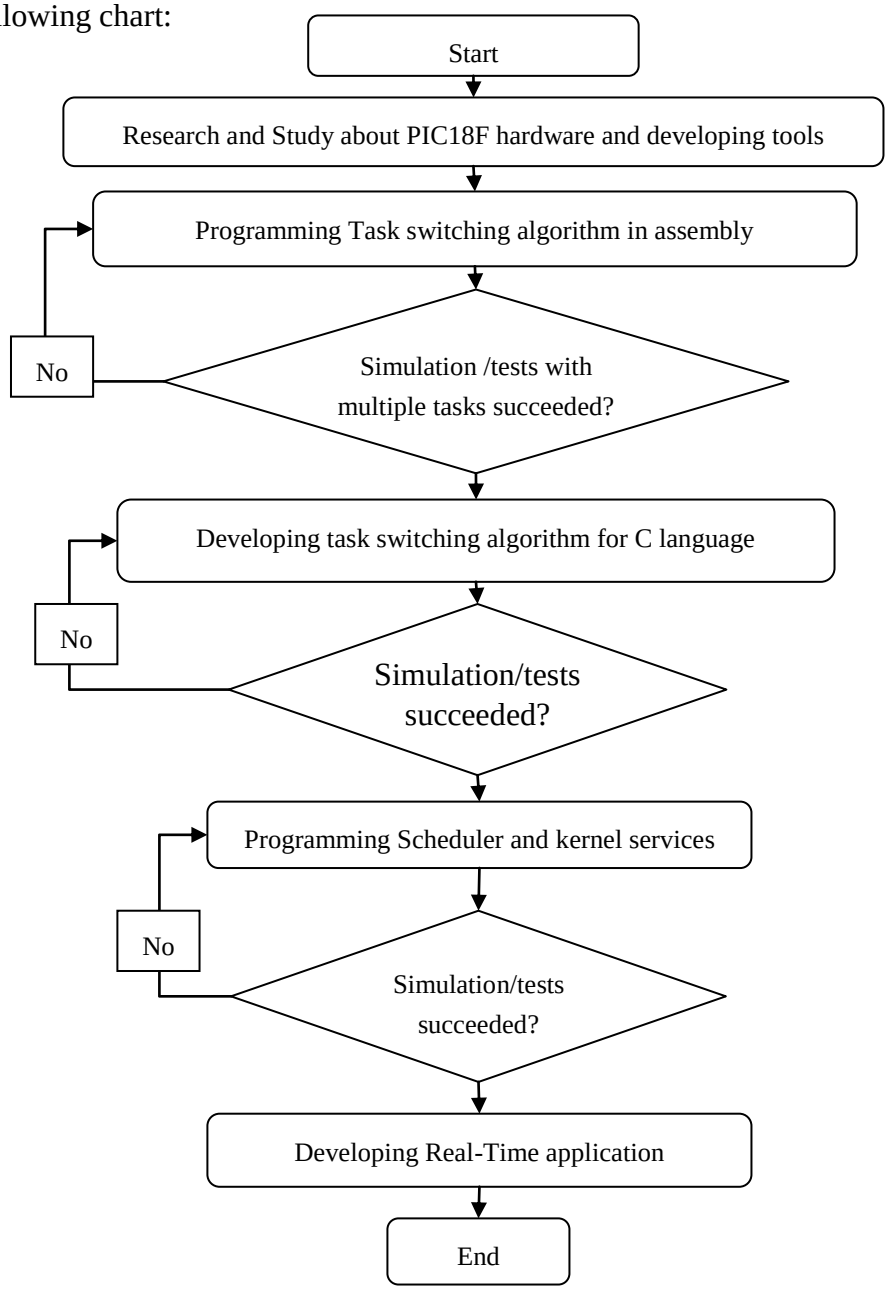


Figure 8: General Flow chart of Project work

The overall flow of the project can be divided into the following milestones:

3.1.1. PIC18Fxxx system Study

A thorough study is carried out to attain the student the basic and primitive knowledge about the system which is intended to be programmed. This part of the project is estimated to have 4-5 weeks of the first semester. A good familiarization with PIC18F instruction set, special function registers (SFR), Memory organization, device settings, peripherals' operation and hardware circuitry has to be gained.

3.1.2. Study of RTOS concepts

Operating systems have several concepts. In this project, an overall study of operating system is to be done. This study will also cover the concept of: operating system concept, RTOS concept, kernel, and scheduling algorithms. Meanwhile, the RTOS named "**PICos18"** is to be simulated and investigated throughout this stage.

3.1.3. Running PICos18 on PIC18Fxxx device

After having little soft background about RTOS, more practical interaction with typical RTOS system is to be made. To achieve this goal, PICos18 is simulated and ported to PIC18 microcontroller and then its performance is further investigated. At this stage, the student is in favour of monitoring, examining and evaluating the performance of this RTOS. This gives the student a good sense of how RTOS behaves. The main challenge at this stage would probably be the adaption of PICos18 to microcontroller settings and how applications and tasks are created and made in PICos18 system.

3.1.4. Development of RTOS

At this stage, the coding, algorithms and the hardware interface of the system is to be started. At the very starting of this stage, simple codes to setup timers and interrupts will be developed in assembly environment. This is because assembly programming is straight forward to microcontroller hardware in addition to its light size comparing to C generated code. Timer0 with its associated interrupt in the microcontroller will be utilized as kernel timer.

When kernel timer interrupting facility is available, task switching algorithm will be developed. For instance, tasks will be written in assembly language and then the assembly version for task switcher will be tested. As soon as the assembly version runs successfully, a C version of the task switching will be created.

After the work with switching mechanism is successfully done, development of timing based functions (e.g. delayMs), scheduling algorithms, and events based routines are to be made.

3.1.5. RTOS testing and troubleshooting

Finally, the project will be concluded by demonstrating its operability on a practical application which is based on the developed RTOS. A suitable application will be selected and designed at the end of this project.

3.2. Equipment and Tools

To implement the RTOS on PIC18Fxxx devices, some equipments and software are utilized. These tools and equipment include:

3.2.1. PIC18Fxxx starting kit

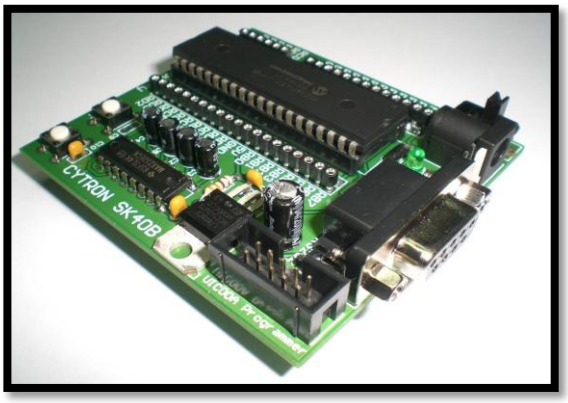


Figure 9: PIC18F developing kit [9]

This kit is a practical and suitable platform for developing codes and programs on 18F series devices. All the basic and necessary connections to microcontroller are built. This board is running on 20MHz oscillator frequency. The layout of the board and the hardware circuitry can be viewed in appendix A.

3.2.2. PICKit 2 programmer

The programmer is used to load the hex file (produced by compiler or assembler) into the PIC memory.

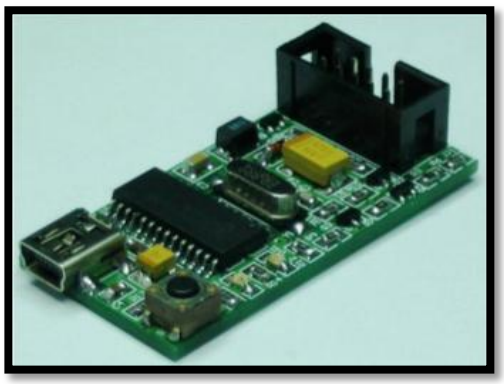


Figure 10: PIC18F programmer [10]

3.2.3. MPLAB IDE

MBLAB IDE version 8.33 is used to development programs and codes for PIC18F. It also contains a complete framework (includes simulator, programmer, debugger and compilers).

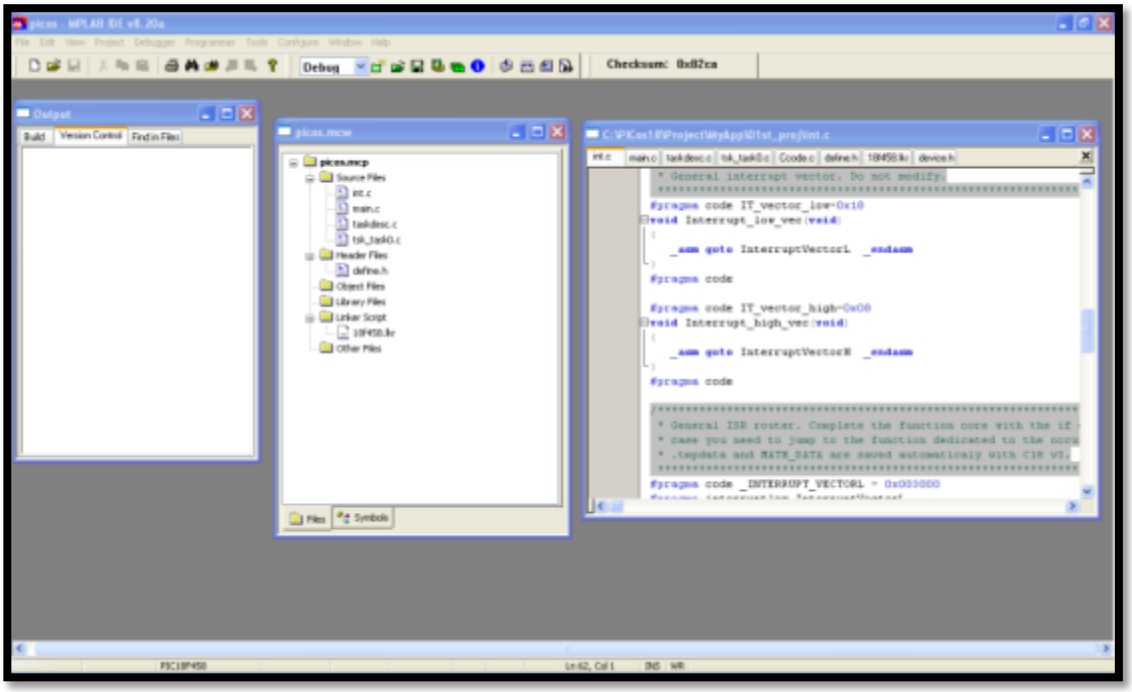


Figure 11: Snapshot for MPLAB IDE

3.2.4. MPLAB SIM Simulator

MPLAB SIM simulator is part of MPLAB IDE software. This software is used to simulate assembly and c codes. It offers good features to monitor the registers of the PIC and also monitoring the timing of instructions' execution.

3.2.5. PICkit 2 Debugger

This is a very useful and simple In-Circuit debugger. It connects to PICkit2 programmer. It gives good debugging facilities especially when peripherals are interfaced with microcontrollers. It only supports one break point at a time.

3.2.6. C18 C COMPILER

C18 C compiler is designed to work with PIC18 devices and to work under MPLAB IDE integrated environment.

By using this compiler, programmers may edit and developed application based on the high level C language. Moreover, some built-in function are available for fast development process for microcontroller hardware (e.g. CAN and I2C serial communication).

CHAPTER 4
RESULTS AND DISCUSSION

4.1 RESULTS

During the work process throughout this project, the following results are obtained:

- A multitasking kernel for "Taj RTOS" for assembly programming environment.
- C version of "Taj RTOS" multitasking kernel based on round-robin scheduling algorithm.
- Testing "Taj RTOS" with LCD, keypad, 7 segment display units and LEDs output based task.

4.1.1. Taj RTOS for Assembly programming environment

A simple multi-tasking kernel is developed to share microcontroller's CPU time on the tasks which are written in assembly. To achieve this, two assembly subroutines are created to form the kernel which are: RTOS Initializer and Task switcher. (please look at kernel code in appendix E)

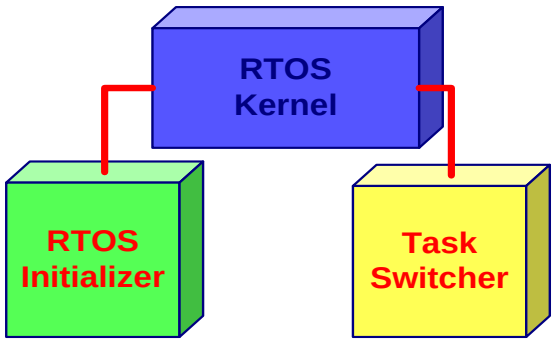


Figure 12: Taj RTOS Kernel for assembly environment

a) Design concept:

To make the multitasking processing for tasks possible, each task has to be given small time of CPU to execute its instructions. In this kernel, each task is given 1ms of CPU time to execute its instruction before it is swept and another task is loaded to the CPU. This timing is done by configuring TIMER0 to generate interrupt every 1ms. Whenever the interrupt occurs, all the important registers which tasks use are stored in RAM and it is retrieved when the task is restored again.

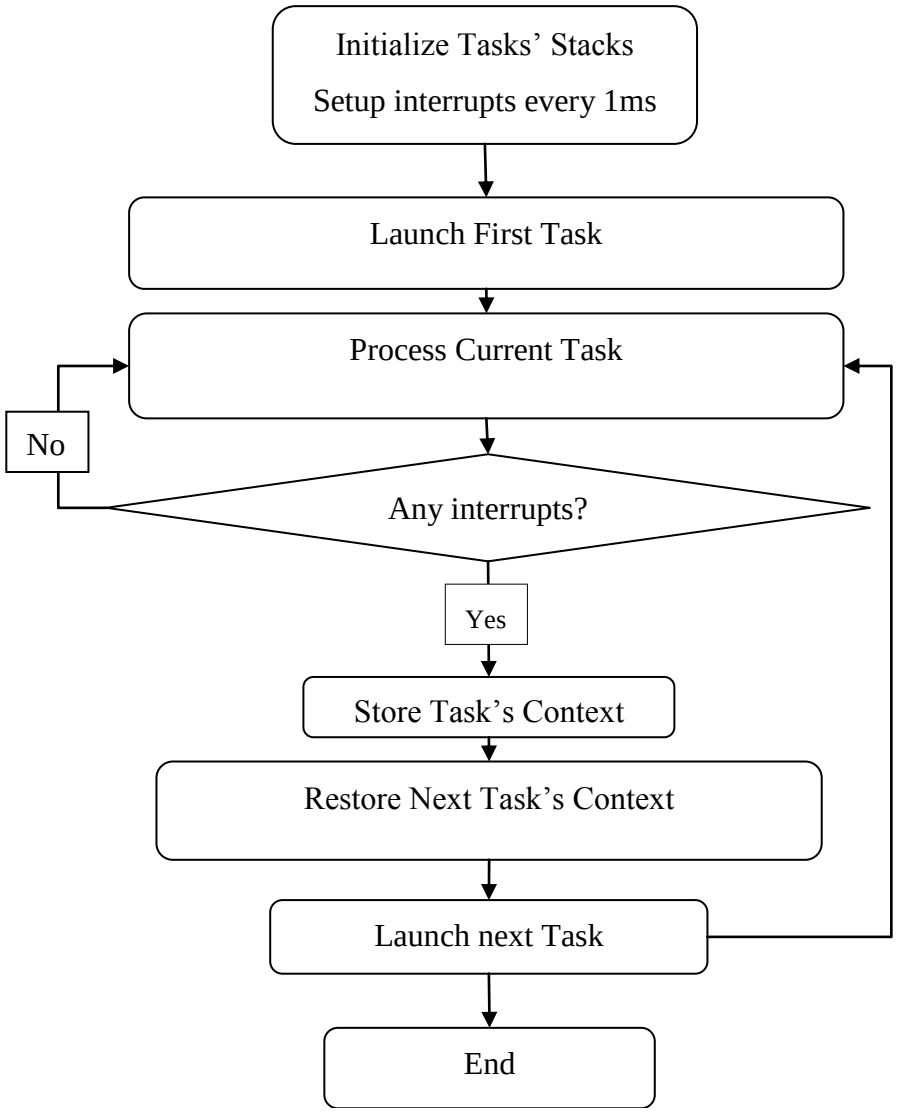


Figure 13: Flow chart of Taj RTOS kernel operation

b) Overview of RTOS code

RTOS code is written in 1 assembly file named “kernel.asm” where RTOS initializer and Task switcher reside. Aside from that, four header files are made to declare Tasks, Stacks and RTOS registers. RTOS code is available in Appendix D.

The following figure illustrates the codes and their functions for assembly version of Taj-RTOS.

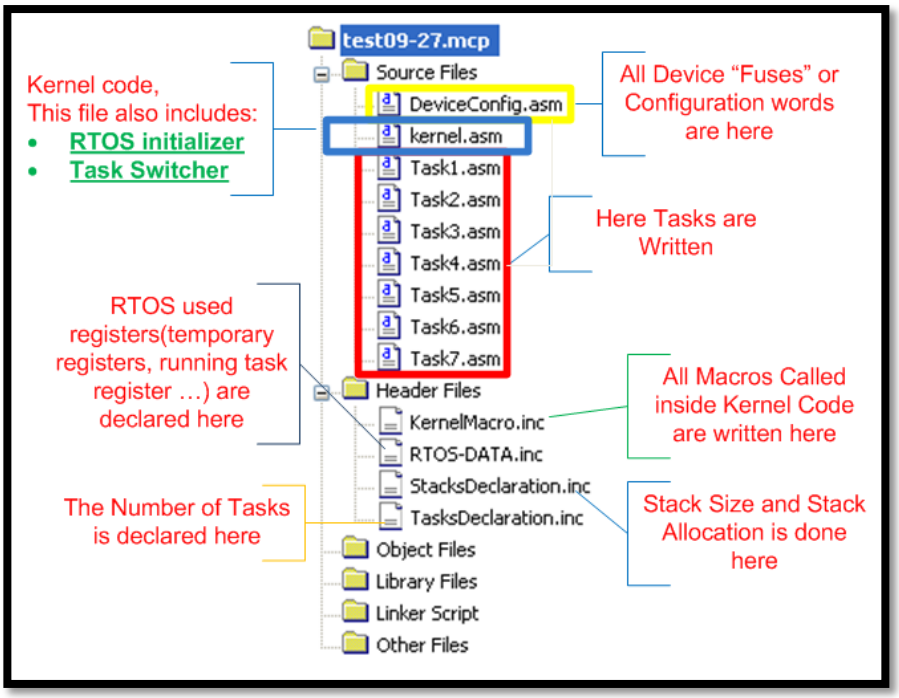


Figure 14: Taj RTOS codes (Assembly Version)

c) RTOS Initializer:

For the RTOS to function, some initializations have to be done before RTOS starts. These initializations are done in this part of the kernel. The typical initializations include:

i) Initializing TIMER0 to generate high priority periodic interrupt every 1 ms.

