A&QT-R 2002 (THETA 13)
2002 IEEE-TTTC International Conference on Automation, Quality and Testing, Robotics
May 23-25, 2002, Cluj-Napoca, Romania

A SIMPLE MULTI-TASKING OPERATING SYSTEM

Alin MURESAN, Dan CIMPOCA

NOKIA NETWORKS OY, P.O. Box 330, FIN-00045 Nokia Group, FINLAND
Ext-Alin.Muresan@nokia.com, Ext-Dan.Cimpoca@nokia.com

Abstract: The main idea of implementing a simple **M**ulti-**T**asking **O**perating **S**ystem (MTOS) was to take full advantages of the hardware resources available in simple devices like the mid-range family of microcontrollers from Microchip. The implementation described in this paper makes the enhanced FLASH/EEPROM 8 bit microcontroller PIC16F84A a powerful tool for a broad range of applications.

Keywords: multi-tasking, operating system, microcontroller, process

1 INTRODUCTION

The increased computing power and controlling of several devices is difficult to achieve with the sequential programming style. This issue becomes obvious when using low power, low resources microcontrollers. The approach of multi-tasking operating systems is the logical solution for the problem. Such a system is described in Application Note AN777, available at www.microchip.com. The application uses Salvo™ RTOS (Real-Time Operating System) v2.1, is written in C language using HI-TECH C compiler and runs on a PIC16F877 (high-range microcontroller) [2].

The application described in present paper is written in assembly language, using license-free tools, freely available on Microchip's website (MPASM v02.20 as assembler and MPSIM 5.20 as simulation tool).

2 OPERATING SYSTEM'S FEATURES

The application presented implements a cooperative multi-tasking operation system that runs on a PIC16F84A microcontroller, having the following resources:

- Operating speed: max. 20 MHz clock input;
- 13 I/O lines;
- 1024 words (x14 bits) of program memory;
- 68 bytes of Data RAM;
- 64 bytes of Data EEPROM;
- One 8-bit timer/counter with 8-bit programmable prescaler.

Scheduling is based upon processes' dynamic priorities. The timer interrupt handler is responsible for updating priorities (Figure 3) and the Process Manager for scheduling (Figure 4).

In the current implementation the MTOS can handle up to 8 tasks (processes). Processes' code is located in the upper 256 words of the program memory. Process0 starts at address 300H, process1 at 320H, process2 at 340H etc. This arrangement was chosen to simplify the context switching mecanism and to reduce the amount of memory used to retrieve context specific data. Each process has associated 6 bytes of data, first 4 residing in EEPROM data memory and the other 2 in RAM data memory. Table 1 describes the use of each byte.

BYTE	LOCATION	USE
Byte 1	EEPROM	Lower byte of the process' address (e.g. 20H for process1)
Byte 2	EEPROM	Initial waiting time
Byte 3	EEPROM	Initial priority
Byte 4	EEPROM	Reserved for future development
Byte 5	RAM	Current waiting time
Byte 6	RAM	Current priority

Table 1

During the initialisation phase, the initial waiting time and initial priority of each process are loaded from EEPROM to RAM (Byte2 -> Byte5, Byte3 -> Byte6). After executing its specific job, each process will load its own data from EEPROM to RAM.

TMR0 is programmed to issue an interrupt every 1.6 miliseconds (using timer mode and the prescaler). This is the operating system's clock (TICK). The TMR0 is incremented with prescaler rate (P-TICK). The initial waiting time is given in integer multiples (0..255) of TICK, hence the maximum waiting time for a process could be 409.6 miliseconds. The priority range is also 0..255.

In future implementations, Byte4 might be used to store the execution time of the process in P-TICKs. This way the Process Manager (see Figure 2) can check if the process being just about to execute can run without being interrupted by TMR0 (in case of critical processes). The process can be scheduled for a safe execution later, with the highest priority. An executing process can be interrupted only by TMR0 overflow interrupt, or by an external interrupt (if enabled). After executing its job, the process gives the control of the processor to Process Manager.