The following table shows Timer0 initialization.

Table 3: Timer0 intialization and the associated registers setting

Register	Value	Function
INTCON2 Register	0x84	Set Timer0 as high priority interrupt
IPEN bit in RCON register	1	Enable Priority interrupt
T0CON Register	0xc4	Setting 1:16 pre-scale for Timer0

ii) Creating Stack for each Task where the vulnerable registers are stored and retrieved.

For each task a stack is created to save the vulnerable registers. The size of the stack equals the size of registers which is classified as "vulnerable" for the tasks to run properly. Those registers are listed in the following table:

Table 4: Vulnerable registers

Register	Maximum Size
Stack pointer (STKPTR)	1
Hardware Stack **(TOSU - TOSH - TOSL) x 30 levels**	90
BSR	1
WREG	1
PROD (H-L)	2
FSR(0,1,2) (H-L)	6
STATUS	1
TABLE POINTER **TBLPTR (U-H-L)** **TABLAT**	4
MAXIMUM TOTAL SIZE	106

Since the hardware stack is not always used, the common practice is to save a portion of it (say 10 levels). This will stack size and the utilization of RAM and therefore reduce the switching time.

iii) Creating Array to store the addresses of the stack for easier reach.

To ease the access to the stacks, the addresses of the stacks have to be stored in one array. Each address has higher byte and lower byte. Therefore the size of the array is,

$$\boldsymbol{Array\ size = number\ of\ Tasks\ x\ 2}$$

Table 5: Stack Pointers Array for 3 Tasks

Index	Value	Index	Value
0	Stack1 Address High byte	1	Stack1 Address Low byte
2	Stack2 Address High byte	3	Stack2 Address Low byte
4	Stack3 Address High byte	5	Stack3 Address Low byte

d) Task switcher:

The function of this section is to switch between tasks whenever required. For the switching to be smooth and fine, all the vulnerable registers have to be stored and restored without any loss on their data. However, care has to be taken when dealing with sensitive registers such as PC and Status registers. Before the switching takes place, the kernel has to locate the next stack to retrieve data from and the current stack to save data to. The following figure illustrates the switching mechanism.

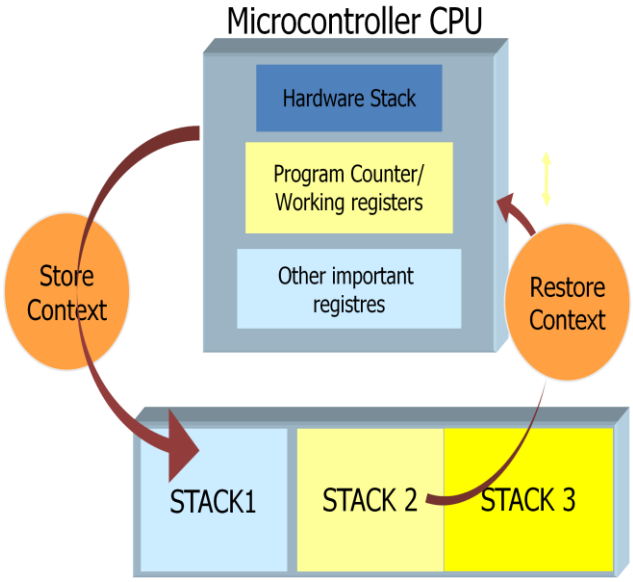


Figure 15: Switching mechanism

During this operation, FSR0 and FSR1 registers are used to point to current and next stack respectively.

The following figure shows the data memory utilization when 3 tasks are used

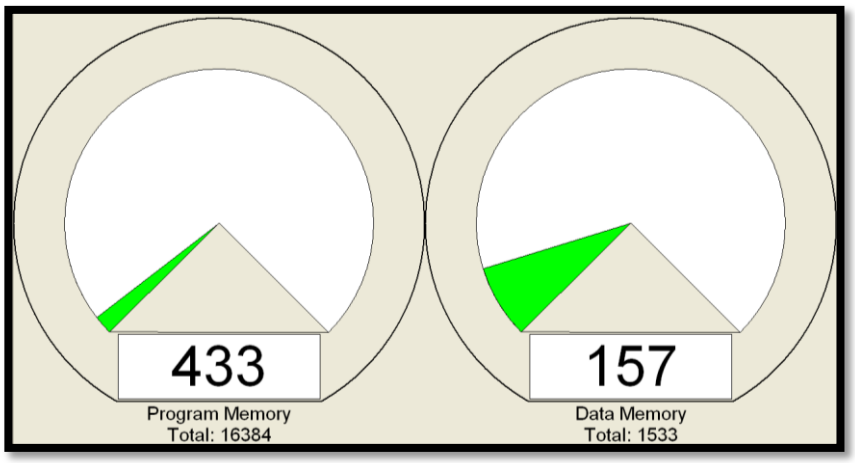


Figure 16: Memory usage for 3 Tasks

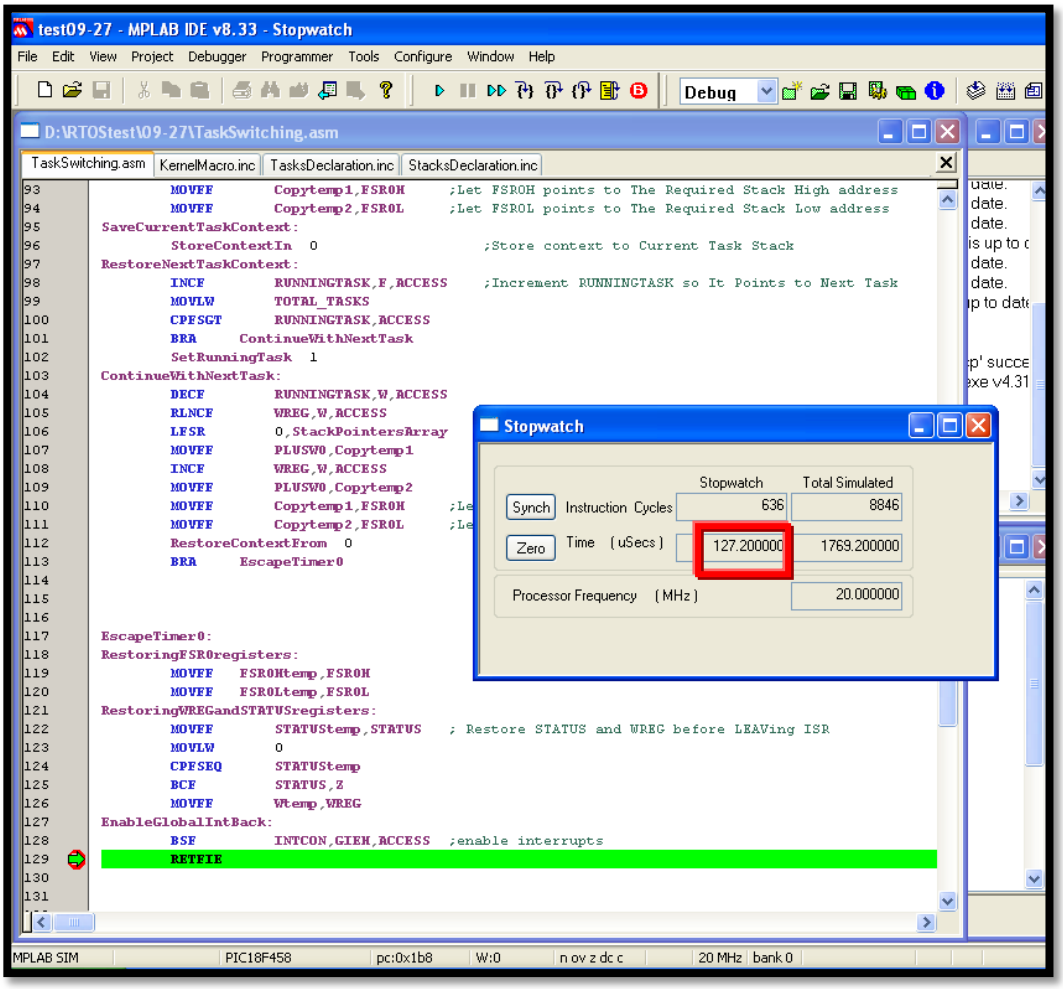


Figure 17: Time elapsed during switching mechanism = 127 µsecond

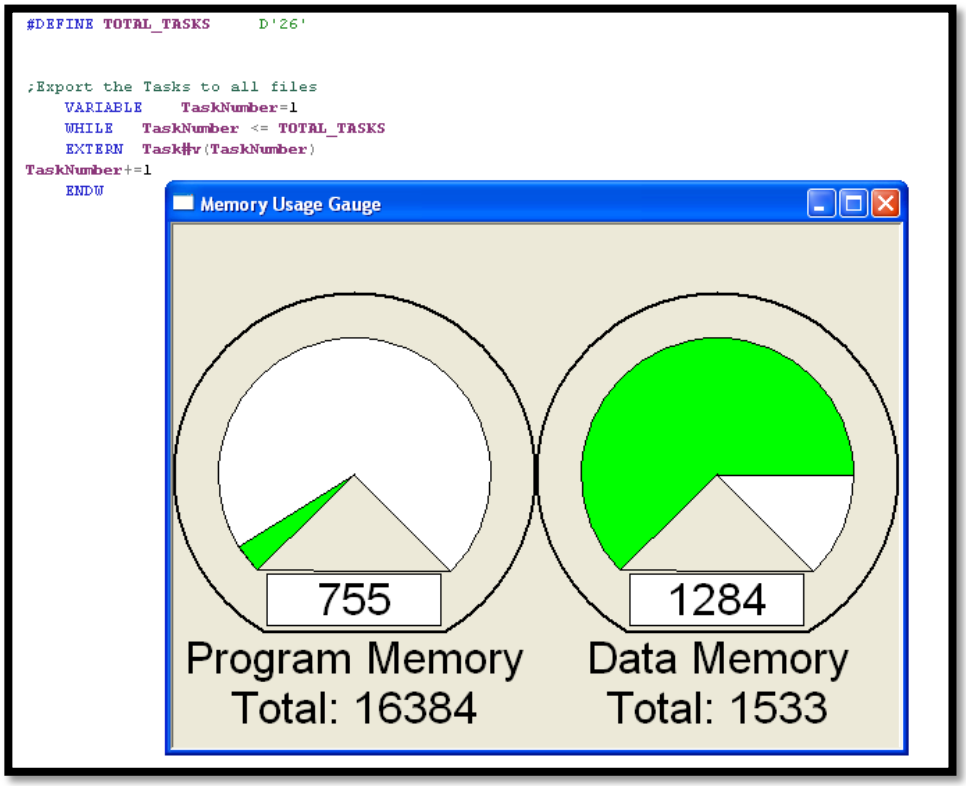


Figure 18: The memory usage for Maximum number of tasks supported=26

So, the features of the assembly version of Taj-RTOS performance can be summarized in the following table:

Attribute	Value
Switching time	50 µsecond
Maximum supported number of Tasks	26
The maximum response time	29 millisecond

4.1.2. C version of Taj RTOS

For the convenience of programming, tasks have to be written in C language. So, the kernel has to be modified to adapt the new C development environment. In C environment, **three** main issues have to be considered:

- C18 compiler initializing code (what kind of initializations are done? And how this affects RTOS?)

- How C functions' parameters and results are passed and retrieved are called (What considerations should RTOS take to deal with functions)
- The allocation of variables, temporary mathematical operations registers and other vulnerable registers (what are those variables? and where are they stored?)

a) Design considerations:

The design concept is still the same. However, some changes are made to adapt the new C environment.

The adaptations made can be listed as follows:

i) In RTOS initializer:

As mentioned in CHAPTER 2: Literature review, C18 compiler has some initializing codes which are called directly after the startup and just before the main function. After the microcontroller leaves this part, FSR registers are already setup. So, this is the most important thing which RTOS initializer has to keep. It has to keep those 3 registers unchanged during RTOS initializations.

ii) Task Switcher:

The simple assembly version of this switcher is modified in two senses:

a. It has to save the contents of the software stack whenever switching is made.
b. It has to optionally save the variables which are used by several functions (e.g. a variable "*counter*" is used by "*delay_1ms()*" function which is in turn used by more than task. So the task switcher has to store the value of "*counter*" whenever switching is done to keep each task's variables untouched by the other tasks)

b) RTOS Initializer:

The new initializer has to keep FSR registers (FSR2H, FSR2L, FSR1H, FSRL1, FSR0H and FSR0L) unchanged.

Table 6: Vulnerable registers for C environment

Register	Maximum Size
Stack pointer (STKPTR)	1
Hardware Stack (TOSU - TOSH - TOSL) x 30 levels	90
Software Stack (just in C environment)	*32
BSR	1
WREG	1
PROD (H-L)	2
FSR(0,1,2) (H-L)	6
STATUS	1
TABLE POINTER TBLPTR (U-H-L) TABLAT	4
MAXIMUM TOTAL SIZE	138

* This value can be less or more depending on how much functions are nested

c) Task switcher:

The new task switcher allocates more size for the stacks used in switching. Moreover the switching time has also increased due to the overhead process of saving and retrieving the data of the software stack.

The following figure shows a comparison between the assembly version and C version of the switcher in term of memory utilization and switching time.

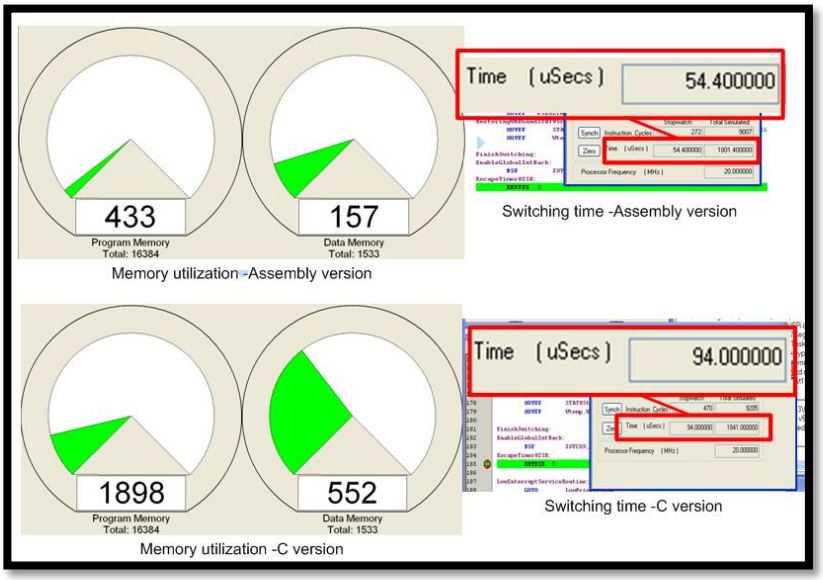


Figure 19: Comparison between C and Assembly versions of task switcher

So, the features of the C version of Taj-RTOS performance can be summarized in the following table:

Table 7: Taj RTOS performance for C version

Attribute	Value
Switching time	94 µsecond
Maximum supported number of Tasks	9
The maximum response time	10 millisecond

4.1.3. Testing of Taj RTOS

After the core of the kernel being designed, an application based on 3 tasks is designed to monitor the performance of the RTOS in managing the multitasking operations.

i) Tasks overview

3 tasks were developed. Task1 interfaces with 8 LEDs connected at PORTC. This task blinks the 8 LEDs in sequence 1 LED at a time. Task2 is a continuously running counter whose value is displayed at LCD connected to PORTD and PORTE. Task3 is programmed to scan a keypad at PORTB and display its value

on 7 segment display unit connected also to PORTC (Where LEDs are also connected).

ii) Project building and compilations

The three tasks are written in 3 separate files. The number of Tasks is declared in "TaskDeclarations.inc".

iii) Running the project using PICkit2 debugger

A break point is put at the start of the switching code. So, whenever the processing of the simulation is halted whenever switching is started. By using PICkit2 debugger connected to the circuit, we can see the switching mechanism and the multitasking behaviour in the real world.

The execution process for the 3 tasks is illustrated by the following figures.

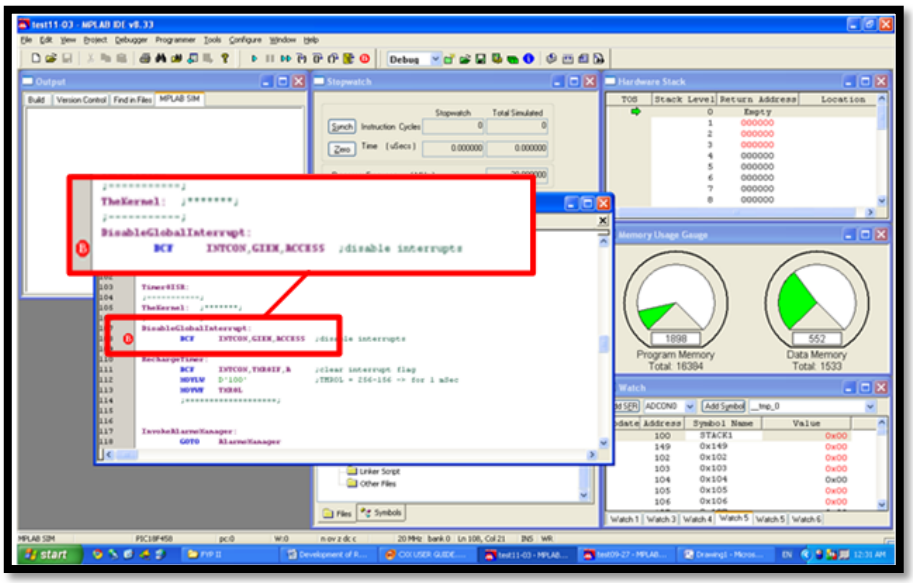


Figure 20: Project debugging with a "break point" at the start of switching code

Figure 21: The circuit with LEDs, Keypad, LCD and the debugger connected

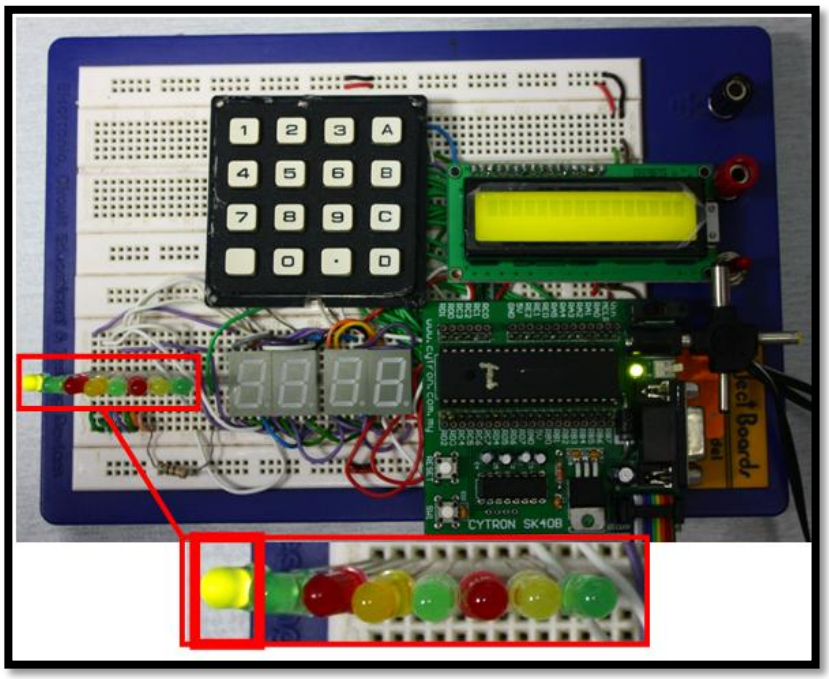


Figure 22: Task1 -only- is being processed

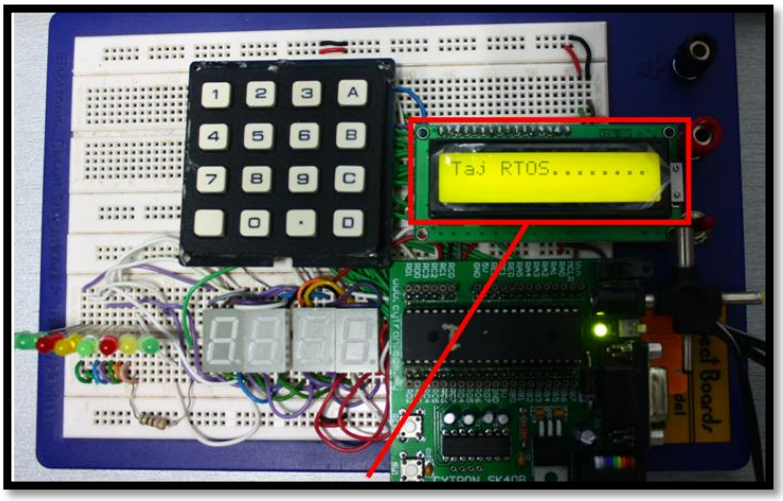


Figure 23: Task2 is -only- being processed

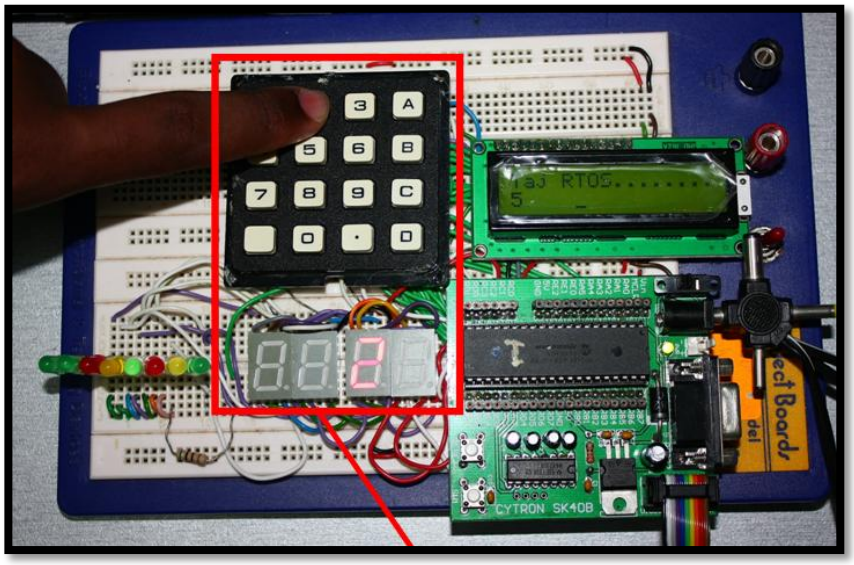


Figure 24: Task3 is -only- being processed

When breakpoint is removed, the microcontroller runs the three tasks -seemingly- at the same time while no task affects the execution of the others.

4.1.4. Writing RTOS based program

To write RTOS based program which might be loaded later to PIC18F/18C devices, users are required to do the following:

1. Using MICROCHIP MPLAB IDE software and Microchip C18 toolsuite. At the time TAJ-RTOS was developed the following software was used:
   - ❖ MPLAB IDE version 8.33
   - ❖ Microchip C18 toolsuite

- ✓ MPLAB C18 C compiler version 3.30
- ✓ MPASM Assembler version 5.30.1
- ✓ MPLINK Object Linker version 4.30.1

Figure 25: MPLAB IDE logo

However, -due to programming flexibility of TAJ-RTOS- TAJ-RTOS might be used for earlier versions of this tools, but no testing has been made regarding this matter.

NOTE: the user may need to install C18 toolsuite from microchip website.

2. In MPLAB environment, a project is to be created and PIC device to be selected

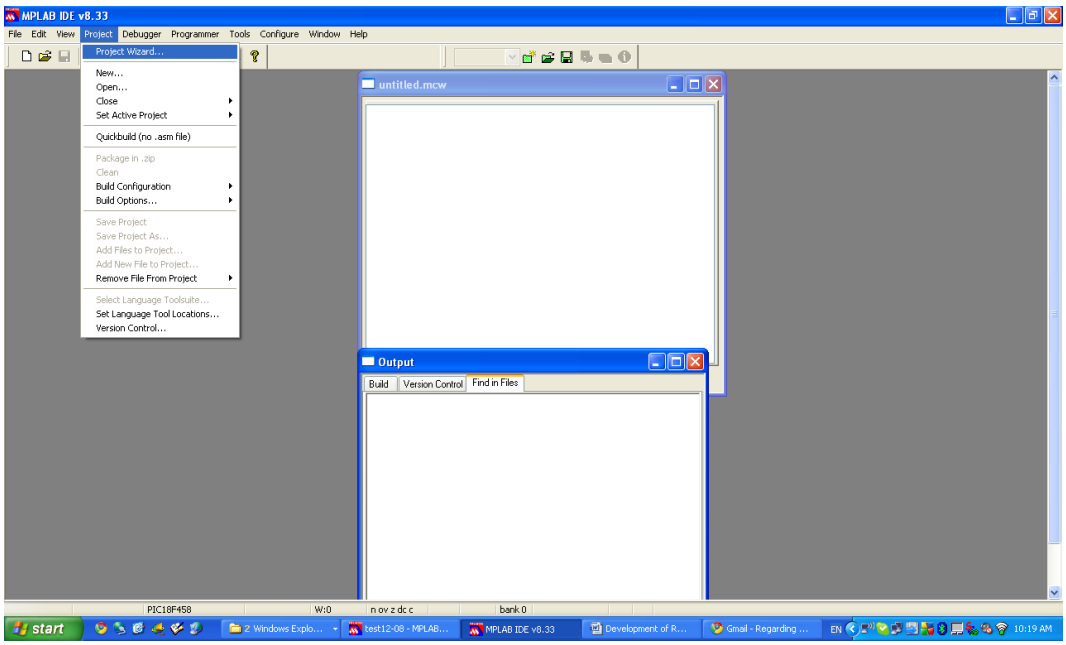


Figure 26: Creating new project

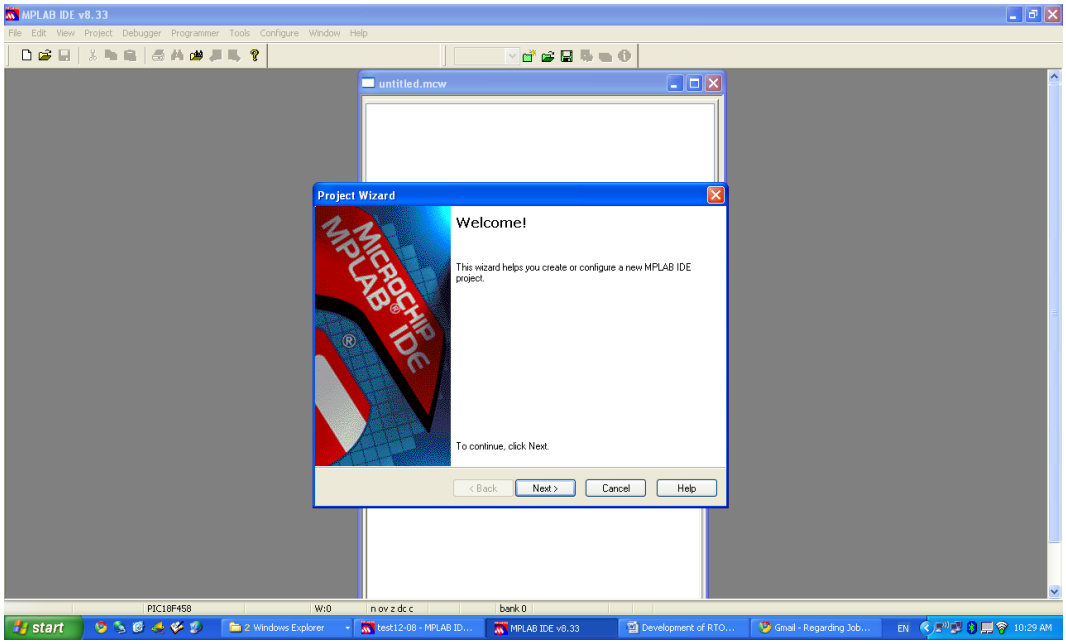


Figure 27: Welcoming Message

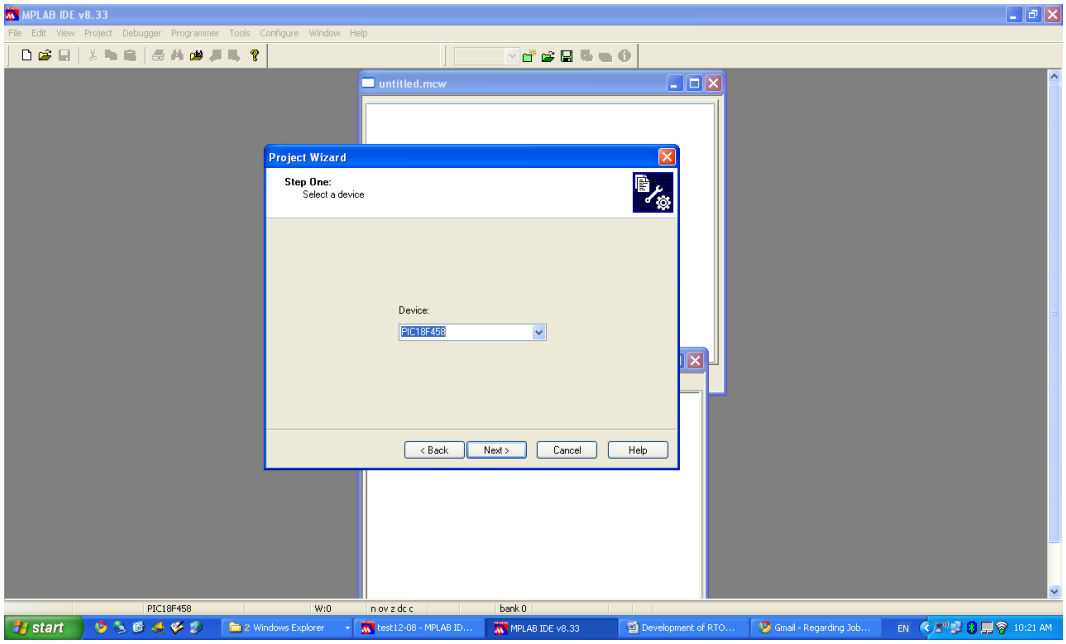


Figure 28: Selecting PIC device

3. Selecting Microchip MPASM toolsuite If (TAJ-RTOS Assembly version is considered)

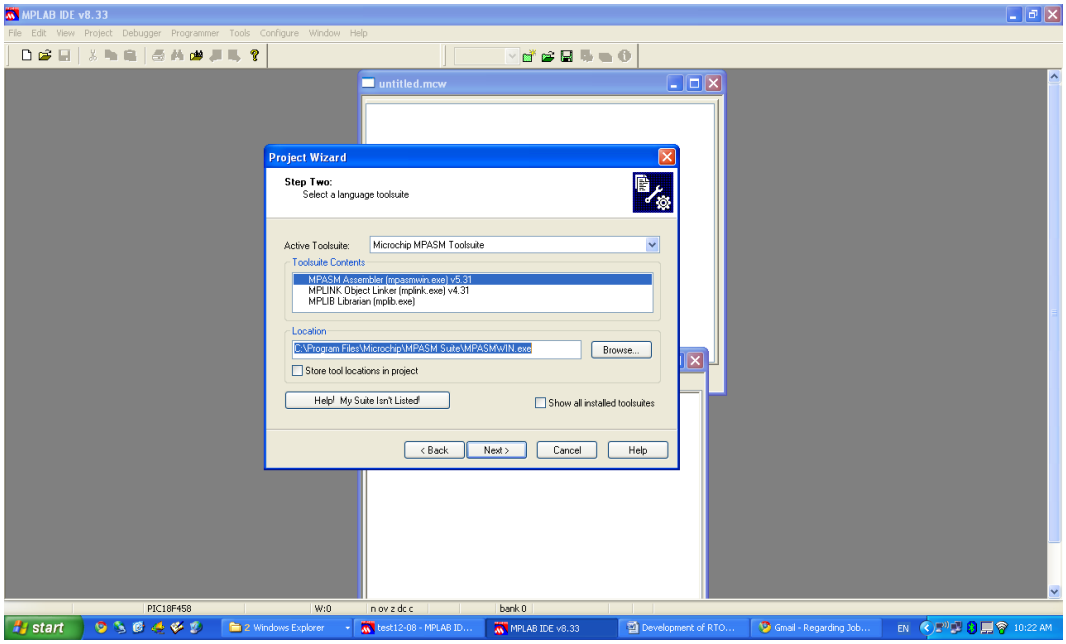


Figure 29: Selecting MPASM Toolsuite

Selecting Microchip C18 toolsuite If (TAJ-RTOS C version is considered)

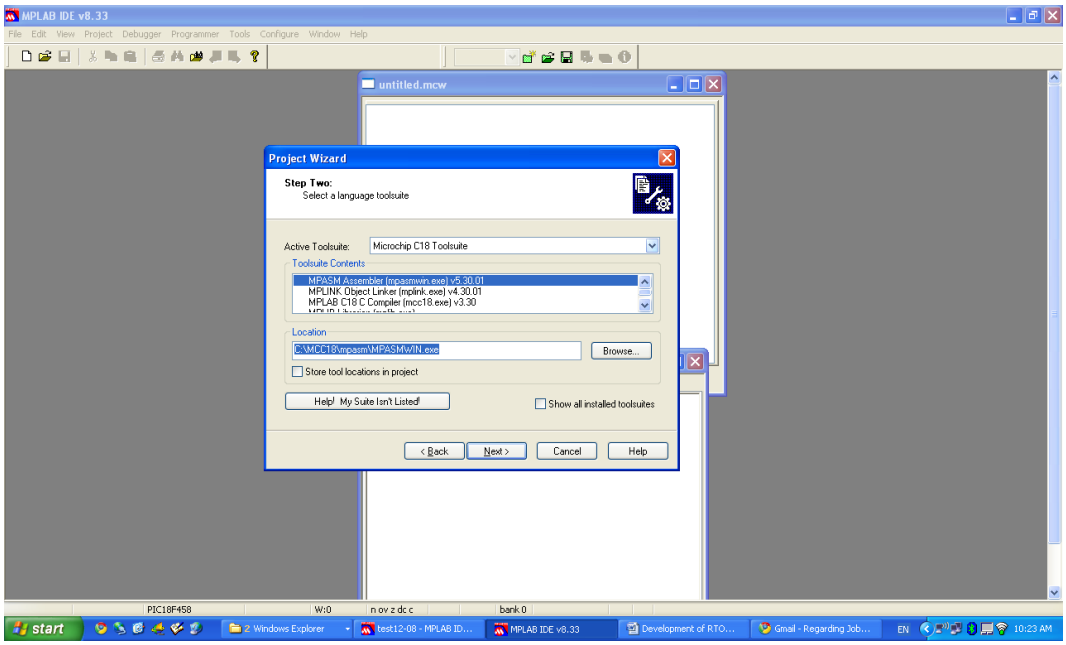


Figure 30: Selecting C18 Toolsuite

4. Adding the following files to the project
    - ❖ TAJ-RTOS assembly version:
        - ✓ RTOSsetting.inc
        - ✓ RTOSMacros.inc
        - ✓ RTOSdeclarations.inc