3 HARDWARE AND SOFTWARE ARCHITECTURES

3.1 *Hardware Architecture*

The system is built on a prototyping circuit board. Currently, no external controlled devices are connected (see Figure 1).

The use of integrated oscillator leaves the pin OSC2/CLKOUT not connected. This pin can be used for timing purposes in external devices, as a square wave signal with frequency $f_{OSC}/4$ (5 MHz in our case) is available.

The microcontroller is connected to a PC via RS232 interface. It uses RA4/T0CK1 (open drain, hence the resistor connected to V_{dd}) pin as transmitter. Commands sent from PC are received on pin RB0/INT. The start bit of RS232 protocol interrupts the microcontroller, and then the external interrupt handler is polling the pin to receive the incoming data. The communication is done at 115200 bauds.

TC232CPE is used as a TTL/RS232 and RS232/TTL levels adapter.

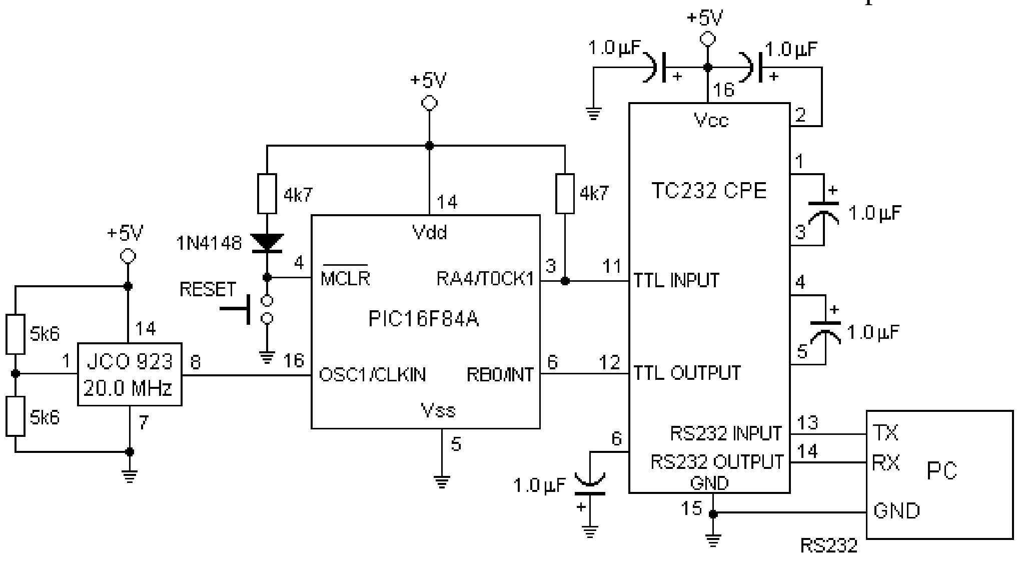


Figure 1

3.2 *Software Architecture*

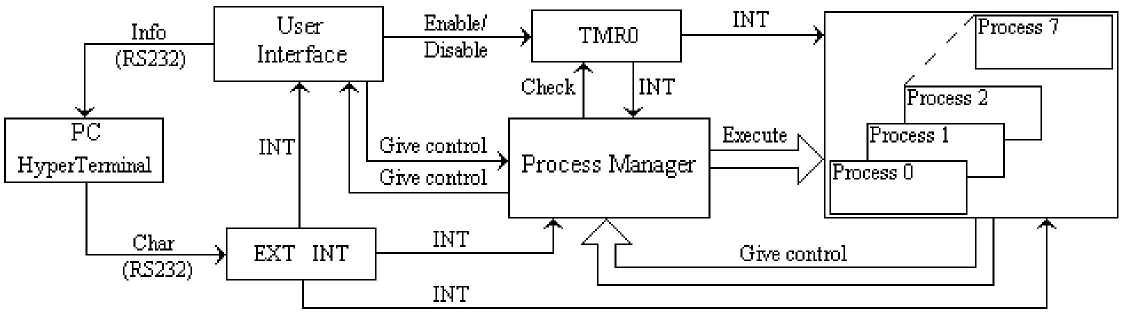


Figure 2

The current implementation gives the possibility of monitoring processor's resources (EEPROM, RAM, SFRs – special function registers, see Figure 5) and processes execution. Each process has associated an 'executable flag'. This way a process can be executed once, continuous, or under the full control of Process Manager (Figure 6). The flow charts of TMR0 interrupt handler and Process Manager, for multi-tasking execution, are depicted in Figure 3 and Figure 4.