- ✓ RTOSkernel.asm
- ✓ Init.asm
- ✓ ISR,asm
- ✓ DeviceConfig.asm

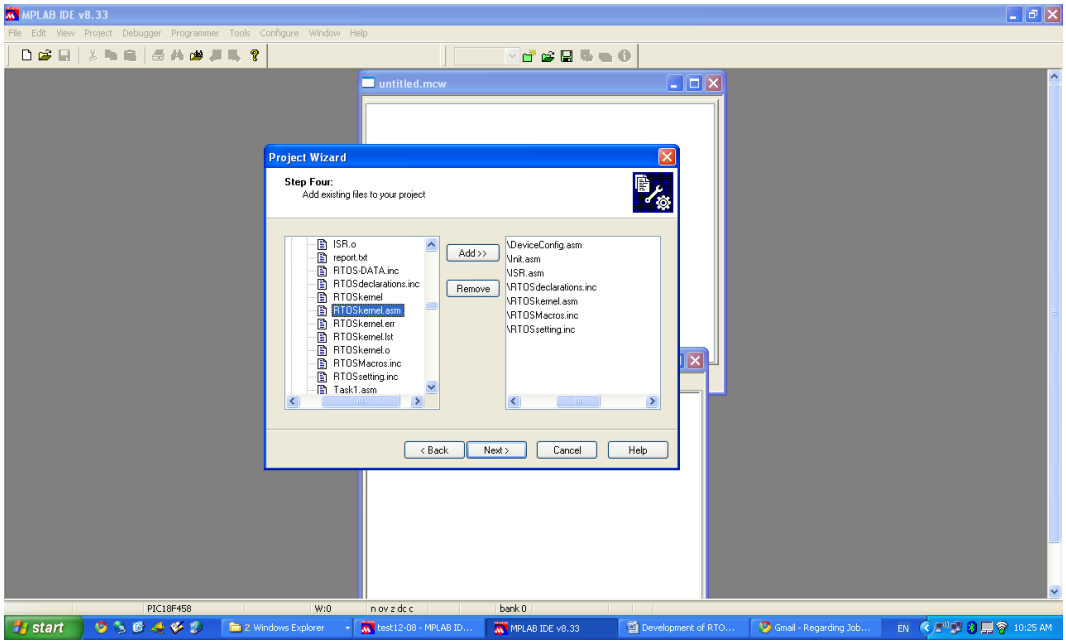


Figure 31: Adding RTOS files –Assembly version

- ❖ TAJ-RTOS C version:
  - ✓ RTOSsetting.inc
  - ✓ RTOSMacros.inc
  - ✓ RTOSdeclarations.inc
  - ✓ RTOSservices.inc
  - ✓ RTOSservices.asm
  - ✓ RTOSkernel.asm
  - ✓ ISR,c
  - ✓ Main.c
  - ✓ DeviceConfig.asm

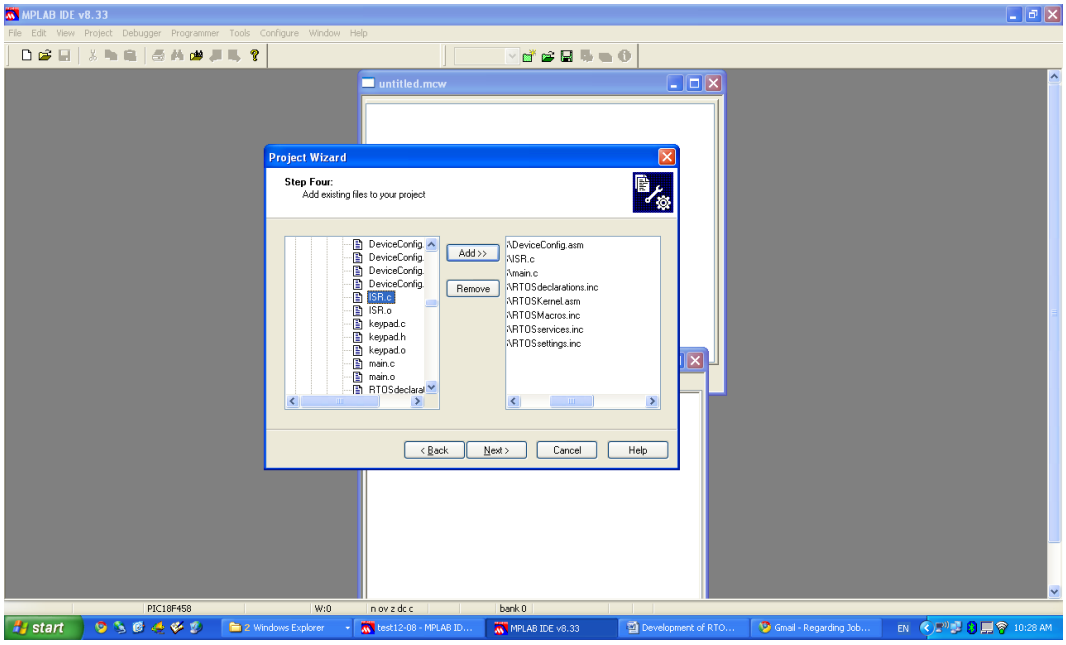


Figure 32: Adding RTOS files for C version

5. After the project is successfully created then, user may start developing required applications directly by proceeding with the following steps:
   - ❖ If Assembly-based programs are considered
     - ✓ Set the type of the processing and device configuration in **DeviceConfig.asm** file
     - ✓ Set the number of tasks required in **RTOSsetting.inc** file
     - ✓ Write any initialization code required by the application in **Init.asm** file (e.g. enable pull-ups, set timer1 interrupt…)
     - ✓ Write any Interrupt service routine in **ISR.asm** file
     - ✓ Tasks' codes might be written in a separate file with the following format:

```
Taskx
        ;code
        ;code
Here GOTO Here;
        GLOBAL Taskx
```

(NOTE: x is the number of the task, e.g. Task1, Task12 …)

- If C-based programs are considered
  - ✓ Set the type of the processing and device configuration in **DeviceConfig.asm** file
  - ✓ Set the number of tasks required, Scheduling Algorithm, Tasks' priorities and frequencies in **RTOSsetting.inc** file
  - ✓ Write any initialization code required by the application in **main.c** file before the line where InitRTOS is called
  - ✓ Write any Interrupt service routine in **ISR.asm** file
  - ✓ Tasks' codes might be written in a separate file with the following format:

```
void Taskx (void)
{
//code
//code
While(1);
}
```

(Where x is the number of the task)

6. In case of C-based programs, user may add the built in functions to interface PIC device with 7segment display, LCD, LEDs and Keypad.
7. After the application is successfully created then, user may start compiling debugging, simulating and downloading the code.
8. For advance settings (related to RTOS operation), user may check the effect of some parameters on the operation of RTOS by modifying their values in RTOSdeclarations.inc file (e.g. Stack size, StackPointerArray size, TEMPDATA and MATHDATA size…)

4.2 DISCUSSION

The task switching process is the core of supporting the multitasking processing in RTOS environment.

The assembly version of RTOS kernel is light -in coding- and exhibits very tiny processing overhead since it just deals with minimum registers and variables.

From the simulation and the coding, the deterministic behaviour of the task switcher could be noticed. With only 3 tasks, the task switching process consumes less than 100µs (for the C version) and less than 50 µs (for the assembly version).

Each task has its own scratch memory stack to save all important registers and variable while switching process. The current size of each stack is around 128 bytes. This may cause problems when the number of tasks grows larger.

It can be noticed that, the assembly version of Taj RTOS can support up to 26 Tasks while the C version supports up to 9 only. This is because fifth of microcontroller's RAM is reserved for the software stack, so RTOS would not have enough space to create stacks for the tasks.

For more responsive processing of the tasks, a pre-emptive scheduling mechanism has to be invoked. This would provide the kernel with the ability to execute the high prioritized ready tasks before the tasks with lower priority. This algorithm would not certainly guarantee that all deadlines to be met but at least an optimal performance could be achieved. However, with pre-emptive scheduling algorithm, deadlocks are very prone to occur.

Inter-task communication and synchronization are features that RTOS may increase the efficiency of RTOS. These services would make it possible for tasks to pass information from one to another. They would also make it possible for tasks to coordinate, so that they can proactively cooperate with one another. But in microcontrollers-wise, this may not be preferable, due to the limited resources within the microcontroller and the stringency of timing.

In comparison to PICos18 RTOS, Taj RTOS has many features that make it different form it -in particular- and from other RTOSes as a whole. These features can be listed in the following table:

Table 8: PICos18 and Taj RTOS comparison

Features	PICos18	Taj RTOS
Switching time	100µS	95µS
Max Number of Tasks	6	9
RAM utilization	286 bytes/task	184 bytes/task
Accessibility of kernel code and kernel parameters	No	Yes
Clarity of code	Not much	Ok (designed to be clear)
Ability of kernel to switch from any task at any moment	No	Yes
Available scheduling algorithms	2	1
Ease of getting it started	Not easy (more steps)	easy (less steps)
Drivers availability	Yes	No
Usage and Application	Industrial	Educational

CHAPTER 5
CONCLUSION AND RECOMMENDATIONS

5.1. CONCLUSION

Taj RTOS is fully capable of supporting the multitasking processing for Assembly and C programming environment. PICos18 is a good example for RTOS. However, its procedural settings make it less friendly to use. With Taj RTOS, RTOS operations are very clear to users and some of RTOS parameters are available for modification.

C18 built-in functions can still be used in Taj RTOS environment, while it is not applicable for other RTOSes (e.g. PICos18).

Taj RTOS employs simple and clear programming methodology which makes it good choice as educational tool.

Taj RTOS easy to understand, easy to use and faster to developed since it takes considerable advantages of both assembly and C codes.

Taj RTOS still has some weaknesses which could be overcome by more reviews for its codes and algorithms.

5.2. RECOMMENDATIONS

Throughout the work of this project, Taj RTOS has demonstrated its full functionality to support multitasking processing. This RTOS can be further enhanced to support various scheduling algorithms. Moreover, Taj RTOS can be made available in different forms to serve different levels and types of users (e.g. applications developers, students, beginners and experts).

Taj RTOS is very suitable for educational purposes, so it is highly recommended to be used as a teaching tool in microcontroller-based courses.

REFERENCES

[1] Wikipedia The free encyclopaedia, 'Embedded System" 10 Feb 2009 http://en.wikipedia.org/wiki/Embedded_system#cite_note-barr-glossary-0

[2] Microchip Technology Inc., "MPLAB® C18 C COMPILER USER'S GUIDE", U.S.A, 2005

[3] Microchip Technology Inc., "PIC18FXX8 Data Sheet 28/40-Pin High-Performance: Enhanced Flash Microcontrollers with CAN Module", U.S.A, 2004

[4] William Stallings, "Operating systems: Internals and Design Principles", 5th ed., Pearson Prentice Hall, 2005, pp 52.

[5] Andrew S. Tanenbaum` and Albert S. Woodhull, "Operating systems; Design and Implementation", 2nd ed., Prentice Hall, 1997, pp 90 and 91.

[6] Pragmatec S.A.R.L., "PICos18 Real time kernel for PIC18: Tutorial and Developer Guide", v 2.01, France, March 2005. Version 2.xx

[7] Pragmatec S.A.R.L., "PICos18 Real time kernel for PIC18: Kernel Interface API", v 2.xx, France, May 2006.

[8] OSEK group, "OSEK/VDX Operating System: Operating system specifications", version 2.1, revision 1, November 2000.

[9] Cytron Technologies, "SK40B PIC microcontroller start-up kit: User's Manual", version 1.1, Malaysia, December 2007.

[10] Cytron Technologies, "UIC00A USB ICSP PIC Programmer: User's Manual", version 1.7, Malaysia, November 2007.

[11] Microchip Technology Inc., "MPLAB® C18 C COMPILER: Getting Started", U.S.A, 2005

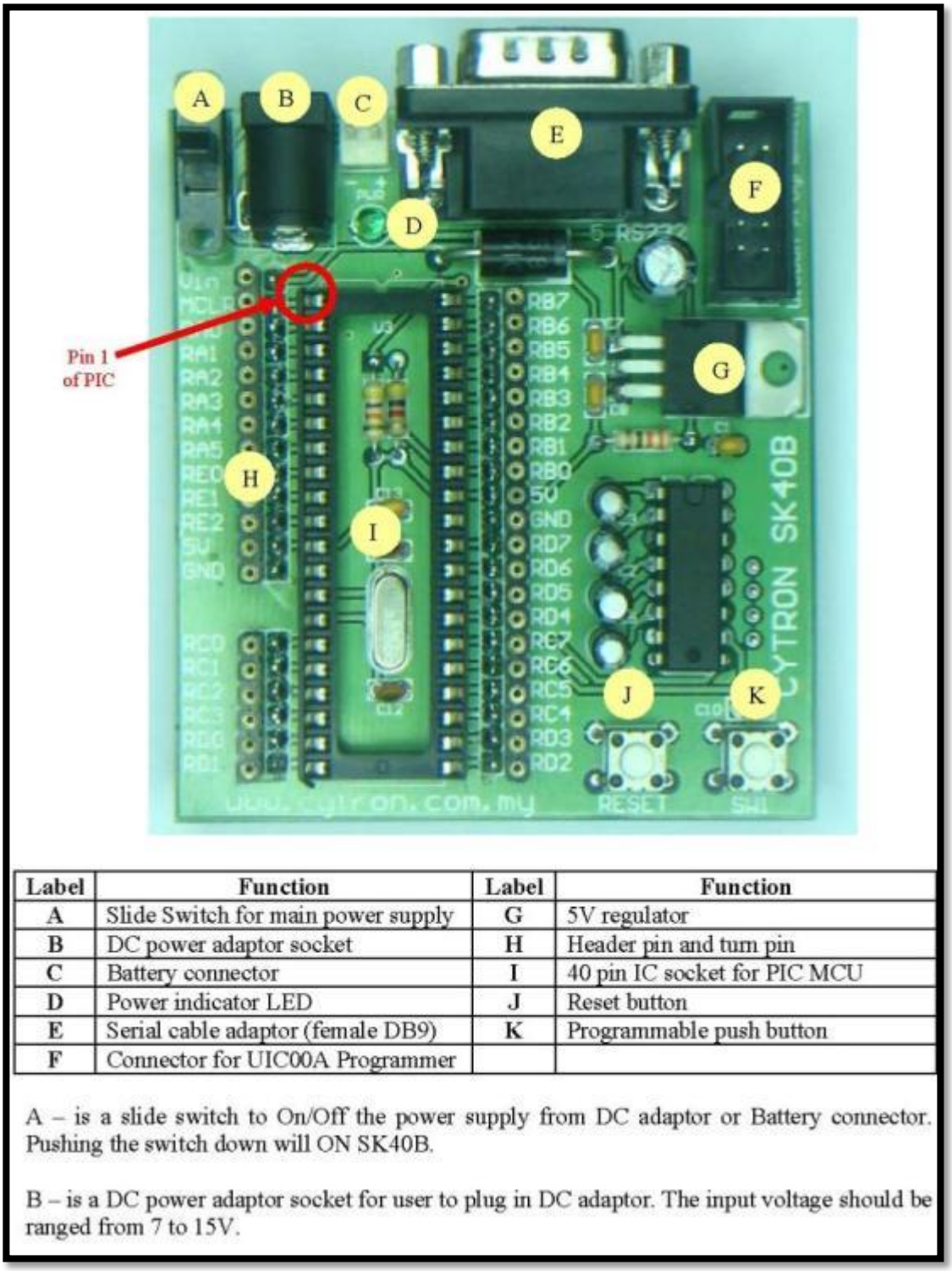
[12] Wikipedia The free encyclopaedia, 'Real Time Operating System" JUNE 2009 http://en.wikipedia.org/wiki/Real-time_operating_system

[13] Robert Betz, “Class-Notes-Introduction to Real-Time operating systems”, Department of Electrical and Computer Engineering, University of Newcastle, Australia, 2001, pp 40.

APPENDICIES

APPENDIX A

PIC18F DEVELOPMENT BOARD DRAWINGS



Label	Function	Label	Function
A	Slide Switch for main power supply	G	5V regulator
B	DC power adaptor socket	H	Header pin and turn pin
C	Battery connector	I	40 pin IC socket for PIC MCU
D	Power indicator LED	J	Reset button
E	Serial cable adaptor (female DB9)	K	Programmable push button
F	Connector for UIC00A Programmer		

A – is a slide switch to On/Off the power supply from DC adaptor or Battery connector. Pushing the switch down will ON SK40B.

B – is a DC power adaptor socket for user to plug in DC adaptor. The input voltage should be ranged from 7 to 15V.