The commands that can be sent from HyperTerminal (ASCII characters) are presented in Table 2. The User Interface interprets them (it knows that a character was received by checking COM flag – see Table 3) and takes appropriate action.

The register that controls operating system's execution is called 'os_flags'. Its bits are described in Table 3. The register can be written using commands sent from HyperTerminal program.

CHAR (ASCII)	ACTION
0..9, A..F	Hex numbers are stored in a ring buffer (size 4), used to write a value to a location in RAM data memory
e	Display EEPROM data memory
s	Display SFR (Special Function Registers)
g	Display GPR (General Purpose Registers – RAM)
w	Write to a RAM location
j	Jump –> gives control to Process Manager

Table 2

BIT	NAME	DESCRIPTION
0	PRI0	First 3 bits store the **PR**ocess **I**ndex: xxxxx000 -> process 0,… xxxxx111 -> process 7 Used only in tracing a certain process.
1	PRI1	
2	PRI2	
3	FIRST	Used to check first executable process.
4	GP	General purpose flag. Used in user interface and in TMR0 interrupt to signal to Process Manager that an interrupt occurred
5	CONT	Execution mode in tracing processes: 0 – execute once; 1 – execute continuous
6	TR	Used to trace 1 process: 0 - no trace; 1 – trace process
7	COM	Used to signal the receiving of a character: 0 - no char received; 1 - char received

Table 3

The system will execute in multi-tasking mode if command ‘j’ is received and TR bit is ‘0’.