Figure 33 PIC18F board layout

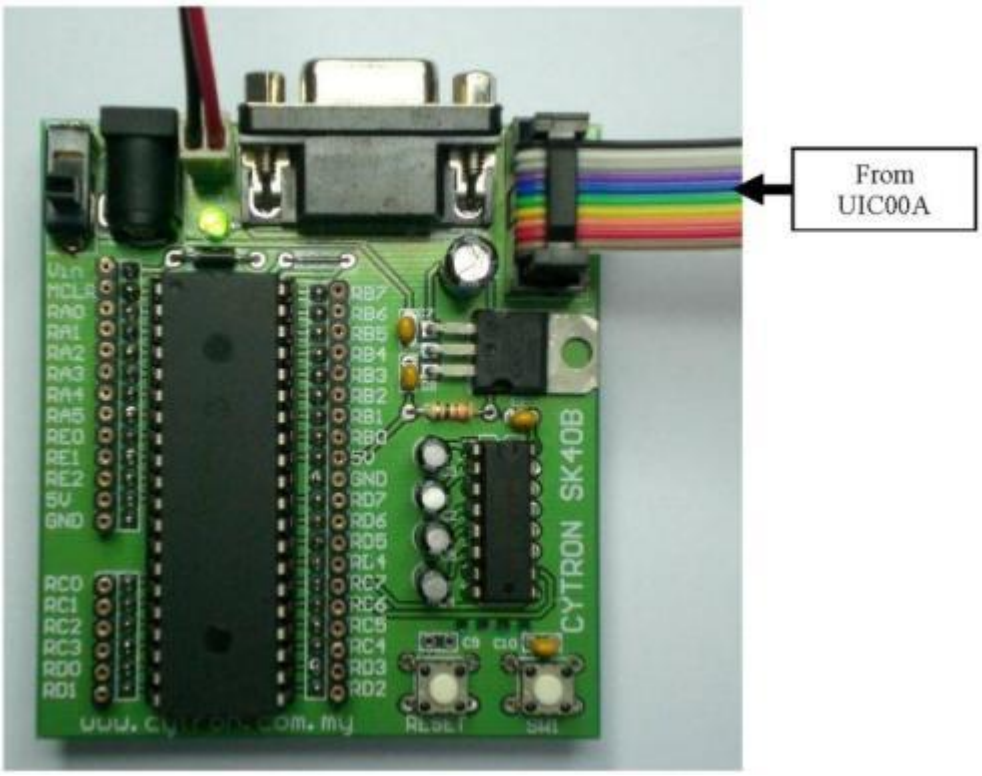


Figure 34: Board connection with PICkit 2 programmer

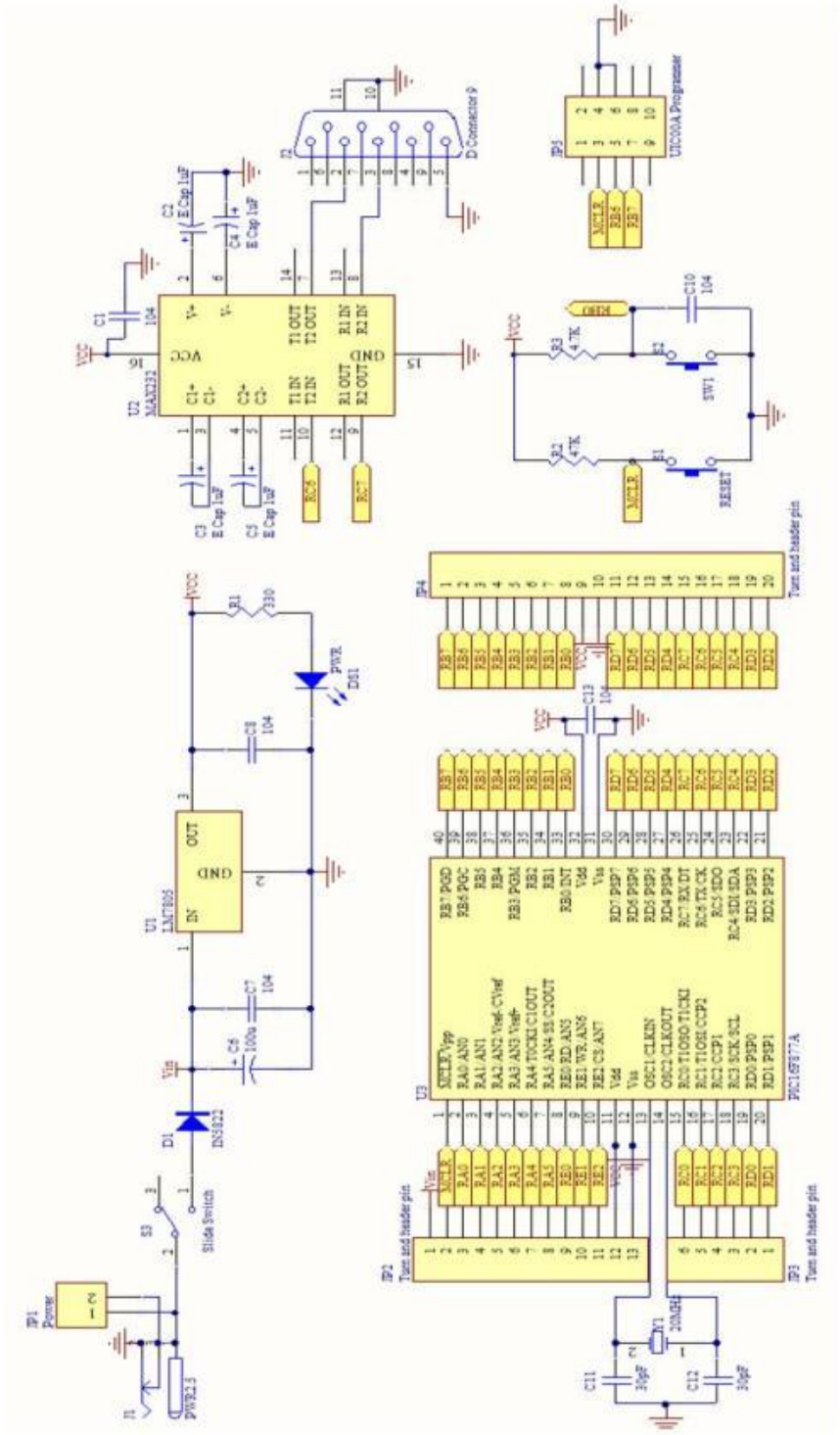
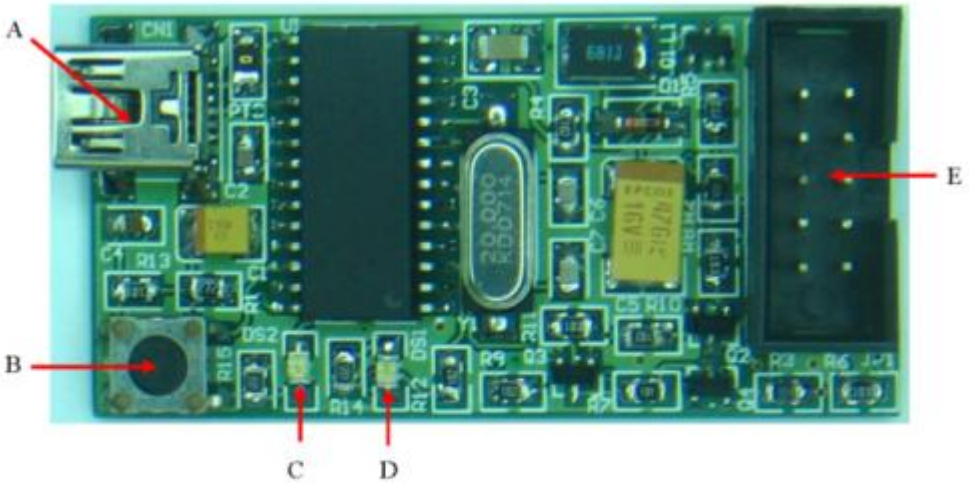


Figure 35: Schematic diagram for PIC18F board

APPENDIX B

PROGRAMMER DRAWINGS



Label	Function	Label	Function
A	Mini USB port socket	D	Busy indicator LED (red)
B	Switch to initiate write device programming	E	IDC Box Header for programming connector
C	Main power supply indicator LED (green)		

Figure 36: Board layout of PIC programmer and its parts' functions

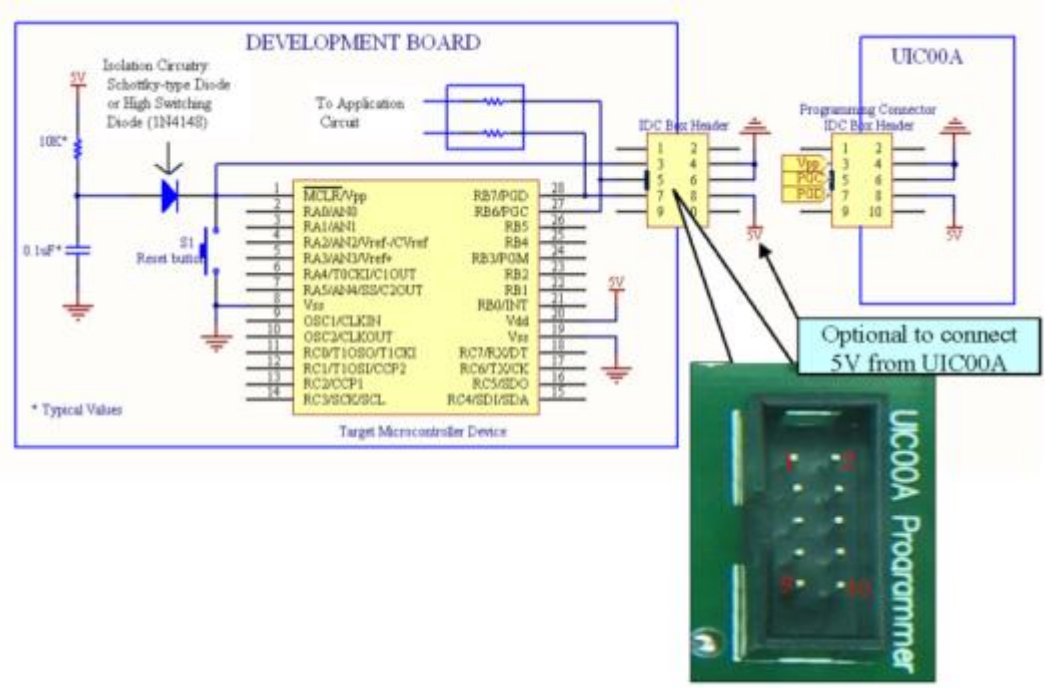


Figure 37: Programmer's connection with the development board

APPENDIX C

C18 COMPILER

DATA TYPES:

Table 9: Integer data types in C18 compiler

Type	Size	Minimum	Maximum
`char`[1,2]	8 bits	-128	127
`signed char`	8 bits	-128	127
`unsigned char`	8 bits	0	255
`int`	16 bits	-32,768	32,767
`unsigned int`	16 bits	0	65,535
`short`	16 bits	-32,768	32,767
`unsigned short`	16 bits	0	65,535
`short long`	24 bits	-8,388,608	8,388,607
`unsigned short long`	24 bits	0	16,777,215
`long`	32 bits	-2,147,483,648	2,147,483,647
`unsigned long`	32 bits	0	4,294,967,295

Note 1: A plain *`char`* is signed by default.
2: A plain *`char`* may be unsigned by default via the `-k` command-line option.

Table 10: Floating Type in C18 compiler

Type	Size	Minimum Exponent	Maximum Exponent	Minimum Normalized	Maximum Normalized
`float`	32 bits	-126	128	$2^{-126} \approx 1.17549435e-38$	$2^{128} * (2-2^{-15}) \approx 6.80564693e+38$
`double`	32 bits	-126	128	$2^{-126} \approx 1.17549435e-38$	$2^{128} * (2-2^{-15}) \approx 6.80564693e+38$

QUALIFIERS:

Table 11: "near" and "far" qualifiers in C18

	rom	ram
far	Anywhere in program memory	Anywhere in data memory (default)
near	In program memory with address less than 64K	In access memory

Table 12: Pointer size and "rom and ram" qualifiers

Pointer Type	Example	Size
Data memory pointer	`char * dmp;`	16 bits
Near program memory pointer	`rom near char * npmp;`	16 bits
Far program memory pointer	`rom far char * fpmp;`	24 bits

APPENDIX D

RESULTS

THE 18F458 SPECIFICATIONS:

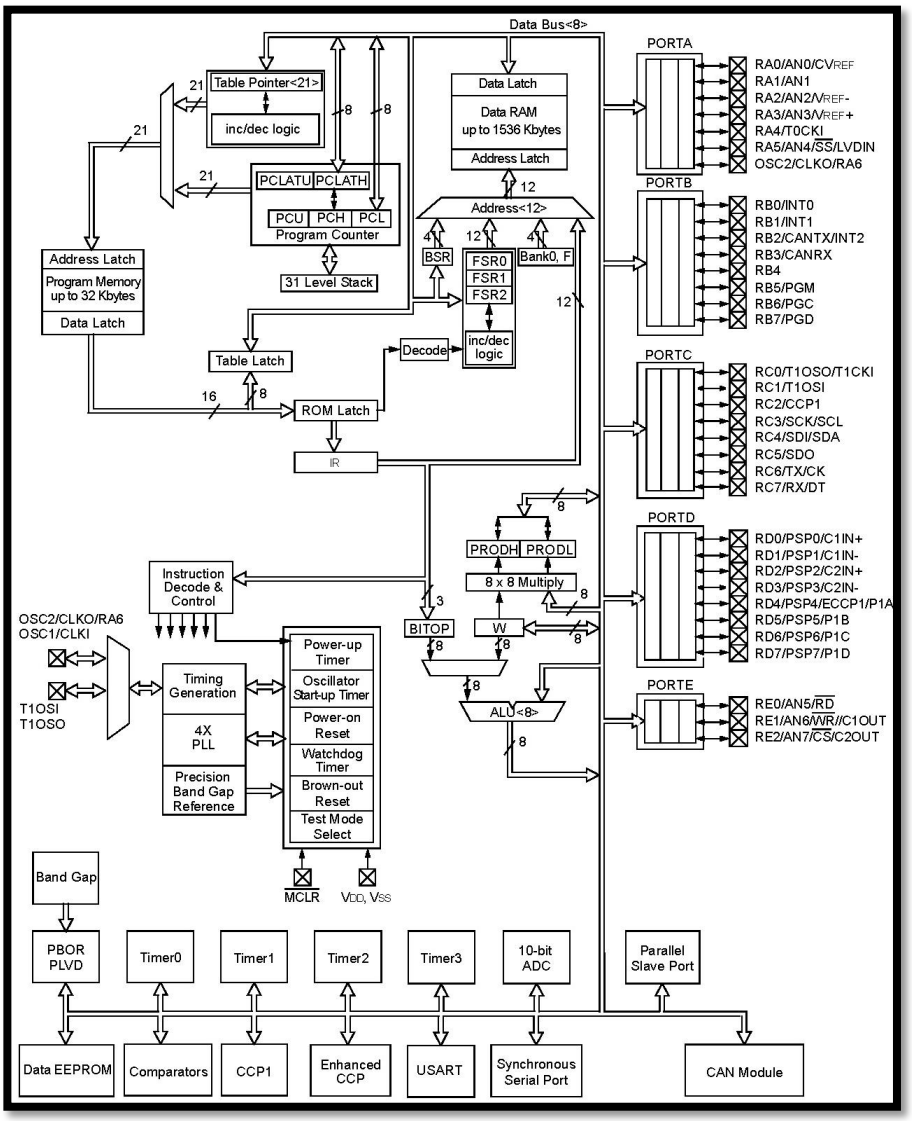


Figure 38: PIC18F458 block diagram

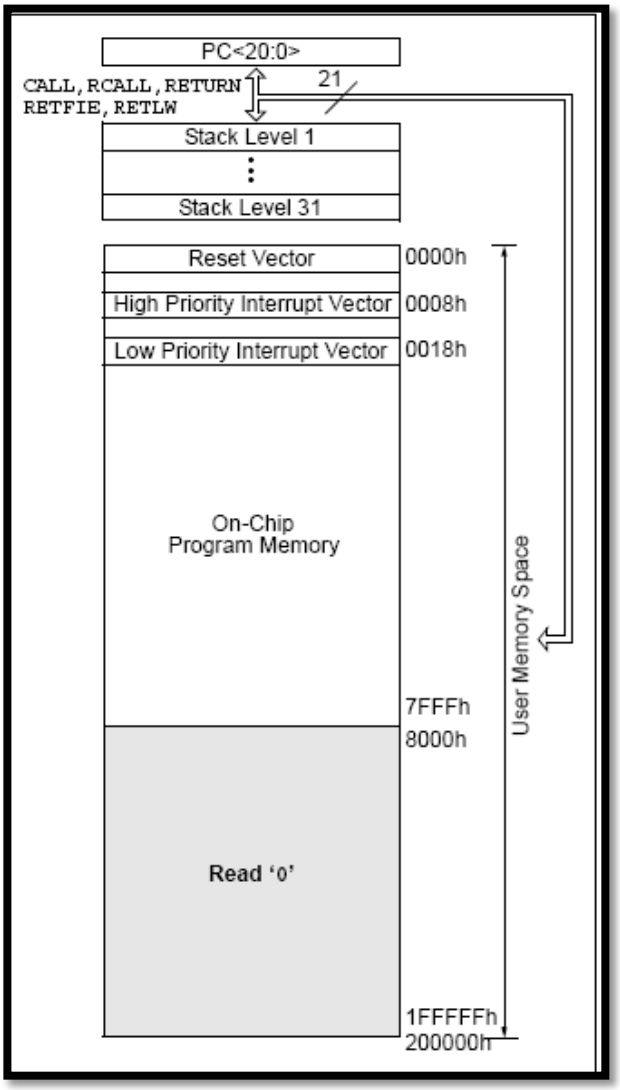


Figure 39: Program memory map for 18F458/452

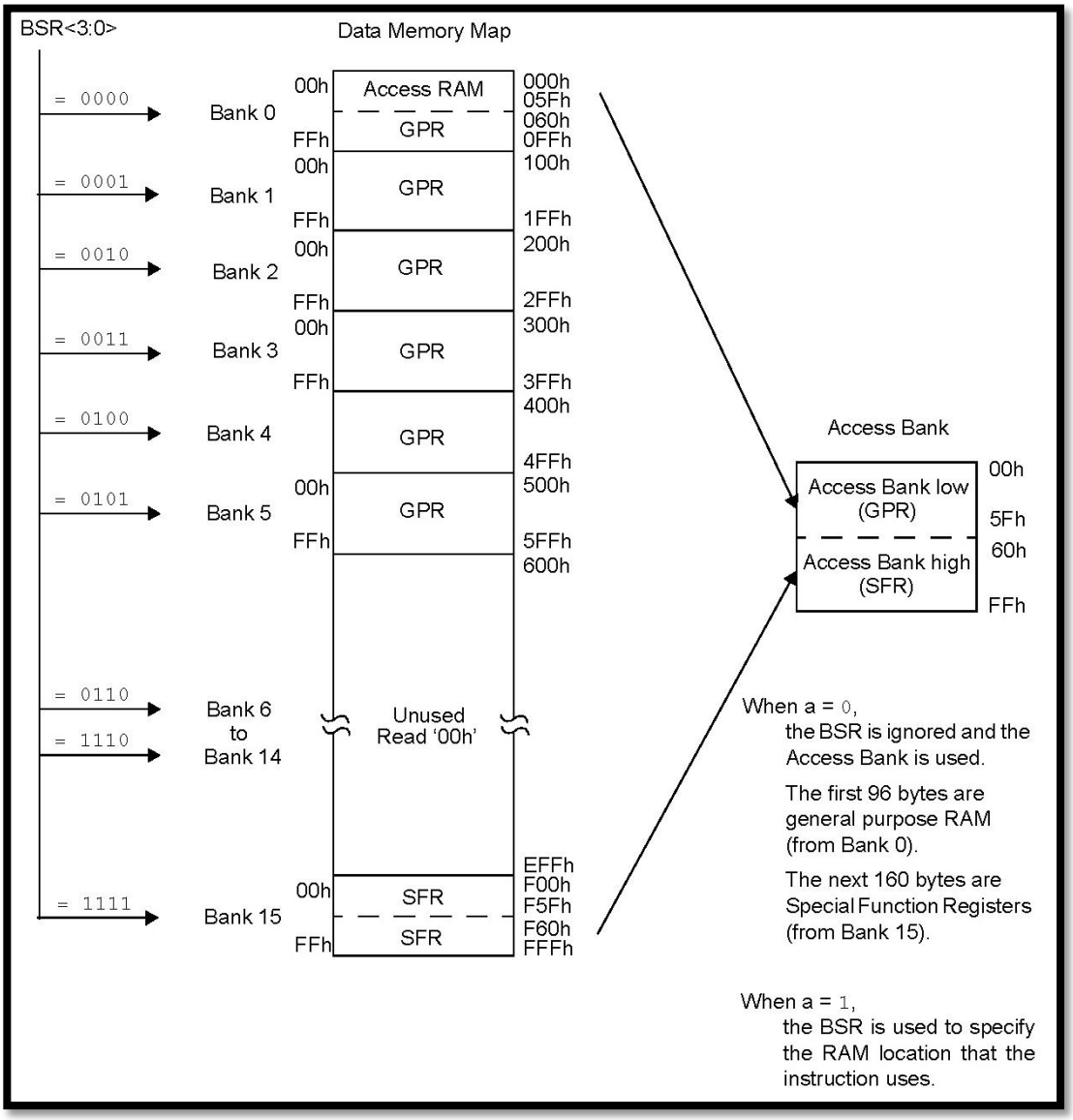


Figure 40: Data Memory Map for 18F458

Features		PIC18F248	PIC18F258	PIC18F448	PIC18F458
Operating Frequency		DC – 40 MHz	DC – 40 MHz	DC – 40 MHz	DC – 40 MHz
Internal Program Memory	Bytes	16K	32K	16K	32K
	# of Single-Word Instructions	8192	16384	8192	16384
Data Memory (Bytes)		768	1536	768	1536
Data EEPROM Memory (Bytes)		256	256	256	256
Interrupt Sources		17	17	21	21
I/O Ports		Ports A, B, C	Ports A, B, C	Ports A, B, C, D, E	Ports A, B, C, D, E
Timers		4	4	4	4
Capture/Compare/PWM Modules		1	1	1	1
Enhanced Capture/Compare/ PWM Modules		—	—	1	1
Serial Communications		MSSP, CAN, Addressable USART	MSSP, CAN, Addressable USART	MSSP, CAN, Addressable USART	MSSP, CAN, Addressable USART
Parallel Communications (PSP)		No	No	Yes	Yes
10-bit Analog-to-Digital Converter		5 input channels	5 input channels	8 input channels	8 input channels
Analog Comparators		No	No	2	2
Analog Comparators VREF Output		N/A	N/A	Yes	Yes
Resets (and Delays)		POR, BOR, RESET Instruction, Stack Full, Stack Underflow (PWRT, OST)	POR, BOR, RESET Instruction, Stack Full, Stack Underflow (PWRT, OST)	POR, BOR, RESET Instruction, Stack Full, Stack Underflow (PWRT, OST)	POR, BOR, RESET Instruction, Stack Full, Stack Underflow (PWRT, OST)
Programmable Low-Voltage Detect		Yes	Yes	Yes	Yes
Programmable Brown-out Reset		Yes	Yes	Yes	Yes
CAN Module		Yes	Yes	Yes	Yes
In-Circuit Serial Programming™ (ICSP™)		Yes	Yes	Yes	Yes
Instruction Set		75 Instructions	75 Instructions	75 Instructions	75 Instructions
Packages		28-pin SPDIP 28-pin SOIC	28-pin SPDIP 28-pin SOIC	40-pin PDIP 44-pin PLCC 44-pin TQFP	40-pin PDIP 44-pin PLCC 44-pin TQFP

Figure 41: PIC18Fxx8 devices' features

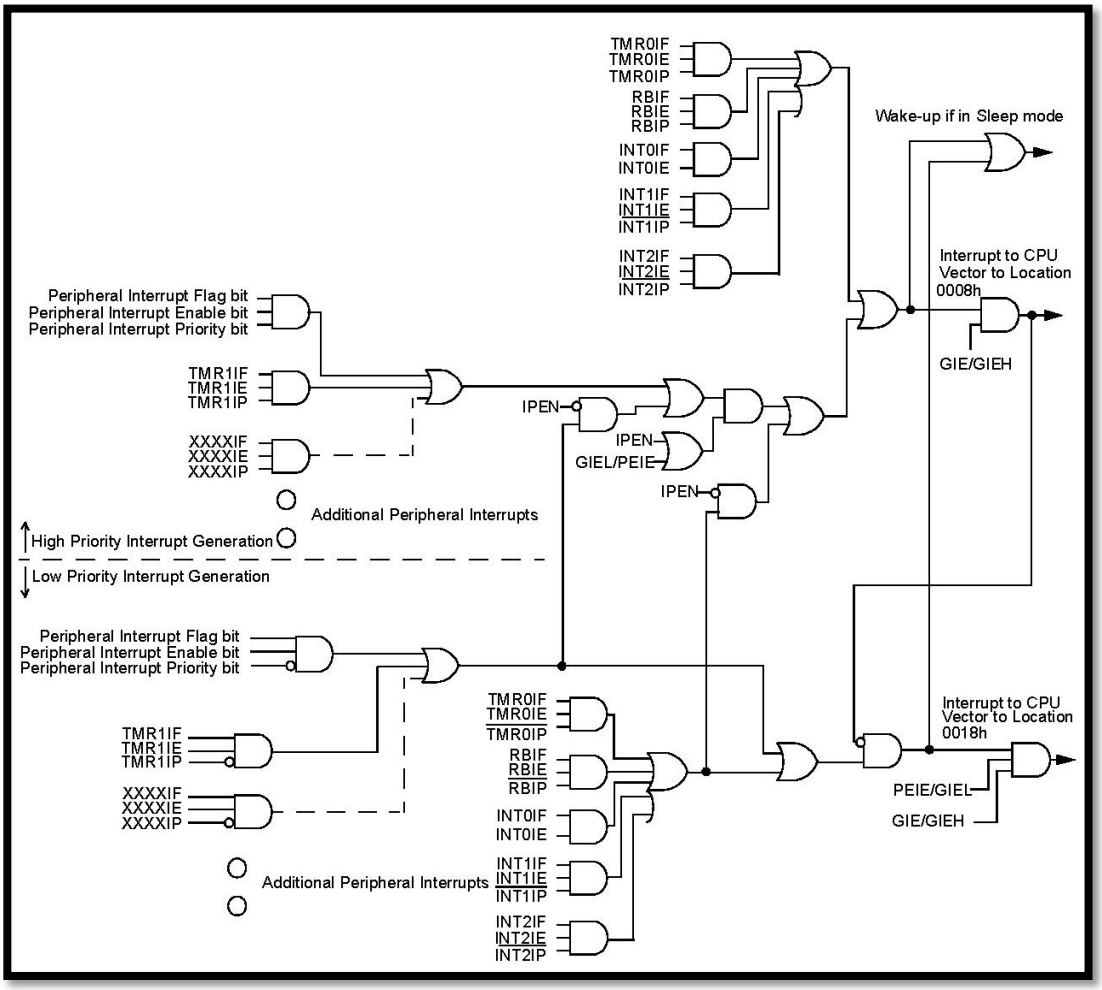


Figure 42: Interrupt schematic diagram for 18F458

APPENDIX E
TAJ RTOS CODES

Assembly version:

File 1: RTOSkernel.ASM

```
;===============================================================================
;**************************Include files Processor setting***************
;===============================================================================
        #include    "p18cxxx.inc"
        #include    "RTOSdeclarations.inc"
        #include    "RTOSMacros.inc"

;**************************Main Code*************************************
;===============================================================================
;*** Structure of program memory ***
ResetVec        CODE        0x00
                goto        Init

InterruptHighV  CODE        0x08
SaveWREGandSTATUSregisters:
                MOVFF       STATUS,STATUStemp   ; Save STATUS and WREG before entering ISR
                MOVWF       Wtemp,ACCESS
                goto        HighInterruptServiceRoutine

InterruptLowV   CODE        0x18
SaveWREGandSTATUSRegisters:
                MOVFF       STATUS,STATUStempL  ; Save STATUS and WREG Before entering ISR
                MOVWF       WtempL,A
                goto        LowPriorityISR

StartRTOS:
    GLOBAL StartRTOS

InitRTOS:
;Backup the following registers that might be used in Init code ....
        MOVFF       STATUS,STATUStemp   ; Save STATUS and WREG before entering RTOS
        MOVWF       Wtemp,ACCESS
BackupFSR0register:
        MOVFF       FSR0H,FSR0Htemp
        MOVFF       FSR0L,FSR0Ltemp

InitKernelTimer:
SetTimer0:
DisableGlobalInt:
        BCF         INTCON, GIE     ;disable global and enable TMR0 interrupt
        BSF         INTCON, T0IE

SetTMR0asHighPriorityInt:
        BSF         INTCON2,TMR0IP  ;TMR0 high priority
EnablePriorityLevel:
        BSF         RCON,IPEN,A     ;enable priority levels
        CLRF        TMR0H,A         ;clear timer
        CLRF        TMR0L,A         ;clear timer

Set1msTMR0:
        MOVLW       B'11000100'
        MOVWF       T0CON           ;set up timer0 - prescaler 1:16
EnableGlobalInt:
        BSF         INTCON,GIEH,A   ;enable interrupts

InitializeRunningTask:
        SetRunningTask  0           ; At First time running -> Set running task to NONE (i.e. No
task is running yet)

InitializeAllStacks:
        InitializeStacks

        InitializeStackPointersArray    0,StackPointersArray


CallTask1:
        SetRunningTask                  1               ;Task1 is running

ReturnFSR0register:
;Return Important Registers Initial Values
        MOVFF   FSR0Htemp,FSR0H
        MOVFF   FSR0Ltemp,FSR0L
ReturnSTATUSwregRegister:
        MOVFF       STATUStemp,STATUS   ; Restore STATUS and WREG before LEAVing ISR
        MOVFF       Wtemp,WREG

        GOTO                            Task1           ;execute Task1
;_______________________________________________________________________________;
```

```
HighInterruptServiceRoutine:
CheckInterruptSource:
        BTFSS   INTCON,TMR0IF,A     ;check for TMR0 overflow
        GOTO    HighPriorityISR


BackupFSR0registers:
        MOVFF   FSR0H,FSR0Htemp
        MOVFF   FSR0L,FSR0Ltemp

TheKernel:
DisableGlobalInterrupt:
        BCF         INTCON,GIEH,ACCESS  ;disable interrupts

RechargeTimer:
        BCF     INTCON,TMR0IF,A     ;clear interrupt flag
        MOVLW   D'100'              ;TMR0L = 256-156 -> for 1 mSec
        MOVWF   TMR0L
        ;==================;

TaskSwitcher:

DetermineRunningTask:
        DECF        RUNNINGTASK,W,ACCESS
        RLNCF       WREG,W,ACCESS
        LFSR        0,StackPointersArray
        MOVFF       PLUSW0,Copytemp1
        INCF        WREG,W,ACCESS
        MOVFF       PLUSW0,Copytemp2
        MOVFF       Copytemp1,FSR0H     ;Let FSR0H points to The Required Stack High address
        MOVFF       Copytemp2,FSR0L     ;Let FSR0L points to The Required Stack Low address
SaveCurrentTaskContext:
        StoreContextIn  0                   ;Store context to Current Task Stack
RestoreNextTaskContext:
        INCF        RUNNINGTASK,F,ACCESS    ;Increment RUNNINGTASK so It Points to Next Task
        MOVLW       TOTAL_TASKS
        CPFSGT      RUNNINGTASK,ACCESS
        BRA     ContinueWithNextTask
        SetRunningTask  1
ContinueWithNextTask:
        DECF        RUNNINGTASK,W,ACCESS
        RLNCF       WREG,W,ACCESS
        LFSR        0,StackPointersArray
        MOVFF       PLUSW0,Copytemp1
        INCF        WREG,W,ACCESS
        MOVFF       PLUSW0,Copytemp2
        MOVFF       Copytemp1,FSR0H     ;Let FSR0H points to The Required Stack High address
        MOVFF       Copytemp2,FSR0L     ;Let FSR0L points to The Required Stack Low address
        RestoreContextFrom  0


EscapeTimer0:
RestoringFSR0registers:
        MOVFF   FSR0Htemp,FSR0H
        MOVFF   FSR0Ltemp,FSR0L
RestoringWREGandSTATUSregisters:
        MOVFF       STATUStemp,STATUS   ; Restore STATUS and WREG before LEAVing ISR
        MOVFF       Wtemp,WREG
EnableGlobalIntBack:
        BSF         INTCON,GIEH,ACCESS  ;enable interrupts

        RETFIE



        END                 ; end of assembly language program
```

File 2: KernelMacro.INC

```
;===============================================================================
;**************************MACROs*****************************************
;Description:
;           This file contains all macros called in Kernel file
;           List of Macros:
;           1. AssignValue2Register:        assign <value> to <reg> at Any RAM mem.
;           2. AssignValue2AccessRegister:  assign <value> to <reg> at access RAM
;           3. SetRunningTask:              Set <value> to <RUNNINGTASK>    register
;           4. InitializeStack:             Initialize The -RTOS created- stacks of Tasks where al
important registers are saved
;           5. InitializeStacks:            Initialize All Tasks' Stacks -created by RTOS to save
important registers-
;           5. StoreContextIn:              Store Important Registers To Memory block where FSR0 i
pointing to
;           6. ReStoreContextFrom:          ReStore Important Registers From Memory block where FS
is pointing to
;===============================================================================

AssignValue2Register        MACRO   value,register
                            MOVLB   high register
                            MOVLW   value
                            MOVWF   low register,BANKED
                            ENDM

AssignValue2AccessRegister  MACRO   value,register
                            MOVLW   value
                            MOVWF   register,ACCESS
                            ENDM

SetRunningTask              MACRO   TaskNo
                            MOVLW   TaskNo
                            MOVWF   RUNNINGTASK,ACCESS
                            ENDM

InitializeStack             MACRO       Stackaddress,TaskNumber
                            LFSR        0,Stackaddress
                            MOVLW       1                           ;FillHWstackPointer
                            MOVWF       POSTINC0                    ;FillHWstackReturnAddress To
Tasks' starting addresses
                            MOVLW       upper Task#v(TaskNumber)    ;Move TOSU
                            MOVWF       POSTINC0
                            MOVLW       high Task#v(TaskNumber)     ;Move TOSH
                            MOVWF       POSTINC0
                            MOVLW       low Task#v(TaskNumber)      ;Move TOSL
                            MOVWF       POSTINC0
                            MOVFF       BSR,POSTINC0
                            MOVFF       Wtemp,POSTINC0
                            MOVFF       STATUStemp,POSTINC0
                            MOVFF       PRODH,POSTINC0
                            MOVFF       PRODL,POSTINC0
                            MOVFF       FSR0Htemp,POSTINC0
                            MOVFF       FSR0Ltemp,POSTINC0
                            MOVFF       FSR1H,POSTINC0
                            MOVFF       FSR1L,POSTINC0
                            MOVFF       FSR2H,POSTINC0
                            MOVFF       FSR2L,POSTINC0
                            MOVFF       TBLPTRU,POSTINC0
                            MOVFF       TBLPTRH,POSTINC0
                            MOVFF       TBLPTRL,POSTINC0
                            MOVFF       TABLAT,POSTINC0
                            ENDM

InitializeStacks            MACRO
                            LOCAL   TaskNumber
TaskNumber=1
                            WHILE   TaskNumber<=TOTAL_TASKS
                            InitializeStack STACK#v(TaskNumber),TaskNumber
TaskNumber+=1
                            ENDW
                            ENDM

InitializeStackPointersArray    MACRO   FSR, Array
                            LFSR    FSR,Array
                            LOCAL   StackNumber=1
                            WHILE   StackNumber <= TOTAL_TASKS
                            MOVLW   HIGH STACK#v(StackNumber)
                            MOVWF   POSTINC0,ACCESS
                            MOVLW   LOW STACK#v(StackNumber)
                            MOVWF   POSTINC0,ACCESS
StackNumber+=1
```

```
                        ENDW
                        ENDM

        variable    labelGenerator=1
StoreContextIn          MACRO               FSRno
labelGenerator+=1
                        MOVFF               STKPTR,POSTINC#v(FSRno)
StoreHWStack#v(labelGenerator)                          ;To avoid the dublication of
label
                        MOVFF               TOSU,POSTINC#v(FSRno)
                        MOVFF               TOSH,POSTINC#v(FSRno)
                        MOVFF               TOSL,POSTINC#v(FSRno)
                        DECFSZ              STKPTR,F,A
                        GOTO                StoreHWStack#v(labelGenerator)
                        MOVFF               BSR,POSTINC#v(FSRno)
                        MOVFF               Wtemp,POSTINC#v(FSRno)
                        MOVFF               STATUStemp,POSTINC#v(FSRno)
                        MOVFF               PRODH,POSTINC#v(FSRno)
                        MOVFF               PRODL,POSTINC#v(FSRno)
                        MOVFF               FSR0Htemp,POSTINC#v(FSRno)
                        MOVFF               FSR0Ltemp,POSTINC#v(FSRno)
                        MOVFF               FSR1H,POSTINC#v(FSRno)
                        MOVFF               FSR1L,POSTINC#v(FSRno)
                        MOVFF               FSR2H,POSTINC#v(FSRno)
                        MOVFF               FSR2L,POSTINC#v(FSRno)
                        MOVFF               TBLPTRU,POSTINC#v(FSRno)
                        MOVFF               TBLPTRH,POSTINC#v(FSRno)
                        MOVFF               TBLPTRL,POSTINC#v(FSRno)
                        MOVFF               TABLAT,POSTINC#v(FSRno)
                        ENDM

RestoreContextFrom      MACRO               FSRno
labelGenerator+=1
                        MOVFF               POSTINC#v(FSRno),STKPTR
                        MOVFF               STKPTR,STKPTRtemp
RestoreHWStack#v(labelGenerator)
                        MOVF                POSTINC#v(FSRno),W
                        MOVWF               TOSU
                        MOVF                POSTINC#v(FSRno),W
                        MOVWF               TOSH
                        MOVF                POSTINC#v(FSRno),W
                        MOVWF               TOSL
                        DECFSZ              STKPTR,F,A
                        GOTO                RestoreHWStack#v(labelGenerator)
                        MOVFF               STKPTRtemp,STKPTR
                        MOVFF               POSTINC#v(FSRno),BSR
                        MOVFF               POSTINC#v(FSRno),Wtemp
                        MOVFF               POSTINC#v(FSRno),STATUStemp
                        MOVFF               POSTINC#v(FSRno),PRODH
                        MOVFF               POSTINC#v(FSRno),PRODL
                        MOVFF               POSTINC#v(FSRno),FSR0Htemp
                        MOVFF               POSTINC#v(FSRno),FSR0Ltemp
                        MOVFF               POSTINC#v(FSRno),FSR1H
                        MOVFF               POSTINC#v(FSRno),FSR1L
                        MOVFF               POSTINC#v(FSRno),FSR2H
                        MOVFF               POSTINC#v(FSRno),FSR2L
                        MOVFF               POSTINC#v(FSRno),TBLPTRU
                        MOVFF               POSTINC#v(FSRno),TBLPTRH
                        MOVFF               POSTINC#v(FSRno),TBLPTRL
                        MOVFF               POSTINC#v(FSRno),TABLAT
                        ENDM

;=============================================================================
```