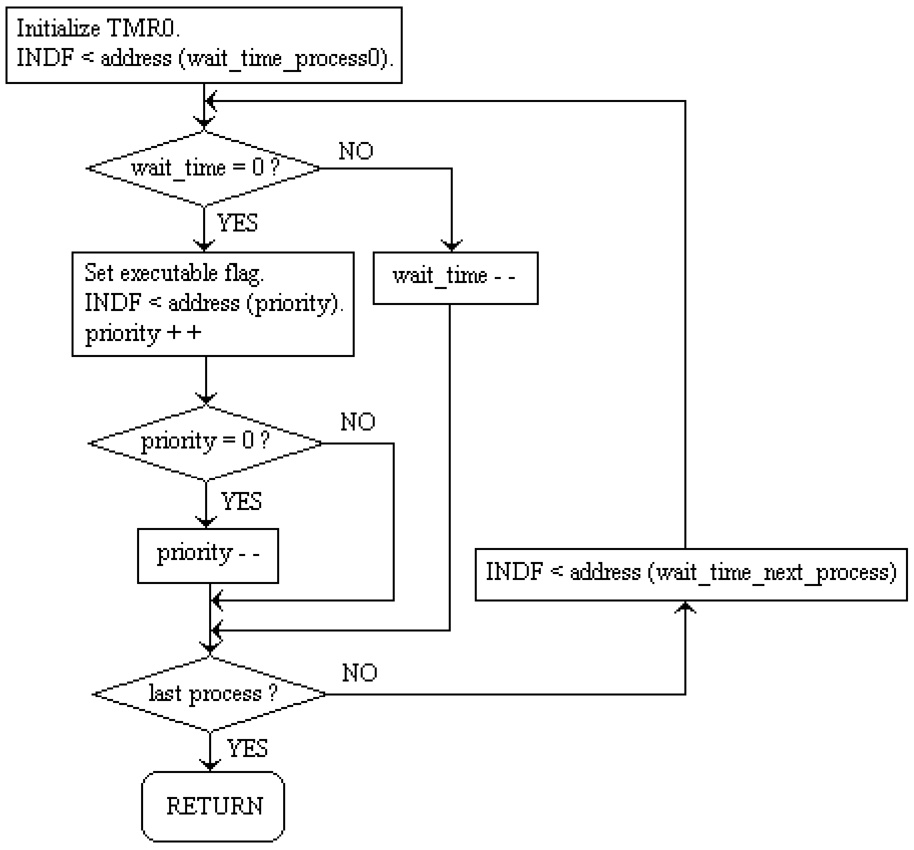


Figure 3

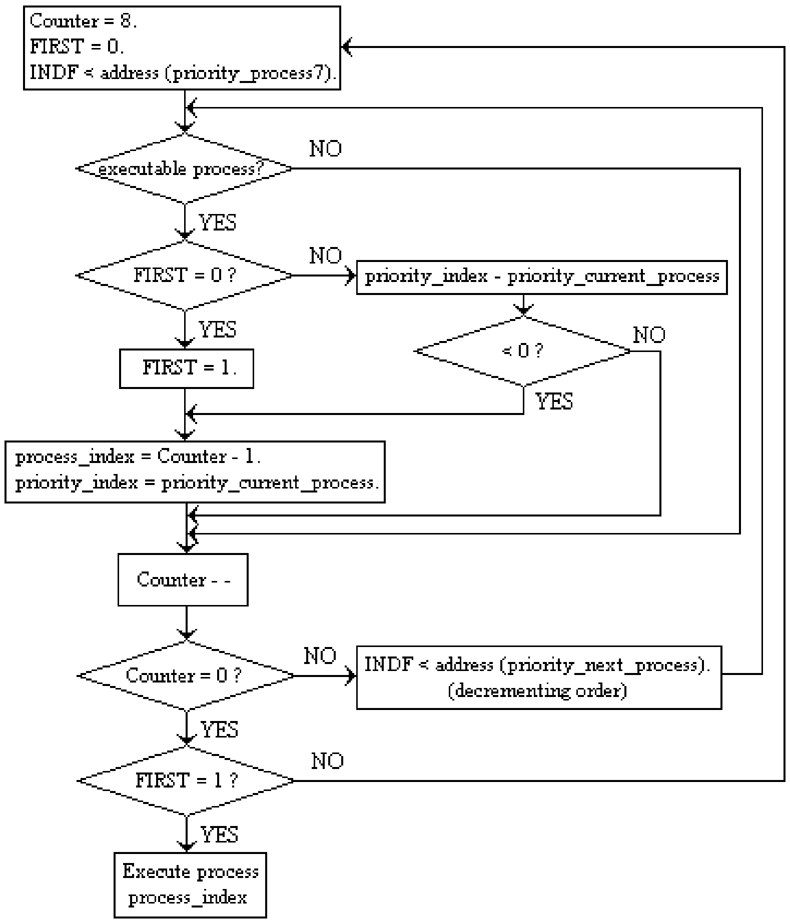

Counter = 8.
FIRST = 0.
INDF < address (priority_process7).
executable process?
NO
YES
FIRST = 0 ?
NO
priority_index - priority_current_process
YES
FIRST = 1.
< 0 ?
NO
YES
process_index = Counter - 1.
priority_index = priority_current_process.
Counter - -
Counter = 0 ?
NO
INDF < address (priority_next_process).
(decrementing order)
YES
FIRST = 1 ?
NO
YES
Execute process
process_index