File 3: RTOSDeclarations.INC

```
#include    "RTOSsetting.inc"

        UDATA_ACS                       ;Declaring temp registers in access memory
Wtemp       RES         1
STKPTRtemp  RES         1
STATUStemp  RES         1
STATUStempL RES         1
WtempL      RES         1
RUNNINGTASK RES         1
FSR0Htemp   RES         1
FSR0Ltemp   RES         1
Copytemp1   RES         1
Copytemp2   RES         1


;Tasks Declaration
;Export the Tasks to all files
    VARIABLE    TaskNumber=1
    WHILE   TaskNumber <= TOTAL_TASKS
    EXTERN  Task#v(TaskNumber)
TaskNumber+=1
    ENDW

;Export Init AND ISR labels:
    EXTERN  Init, HighPriorityISR,LowPriorityISR


;Stacks Declaration
;Decription: To declare the required stack and temp data (wtemp, statustemp) for the declared tasks
in "TasksDeclaration.inc" file

#DEFINE     STACKSIZE   D'1'+D'3'*D'31'+D'1'+D'1'+D'1'+D'12'
; About STACKSIZE:
; The stack size depends on how many registers are required to be saved
; For 1:STKPTR,31*3:Stack Spaces, 1:BSR, 1:STATUS, 1:WREG, 12:extra
; 12 SFRs Extention: PRODH-L (2 SFRs), FSR0,1,2(6 SFRs) , TBLPTR(U-H-L) (3 SFRs), TABLAT

;Reserve Space for stacks depending on the definition of tasks
;Declaring Scratch memory section for storing task1 variables when task switching
;Stack size = 'STACKSIZE' Byte

;Reserve Space for stacks depending on the definition of tasks
    VARIABLE    TaskNumber=1
    WHILE   TaskNumber<=TOTAL_TASKS
_STACK#v(TaskNumber)    UDATA                   ;Declaring Scratch memory section for storing task1
variables when task switching
STACK#v(TaskNumber)     RES         STACKSIZE
TaskNumber+=1
    ENDW


;   Declare Stacks' pointers
;Reserve Space for stack pointers depending on the definition of tasks
_StackPointersSpace     UDATA
StackPointersArray      RES         (TOTAL_TASKS*2)
```

File 4: DeviceConfig:

```
#include    <p18f458.inc>
;==============================================================================
;************************Configuration Bits*****************************
;==============================================================================
;*** Setting Microcontroller to 20MHz, NOWDG, NOPROTECTION, NOBOR *********
    CONFIG  OSCS=OFF , OSC =HS                              ;No oscilator switch and 20Mhz oscilator
    CONFIG  BOR =OFF , PWRT=OFF                             ;Brown out reset=off , power up timer ->
off
    CONFIG  WDT=OFF                                         ;Watch dog timer -> off
    CONFIG  DEBUG=OFF, LVP =OFF , STVR=OFF                  ;Background Debugger, Low Voltage ICSP,
Stack over/underflow Reset -> OFF
    CONFIG  CP0 =OFF , CP1 =OFF , CP2 =OFF , CP3 =OFF       ;Code Protection Block 0,1,2,3 -> OFF
    CONFIG  CPB =OFF , CPD =OFF                             ;Boot Block Code Protection, Data EEPROM
Code Protection-> OFF
    CONFIG  WRT0=OFF , WRT1=OFF , WRT2=OFF , WRT3=OFF       ;Write Protection Block 0,1,2,3 -> OFF
    CONFIG  WRTB=OFF , WRTC=OFF , WRTD=OFF                  ;Boot Block ,Configuration Register,Data
EEPROM Write Protection -> OFF
    CONFIG  EBTR0=OFF, EBTR1=OFF, EBTR2=OFF, EBTR3=OFF      ; Table Read Protection Block 0,1,2,3 ->
OFF
    CONFIG  EBTRB=OFF                                       ;Boot Block Table Read Protection -> OFF
;==============================================================================================
    END
```

File 5: ISR.asm

```
        CODE

HighPriorityISR
; **** Put High priority interrupt service routine code here ****
        RETFIE
LowPriorityISR
; **** Put Low priority interrupt service routine code here ****
        RETFIE

    GLOBAL  HighPriorityISR,LowPriorityISR

    END
```

File 6: Init.asm

```
Init:

;Write any initialization code here
    ;---------;


    GOTO StartRTOS

    GLOBAL  Init
    EXTERN  StartRTOS
    END
```

C version:

1. RTOSsetting.inc

```
#define TOTAL_TASKS     D'13'


#define OSCFREQ         D'20'       '


#define RoundRobin          1
#define RateMonotonic       2
#define Preemptive          3
#define SchedulingAlgorithm RoundRobin


#define Priority1
#define Priority2


#define Frequency1
#define Frequency2
```

2. RTOSMacros.inc file:

```
                        ;=================================================;
                        ;******************RTOS   MACROs********************;
                        ;=================================================;

                                    ;------------;
                                ;********Description********;
                                    ;------------;
;           This file contains all macros called in Kernel file
;           List of Macros:

;-------------------------------------------------------------------------------------------------------------
-------------------------------
;____________Macro Name_____________________|____________Macro
Description_____________________________________
;           1. AssignValue2Register:         |   assign <value> to <reg> at Any RAM mem.
;           2. AssignValue2AccessRegister:   |   assign <value> to <reg> at access RAM
;           3. SetRunningTask:               |   Set <value> to <RUNNINGTASK>    register
;           4. InitializeStack:              |   Initialize The -RTOS created- stacks of Tasks where
all important registers are saved
;           5. InitializeStacks:             |   Initialize All Tasks' Stacks -created by RTOS to
save important registers-
;           5. StoreContextIn:               |   Store Important Registers To Memory block where FSR0
is pointing to
;           6. ReStoreContextFrom:           |   ReStore Important Registers From Memory block where
FSR0 is pointing to
;           6. ReStoreContextFrom:           |   ReStore Important Registers From Memory block where
FSR0 is pointing to
;-------------------------------------------------------------------------------------------------------------
-------------------------------
                                ;~~~~~;This file is created by: Mohamed Tag 2009;~~~~~;

        variable    labelGenerator=1
        ;------------;
;********Description********;
        ;------------
        ;This Assembler variable is incremented whenever macro is called and then appended to labels
        ;This is to avoid lables duplication when macros are substituted in the main code where it's
called
        ;------------;
;******Description End******;
        ;------------

;01-----------------------------------------------------------------------------------------------------------
----------;
SetRunningTask:                                 MACRO           TaskNo
;***********************;
;-----------------------------------------------------------------------------------------------------------
----------;
                        MOVLW   TaskNo
                        MOVWF   RUNNINGTASK,ACCESS
                        ENDM
;01xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx  MACRO
xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx;
                                ;xxxxxxxxxxxxx END xxxxxxxxxxxxxxxxxxxxx;


;02-----------------------------------------------------------------------------------------------------------
----------;
InitializeStack:                            MACRO                   Stackaddress,TaskNumber
;***********;
;-----------------------------------------------------------------------------------------------------------
----------;
labelGenerator+=1
                        LFSR        0,Stackaddress
                        MOVLW       1                       ;FillHWstackPointer
                        MOVWF       POSTINC0                ;FillHWstackReturnAddress To
Tasks' starting addresses
                        MOVLW       upper Task#v(TaskNumber)    ;Move TOSU
                        MOVWF       POSTINC0
                        MOVLW       high Task#v(TaskNumber)     ;Move TOSH
                        MOVWF       POSTINC0
                        MOVLW       low Task#v(TaskNumber)      ;Move TOSL
                        MOVWF       POSTINC0
                        MOVFF       BSR,POSTINC0            ;BSR
                        ;NOTE: IMPORTANT REGISTERS (W,STATUS,FSR0) ARE SAVEVD IN temp MEMORY
(E.G. W->Wtemp)
                        ;THIS IS DONE WHEN RTOS INITIALIZATION IS ENTERED
                        ;THIS IS BECAUSE InitRTOS UTILIZE THOSE REGISTERS
                        MOVFF       Wtemp,POSTINC0          ;Wtemp,ACCESS
                        MOVFF       STATUStemp,POSTINC0     ;STATUStemp,ACCESS
```

```
                            MOVFF       PRODH,POSTINC0              ;PRODH,ACCESS
                            MOVFF       PRODL,POSTINC0              ;PRODL,ACCESS
                            MOVFF       FSR0Htemp,POSTINC0          ;FSR0Htemp,ACCESS
                            MOVFF       FSR0Ltemp,POSTINC0          ;FSR0Ltemp,ACCESS
                            MOVFF       FSR1H,POSTINC0              ;FSR1H,ACCESS
                            MOVFF       FSR1L,POSTINC0              ;FSR1L,ACCESS
                            MOVFF       FSR2H,POSTINC0              ;FSR2H,ACCESS
                            MOVFF       FSR2L,POSTINC0              ;FSR2L,ACCESS
                            MOVFF       TBLPTRU,POSTINC0            ;TBLPTRU,ACCESS
                            MOVFF       TBLPTRH,POSTINC0            ;TBLPTRH,ACCESS
                            MOVFF       TBLPTRL,POSTINC0            ;TBLPTRL,ACCESS
                            MOVFF       TABLAT,POSTINC0             ;TABLAT,ACCESS
                            MOVFF       PCLATU,POSTINC0             ;PCLATU,ACCESS
                            MOVFF       PCLATH,POSTINC0             ;PCLATH,ACCESS
                            MOVFF       PORTA,POSTINC0              ;PORTA,ACCESS
                            MOVFF       PORTC,POSTINC0              ;PORTC,ACCESS
                            MOVFF       FSR1H,Copytemp1             ; Save a copy of another FSR
                            MOVFF       FSR1L,Copytemp2
                            LFSR        1, _stack                   ;FSR1->SOFTWARE STACK
                            MOVLW       SWstackDepth
                            TSTFSZ      WREG,ACCESS
                            BRA         InitializeSWStack#v(labelGenerator)
                            BRA         SkipSWstack#v(labelGenerator)
InitializeSWStack#v(labelGenerator)                             ;To avoid the dublication of label
                            MOVFF       POSTINC1,POSTINC0
                            DECFSZ      WREG,W,ACCESS
                            GOTO        InitializeSWStack#v(labelGenerator)
SkipSWstack#v(labelGenerator)
                            MOVFF       Copytemp1,FSR1H             ; Restore the original value of
FSR
                            MOVFF       Copytemp2,FSR1L
                            ENDM
;02xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx  MACRO
xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx;
                                    ;xxxxxxxxxxxxx END xxxxxxxxxxxxxxxxxxxx;


;03-------------------------------------------------------------------------------------------------------
----------;
InitializeStacks:                                   MACRO
;*****************************;
;-------------------------------------------------------------------------------------------------------
----------;
                            LOCAL   TaskNumber
TaskNumber=1
                            WHILE   TaskNumber<=TOTAL_TASKS
                            InitializeStack STACK#v(TaskNumber),TaskNumber
TaskNumber+=1
                            ENDW
                            ENDM
;03xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx  MACRO
xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx;
                                    ;xxxxxxxxxxxxx END xxxxxxxxxxxxxxxxxxxx;


;04-------------------------------------------------------------------------------------------------------
----------;
InitializeStackPointersArray:                       MACRO
;************************************;
;-------------------------------------------------------------------------------------------------------
----------;
                            LFSR    0,StackPointersArray
                            LOCAL   StackNumber=1
                            WHILE   StackNumber <= TOTAL_TASKS
                            MOVLW   HIGH STACK#v(StackNumber)
                            MOVWF   POSTINC0,ACCESS
                            MOVLW   LOW STACK#v(StackNumber)
                            MOVWF   POSTINC0,ACCESS
StackNumber+=1
                            ENDW
                            ENDM
;04xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx  MACRO
xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx;
                                    ;xxxxxxxxxxxxx END xxxxxxxxxxxxxxxxxxxx;

;05-------------------------------------------------------------------------------------------------------
----------;
StoreContextIn:                                     MACRO
;************************************;
;-------------------------------------------------------------------------------------------------------
```

```
----------;
labelGenerator+=1
                        MOVFF           STKPTR,POSTINC0
StoreHWStack#v(labelGenerator):                         ;To avoid the dublication of
label
                        MOVFF           TOSU,POSTINC0
                        MOVFF           TOSH,POSTINC0
                        MOVFF           TOSL,POSTINC0
                        DECFSZ          STKPTR,F,A
                        GOTO            StoreHWStack#v(labelGenerator)
StoreImportantSFRs#v(labelGenerator):
;Important Registers are saved here
                        MOVFF           BSR,POSTINC0
                        MOVFF           Wtemp,POSTINC0
                        MOVFF           STATUStemp,POSTINC0
                        MOVFF           PRODH,POSTINC0
                        MOVFF           PRODL,POSTINC0
                        MOVFF           FSR0Htemp,POSTINC0
                        MOVFF           FSR0Ltemp,POSTINC0
                        MOVFF           FSR1H,POSTINC0
                        MOVFF           FSR1L,POSTINC0
                        MOVFF           FSR2H,POSTINC0
                        MOVFF           FSR2L,POSTINC0
                        MOVFF           TBLPTRU,POSTINC0
                        MOVFF           TBLPTRH,POSTINC0
                        MOVFF           TBLPTRL,POSTINC0
                        MOVFF           TABLAT,POSTINC0
                        MOVFF           PCLATU,POSTINC0
                        MOVFF           PCLATH,POSTINC0
                        MOVFF           PORTA,POSTINC0
                        MOVFF           PORTC,POSTINC0
                        MOVFF           FSR1H,Copytemp1 ; Save a copy of another FSR
                        MOVFF           FSR1L,Copytemp2
StoreSWstack#v(labelGenerator):
                        LFSR            1, _stack               ;Load SW stack start
address into FSR
                        MOVLW           SWstackDepth
                        TSTFSZ          WREG,ACCESS
                        BRA             StoreSWStack#v(labelGenerator)
                        BRA             SkipSWstackStoring#v(labelGenerator)

StoreSWStack#v(labelGenerator)                                  ;To avoid the
dublication of label
                        MOVFF           POSTINC1,POSTINC0
                        DECFSZ          WREG,W,ACCESS
                        GOTO            StoreSWStack#v(labelGenerator)
SkipSWstackStoring#v(labelGenerator):
StoreSharedSections#v(labelGenerator):
                        SaveSectionToFSR0       MATH_DATA
                        SaveSectionToFSR0       .tmpdata
                        MOVFF           Copytemp1,FSR1H ; Restore the original value of
FSR
                        MOVFF           Copytemp2,FSR1L
                        ENDM
;05xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx  MACRO
xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx;
                                ;xxxxxxxxxxxxx END xxxxxxxxxxxxxxxxxxxxx;

;06----------------------------------------------------------------------------------------------------
----------;
RestoreContextFrom:                             MACRO
;***********************************;
;----------------------------------------------------------------------------------------------------
----------;
labelGenerator+=1
                        MOVFF           POSTINC0,STKPTR
                        MOVFF           STKPTR,STKPTRtemp
RestoreHWStack#v(labelGenerator)
                        MOVF            POSTINC0,W
                        MOVWF           TOSU
                        MOVF            POSTINC0,W
                        MOVWF           TOSH
                        MOVF            POSTINC0,W
                        MOVWF           TOSL
                        DECFSZ          STKPTR,F,A
                        GOTO            RestoreHWStack#v(labelGenerator)
                        MOVFF           STKPTRtemp,STKPTR
                        MOVFF           POSTINC0,BSR
                        MOVFF           POSTINC0,Wtemp
                        MOVFF           POSTINC0,STATUStemp
```

```
                            MOVFF               POSTINC0,PRODH
                            MOVFF               POSTINC0,PRODL
                            MOVFF               POSTINC0,FSR0Htemp
                            MOVFF               POSTINC0,FSR0Ltemp
                            MOVFF               POSTINC0,FSR1H
                            MOVFF               POSTINC0,FSR1L
                            MOVFF               POSTINC0,FSR2H
                            MOVFF               POSTINC0,FSR2L
                            MOVFF               POSTINC0,TBLPTRU
                            MOVFF               POSTINC0,TBLPTRH
                            MOVFF               POSTINC0,TBLPTRL
                            MOVFF               POSTINC0,TABLAT
                            MOVFF               POSTINC0,PCLATU
                            MOVFF               POSTINC0,PCLATH
                            MOVFF               POSTINC0,PORTA
                            MOVFF               POSTINC0,PORTC
                            MOVFF               FSR1H,Copytemp1 ; Save a copy of another FSR
                            MOVFF               FSR1L,Copytemp2
                            LFSR                1, _stack               ;Load SW stack start address
into FSR
                            MOVLW               SWstackDepth
                            TSTFSZ              WREG,ACCESS
                            BRA                 ReStoreSWStack#v(labelGenerator)
                            BRA                 SkipReSWstack#v(labelGenerator)
ReStoreSWStack#v(labelGenerator):
                            MOVFF               POSTINC0,POSTINC1
                            DECFSZ              WREG,W,ACCESS
                            GOTO                ReStoreSWStack#v(labelGenerator)
SkipReSWstack#v(labelGenerator):
RestoreSharedSections#v(labelGenerator):
                            RestoreSectionFromFSR0  MATH_DATA
                            RestoreSectionFromFSR0  .tmpdata
RecoveringFSR1#v(labelGenerator):
                            MOVFF               Copytemp1,FSR1H ; Restore the original value of FSR
                            MOVFF               Copytemp2,FSR1L
                            ENDM
;06xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx  MACRO
xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx;
                                    ;xxxxxxxxxxxxx END xxxxxxxxxxxxxxxxxxxx;

;07----------------------------------------------------------------------------------------------------
----------;
SaveSectionToFSR0:                              MACRO                       SectionName
;***********;
;----------------------------------------------------------------------------------------------------
----------;
labelGenerator+=1
                            scnstart_lfsr       1, SectionName              ;Load FSR1 by Section
start Address
                            MOVLW               scnsz_low SectionName
                            TSTFSZ              WREG,ACCESS
                            BRA                 StoreSection#v(labelGenerator)
                            BRA                 SkipSectionSaving#v(labelGenerator)
StoreSection#v(labelGenerator):
                            MOVFF               POSTINC1,POSTINC0
                            DECFSZ              WREG,W,ACCESS
                            BRA                 StoreSection#v(labelGenerator)
SkipSectionSaving#v(labelGenerator)
                            ENDM
;07xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx  MACRO
xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx;
                                    ;xxxxxxxxxxxxx END xxxxxxxxxxxxxxxxxxxx;

;08----------------------------------------------------------------------------------------------------
----------;
RestoreSectionFromFSR0:                             MACRO                       SectionName
;***********;
;----------------------------------------------------------------------------------------------------
----------;
labelGenerator+=1
                            scnstart_lfsr       1, SectionName              ;Load FSR1 by Section
start Address
                            MOVLW               scnsz_low SectionName
                            TSTFSZ              WREG,ACCESS
                            BRA                 ReStoreSection#v(labelGenerator)
                            BRA                 SkipSectionRestoring#v(labelGenerator)
ReStoreSection#v(labelGenerator):
                            MOVFF               POSTINC0,POSTINC1
                            DECFSZ              WREG,W,ACCESS
                            BRA                 ReStoreSection#v(labelGenerator)
```

```
SkipSectionRestoring#v(labelGenerator)
                        ENDM
;08xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx  MACRO
xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx;
                                    ;xxxxxxxxxxxxx END xxxxxxxxxxxxxxxxxxxx;

;09-------------------------------------------------------------------------------------------------
----------;
SaveSectionPortionToFSR0:                       MACRO                   SectionName,Depth
;***********;
;-------------------------------------------------------------------------------------------------
----------;
labelGenerator+=1
                        scnstart_lfsr       1, SectionName              ;Load FSR1 by Section
start Address
                        MOVLW               Depth
                        TSTFSZ              WREG,ACCESS
                        BRA                 StoreSectionPortion#v(labelGenerator)
                        BRA                 SkipSectionPortionSaving#v(labelGenerator)
StoreSectionPortion#v(labelGenerator):
                        MOVFF               POSTINC1,POSTINC0
                        DECFSZ              WREG,W,ACCESS
                        BRA                 StoreSectionPortion#v(labelGenerator)
SkipSectionPortionSaving#v(labelGenerator)
                        ENDM
;09xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx  MACRO
xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx;
                                    ;xxxxxxxxxxxxx END xxxxxxxxxxxxxxxxxxxx;

;10-------------------------------------------------------------------------------------------------
----------;
RestoreSectionPortionFromFSR0:                      MACRO           SectionName,Depth
;***********;
;-------------------------------------------------------------------------------------------------
----------;
labelGenerator+=1
                        scnstart_lfsr       1, SectionName              ;Load FSR1 by Section
start Address
                        MOVLW               Depth
                        TSTFSZ              WREG,ACCESS
                        BRA                 ReStoreSectionPortion#v(labelGenerator)
                        BRA                 SkipSectionPortionRestoring#v(labelGenerator)
ReStoreSectionPortion#v(labelGenerator):
                        MOVFF               POSTINC0,POSTINC1
                        DECFSZ              WREG,W,ACCESS
                        BRA                 ReStoreSectionPortion#v(labelGenerator)
SkipSectionPortionRestoring#v(labelGenerator)
                        ENDM
;10xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx  MACRO
xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx;
                                    ;xxxxxxxxxxxxx END xxxxxxxxxxxxxxxxxxxx;

;11-------------------------------------------------------------------------------------------------
----------;
IncrementKernelCounter:                             MACRO
;*********************************;
;-------------------------------------------------------------------------------------------------
----------;
;C18 makes unsigned int data type aligned like: L:H
        infsnz  KernelCounter,F     ;Increment Low Byte of Kernel Counter
        incf    KernelCounter+1,F   ;Increment High Byte of Kernel Counter if low one is overflown
        endm
;11xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx  MACRO
xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx;
                                    ;xxxxxxxxxxxxx END xxxxxxxxxxxxxxxxxxxx;
```

3. ROTSdeclarations.inc file

```
                        ;================================================;
                           ;*************RTOS Assember Header file*****;
                        ;================================================;

                                        ;------------;
                                    ;********Description********;
                                        ;------------;
;           This file contains all declarations needed by the Kernel
;           List of Macros:
;-------------------------------------------------------------------------------------------------------------
-------------------------------
;_____________Section Name____________________|_________________
Description_____________________________________________
;           1. RTOStemporaryRegisters:      |   Declare temporary registers needed for data movement
process
;           2. TasksDeclaration:            |   Import Tasks' names and make default values for them
;           3. StacksDeclaration:           |   Declare Stacks Required to save vulneralbe registers
of each Task
;           4. TasksCounter:                |   Declare Counters for each task (required for time
calculations)
;           4. TasksState:                  |   Declare State register for each Task
;-------------------------------------------------------------------------------------------------------------
-------------------------------
                                    ;~~~~~;This file is created by: Mohamed Tag 2009;~~~~~;

#include    "RTOSsettings.inc"

;01-----------------------------------------------------------------------------------------------------------
----------;
RTOStemporaryRegisters                  UDATA_ACS
;************************;
;-----------------------------------------------------------------------------------------------------------
----------;
Wtemp           RES         1
STKPTRtemp      RES         1
STATUStemp      RES         1
RUNNINGTASK     RES         1
FSR0Htemp       RES         1
FSR0Ltemp       RES         1
Copytemp1       RES         1
Copytemp2       RES         1
    GLOBAL  RUNNINGTASK         ;Globalize this variables
;01xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx  SECTION
xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx;
                                    ;xxxxxxxxxxxxx END xxxxxxxxxxxxxxxxxxxx;

;02-----------------------------------------------------------------------------------------------------------
----------;
TasksDeclaration
;************************;
;-----------------------------------------------------------------------------------------------------------
----------;
; Define the tasks priorities here (priorities : 0-255)
;Task1priority  DB      D'1'
;Task2priority  DB      D'1'
    VARIABLE    TaskNumber=1
    WHILE   TaskNumber <= TOTAL_TASKS
_Task#v(TaskNumber)priority     code
Task#v(TaskNumber)priority  DB      #v(TaskNumber)
    GLOBAL  Task#v(TaskNumber)priority
;Export the Tasks to all files
    EXTERN  Task#v(TaskNumber)
TaskNumber+=1
    ENDW
;02xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx  SECTION
xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx;
                                    ;xxxxxxxxxxxxx END xxxxxxxxxxxxxxxxxxxx;

;03-----------------------------------------------------------------------------------------------------------
----------;
StacksDeclaration
;************************;
;-----------------------------------------------------------------------------------------------------------
----------;
#DEFINE     HWstackDepth            D'10'
#DEFINE     MATHDATASIZE            D'2'
#DEFINE     DELAYDATSIZE            D'0'    ;DELAYDAT1,DELAYDAT2:DelayCounter1,DelayCounter2
#DEFINE     TMPDATASIZE             D'2'    ;:__tmp_0
#DEFINE     SWstackDepth            D'20'
#DEFINE     STACKSIZE   D'1'+ D'3'*HWstackDepth +D'1'+D'1'+D'1'+ D'12' + D'2' + SWstackDepth
```

```
+MATHDATASIZE+DELAYDATSIZE+TMPDATASIZE

; About STACKSIZE:
; The stack size depends on how many registers are required to be saved
; For 1:STKPTR,31*3:Stack Spaces, 1:BSR, 1:STATUS, 1:WREG, 12:extra
; 9 SFRs Extention: PRODH-L (2 SFRs), FSR0H-L,FSR1H-L,FSR2H-L(6 SFRs), TBLPTR(U-H-L) (3 SFRs),
TABLAT (1)
; 2: PCLATU, PCLATH
; SWstackDepth: How many bytes are required to be saved from the software stack (This SW is
implemented by C18 compiler)

;Reserve Space for stacks depending on the definition of tasks
;Declaring Scratch memory section for storing task1 variables when task switching
;Stack size = 'STACKSIZE' Byte

;Reserve Space for stacks depending on the definition of tasks
    VARIABLE    TaskNumber=1
    WHILE   TaskNumber<=TOTAL_TASKS
_STACK#v(TaskNumber)    UDATA                   ;Declaring Scratch memory section for storing task1
variables when task switching
STACK#v(TaskNumber)     RES         STACKSIZE
TaskNumber+=1
    ENDW


;   Declare Stacks' pointers
;Reserve Space for stack pointers depending on the definition of tasks
_StackPointersSpace     UDATA
StackPointersArray      RES         (TOTAL_TASKS*2)

;03xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx  SECTION
xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx;
                                        ;xxxxxxxxxxxxx END xxxxxxxxxxxxxxxxxxxxx;

;04----------------------------------------------------------------------------------------------------
----------;
TasksCounters
;**********************;
;----------------------------------------------------------------------------------------------------
----------;
#DEFINE     CounterSize     2               ;Support up to 2^16 = 65'000

;Counter Continuously incremented (Every TIMER0 Interrupt)!
_KernelCounter              UDATA_ACS
KernelCounter               RES         CounterSize
    Global                  KernelCounter

    VARIABLE                TaskNumber=1
_TasksCounters              UDATA               ;Declaring Counters for each Task
    WHILE                   TaskNumber<=TOTAL_TASKS
Counter#v(TaskNumber)       RES         CounterSize

    GLOBAL                  Counter#v(TaskNumber)
TaskNumber+=1
    ENDW
;04xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx  SECTION
xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx;
                                        ;xxxxxxxxxxxxx END xxxxxxxxxxxxxxxxxxxxx;

;05----------------------------------------------------------------------------------------------------
----------;
TasksStates
;**********************;
;----------------------------------------------------------------------------------------------------
----------;
#DEFINE     READY       1
#DEFINE     RUNNING     2
#DEFINE     SUSPENDED   3
#DEFINE     TERMINATED  4

TaskNumber=1
_TasksStates                UDATA_ACS               ;Declaring State for each Task
    WHILE                   TaskNumber<=TOTAL_TASKS
State#v(TaskNumber)         RES     1
    GLOBAL                  State#v(TaskNumber)
TaskNumber+=1
    ENDW

;05xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx  SECTION
xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx;
```

4. RTOSkernel.asm

```
;===============================================================================================;
                                ;Include Files;
;===============================================================================================;
        #include    <p18cxxx.inc>
        #include    "RTOSdeclarations.inc"
        #include    "RTOSMacros.inc"
        EXTERN  LowPriorityISR,HighPriorityISR, _stack
;xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx;
                    ;xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx;

;===============================================================================================;
                                ;Interrupt Vectors High, Low;
                           ;Note: STATUS->STATUStemp & WREG->Wtemp for High Interrupt;
                               ;Low Interrupt Vector to LowInterruptService;
;===============================================================================================;
InterruptHighV  CODE        0x08
SaveWREGandSTATUSregisters:
                MOVFF       STATUS,STATUStemp   ; Save STATUS and WREG before entering ISR
                MOVWF       Wtemp,ACCESS
                goto        HighInterruptServiceRoutine

InterruptLowV   CODE        0x18
SaveWREGandSTATUSRegisters:
                goto        LowPriorityISR
;xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx;
                    ;xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx;


;=============================================================================================================
=========;
                                        ;RTOS INITIALIZATION STARTS HERE;
;=============================================================================================================
=========;
InitRTOS:
    GLOBAL  InitRTOS
;_____________________________________________________________________________;
BACKupImportantRegisters:               ; WHY THIS IS DONE?: ONCE C18 FINISH ITS INITIALIZATION, WE
NEED TO
                                        ; INITIALIZE TASKS SO THAT THEY FEEL THERE IS NONE BESIDE
THEM
                                        ; I.E. JUST IF THERE IS ONE TASK ONLY
                                        ; SO , WHAT EVER CHANGES HAPPEN TO OUR IMPORTANT REGISTERS
DURING RTOS STARTUP
                                        ; HAS TO BE RESTORED WHEN TASKS ARE CALLED (FOR THE FIRST
TIME)
                                        ; NOTE: only STATUS, WREG and FSR0 are affected during RTOS
initialization
                                        ;
        MOVFF       STATUS,STATUStemp   ; Save STATUS and WREG before entering RTOS
        MOVWF       Wtemp,ACCESS
BackupFSR0register:
        MOVFF       FSR0H,FSR0Htemp
        MOVFF       FSR0L,FSR0Ltemp

;_____________________________________________________________________________;

InitKernelTimer:
SetTimer0:
DisableGlobalInt:
        MOVLW       0x20            ;disable global and enable TMR0 interrupt
        MOVWF       INTCON,A
SetTMR0asHighPriorityInt:
        MOVLW       0x84            ;TMR0 high priority
        MOVWF       INTCON2,A
EnablePriorityLevel:
        BSF         RCON,IPEN,A     ;enable priority levels
        CLRF        TMR0H,A         ;clear timer
        CLRF        TMR0L,A         ;clear timer
Set1msTMR0:
        MOVLW       B'11000100'
        MOVWF       T0CON           ;set up timer0 - prescaler 1:16
EnableGlobalInt:
        BSF         INTCON,GIEH,A   ;enable interrupts

;_____________________________________________________________________________;


;_____________________________________________________________________________;
InitializeAllStacksOFtheTASKs:
        InitializeStacks
```

```
        InitializeStackPointersArray

;______________________________________________________________________;
ClearKernelCounter:
        CLRF        KernelCounter,ACCESS    ;Clear High Byte
        CLRF        KernelCounter+1,ACCESS  ;Clear Low  Byte

;______________________________________________________________________;
ReturnFSR0register:
;Return Important Registers Initial Values
        MOVFF   FSR0Htemp,FSR0H
        MOVFF   FSR0Ltemp,FSR0L
ReturnSTATUSwregRegister:
        MOVFF       STATUStemp,STATUS   ; Restore STATUS and WREG before LEAVing ISR
        MOVFF       Wtemp,WREG

;______________________________________________________________________;
CallTask1:
        SetRunningTask              1               ;Task1 is running

;______________________________________________________________________;
LaunchFirstTask:                                    ;Launch first task
        GOTO                        Task1           ;execute Task1
;xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx;
                    ;xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx;


;====================================================================================================
=========;
                                    ;High Priority Interrupt Service ;
                                        ;Kernel Code;
;====================================================================================================
=========;
HighInterruptServiceRoutine:


CheckInterruptSource:
        BTFSS   INTCON,TMR0IF,A     ;check for TMR0 overflow
        GOTO    HighPriorityISR

Timer0ISR:
;-----------;
TheKernel:  ;*******;
;-----------;
DisableGlobalInterrupt:
        BCF     INTCON,GIEH,ACCESS  ;disable interrupts

RechargeTimer:
        BCF     INTCON,TMR0IF,A     ;clear interrupt flag
        MOVLW   D'100'              ;TMR0L = 256-156 -> for 1 mSec
        MOVWF   TMR0L
        ;====================;

;---------------;
UpdateKernelCounter:
        IncrementKernelCounter
;---------------;


InvokeAlarmsManager:
        GOTO    AlarmsManager

;---------------;
AlarmsManager:  ;*******;
;---------------;

InvokeTaskScheduler:
        GOTO    TaskScheduler

;---------------;
TaskScheduler:  ;*******;
;---------------;

InvokeTaskSwitcher:
        GOTO    TaskSwitcher

;---------------;
TaskSwitcher:   ;*******;
;---------------;
```

```
BackupFSR0registers:
        MOVFF   FSR0H,FSR0Htemp
        MOVFF   FSR0L,FSR0Ltemp

DetermineRunningTask:
        DECF        RUNNINGTASK,W,ACCESS
PointToSTACKofRUNNINGtask:
        RLNCF       WREG,W,ACCESS
        LFSR        0,StackPointersArray
        MOVFF       PLUSW0,Copytemp1
        INCF        WREG,W,ACCESS
        MOVFF       PLUSW0,Copytemp2
        MOVFF       Copytemp1,FSR0H     ;Let FSR0H points to The Required Stack High address
        MOVFF       Copytemp2,FSR0L     ;Let FSR0L points to The Required Stack Low address
SaveCurrentTaskContext:
        StoreContextIn                      ;Store context to Current Task Stack
IncrementRunningTask:
        INCF        RUNNINGTASK,F,ACCESS    ;Increment RUNNINGTASK so It Points to Next Task
        MOVLW       TOTAL_TASKS
        CPFSGT      RUNNINGTASK,ACCESS
        BRA     ContinueWithNextTask
        SetRunningTask  1
ContinueWithNextTask:
        DECF        RUNNINGTASK,W,ACCESS
PointToSTACKofNEXTtask:
        RLNCF       WREG,W,ACCESS
        LFSR        0,StackPointersArray
        MOVFF       PLUSW0,Copytemp1
        INCF        WREG,W,ACCESS
        MOVFF       PLUSW0,Copytemp2
        MOVFF       Copytemp1,FSR0H     ;Let FSR0H points to The Required Stack High address
        MOVFF       Copytemp2,FSR0L     ;Let FSR0L points to The Required Stack Low address
        RestoreContextFrom

RestoreBackupRegisters:
RestoringFSR0registers:
        MOVFF   FSR0Htemp,FSR0H
        MOVFF   FSR0Ltemp,FSR0L
RestoringWREGandSTATUSregisters:
        MOVFF       STATUStemp,STATUS   ; Restore STATUS and WREG before LEAVing ISR
        MOVFF       Wtemp,WREG

FinishSwitching:
EnableGlobalIntBack:
        BSF         INTCON,GIEH,ACCESS  ;enable interrupts
EscapeTimer0ISR:
        RETFIE  0


        END                 ; end of assembly language program
```

5. ISR.c file

```
#pragma interrupt HighPriorityISR
void HighPriorityISR (void)
{
    // Write your ISR here (for high priority interrupt)
}

#pragma interruptlow LowPriorityISR
void LowPriorityISR (void)
{
    // Write your ISR here (low priority interrupt)
}
```

6. main.c file

```
#include    <p18f458.h>
#include    "D:\codes\IO\IO.h"  //FUNCTIONs FOR IO Initialization as IO


extern void InitRTOS (void);
extern void Task1 (void);

void __init (void);

void __init (void)
{
    InitializeIO();
}

void main (void)
{
    __init();
//  _asm bra Task1 _endasm
    _asm goto InitRTOS _endasm
}
```