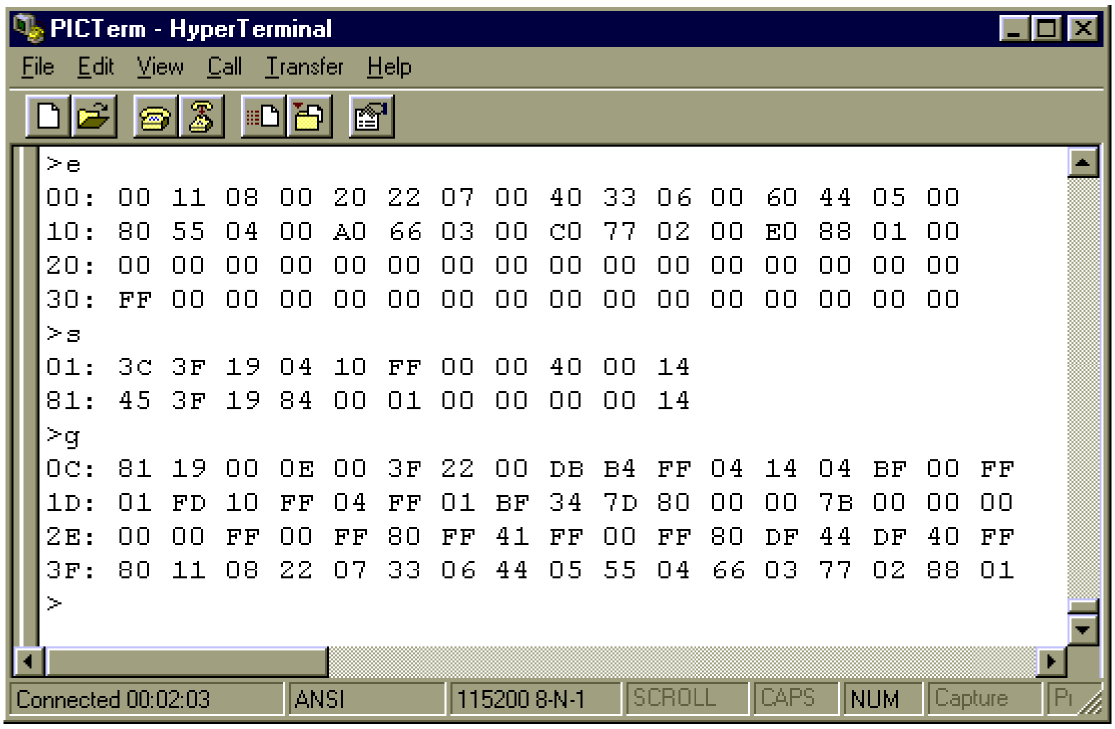
Figure 4

Reffering to Figure 4, the main loop executes multiple of 8 times (each process is checked), until the Process Manager detects an executable process (FIRST = 1). The ‘process_index’ and ‘priority_index’ variables store the index and the priority of the possible candidate to execution. If another process has a higher priority (checked by substraction ‘priority_index’ – ‘priority_current_process’) it becomes the candidate to execution. Indirect addressing is used (INDF register) for process checking.

Figure 5 shows the response of the User Interface to the monitoring commands (displays EEPROM, SFR, GPR). First 32 bytes of the EEPROM contain processes’ data. First 3 bytes represent address, waiting time and the priority of process0, starting at address 4 is data for process1 etc. One might notice that process0 has waiting time 11 and priority 8, process1 has waiting time 22 and priority 7 … process7 has waiting time 88 and priority 1. Figure 6 is a screenshot of the multi-tasking operation. Each process sends its own index value to the HyperTerminal.

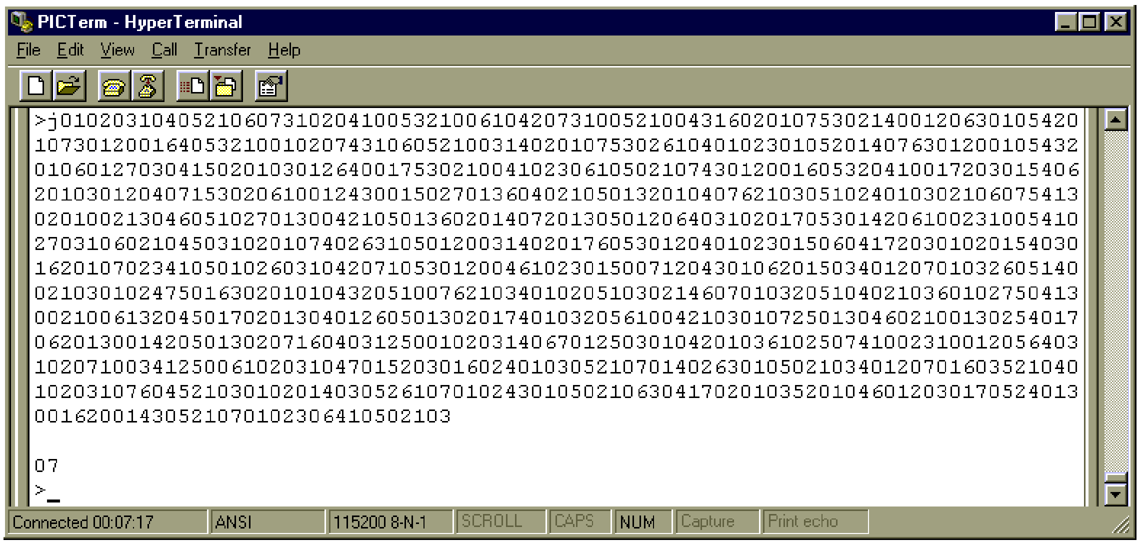
4 CONCLUSION

The full architecture (Figure 2, debugging features included) uses 35 bytes of RAM and 593 words of program memory. The basic architecture (Process Manager, TMR0 interrupt handler and each process executing “NOP” instruction) uses 26 bytes of RAM and 198 words of program memory. The implementation is well suited in applications that involve several controlled devices or periodical actions like the sensorial systems and actuators control in robotic applications. Future releases will include a semaphore-based communication among processes. The extension to 16 processes is straightforward as EEPROM and RAM memories support this.



```
>e
00: 00 11 08 00 20 22 07 00 40 33 06 00 60 44 05 00
10: 80 55 04 00 A0 66 03 00 C0 77 02 00 E0 88 01 00
20: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
30: FF 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
>s
01: 3C 3F 19 04 10 FF 00 00 40 00 14
81: 45 3F 19 84 00 01 00 00 00 00 14
>g
0C: 81 19 00 0E 00 3F 22 00 DB B4 FF 04 14 04 BF 00 FF
1D: 01 FD 10 FF 04 FF 01 BF 34 7D 80 00 00 7B 00 00 00
2E: 00 00 FF 00 FF 80 FF 41 FF 00 FF 80 DF 44 DF 40 FF
3F: 80 11 08 22 07 33 06 44 05 55 04 66 03 77 02 88 01
>
```

Figure 5



```
>j0102031040521060731020410053210061042073100521004316020107530214001206301054201
0730120016405321001020743106052100314020107530261040102301052014076301200105432
0106012703041502010301264001753021004102306105021074301200160532041001720301540 6
2010301204071530206100124300150270136040210501320104076210305102401030210607541 3
0201002130460510270130042105013602014072013050120640310201705301420610023100541 0
2703106021045031020107402631050120031402017605301204010230150604172030102015403 0
1620107023410501026031042071053012004610230150071204301062015034012070103260514 0
0210301024750163020101043205100762103401020510302146070103205104021036010275041 3
0021006132045017020130401260501302017401032056100421030107250130460210013025401 7
0620130014205013020716040312500102031406701250301042010361025074100231001205640 3
1020710034125006102031047015203016024010305210701402630105021034012070160352104 0
1020310760452103010201403052610701024301050210630417020103520104601203017052401 3
001620014305210701023064105021 03

07
>_
```

Figure 6

5 REFERENCES

[1] Microchip Technology Inc., (2001), PIC16F84A Data Sheet, www.microchip.com
[2] Chris Valenti, Andrew E. Kalman, (2001), AN777 – Multi-Tasking on the PIC16F877 with the Salvo™ RTOS, www.microchip.com