EE247: On Multitasking

March 23, 2020

Background:

The Wilkes university course EE247 Programming for Embedded Systems is concerned with operating computers that interact with the real world as a part of systems. Embedded computing typically takes place in the context of a system that may be large or small, but the computer is only a small part of it, though typically an important part. Examples of such systems include such things as microwave ovens, automotive engine controls, navigation systems, and even things as small and trivial as the control of an electric hand held drill. The role of the embedded computer is typically to control things, often including interactions with a human user through a display and perhaps a touchscreen or even a keyboard. At the same time, the computer must control physical things using motors, solenoids, light beams, and other actuators, and sense things in the environment such as temperature, speeds of things, pressure, etc.

Just like human beings, an embedded computer usually must be able to "multitask", that is, do more than one thing at the same time. However, digital computers are inherently "single task" in that they execute instructions sequentially, one instruction at a time. (Modern computers have tricks to speed things up by doing more than one instruction at a time, but they still conform to a model that looks like it is one at a time.) So, how do we get inherently single instruction stream computers to be able to do multiple things simultaneously? That is, multitask?

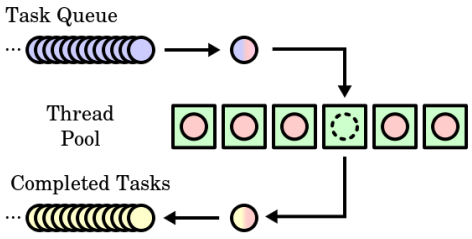
The basic concept is similar to what in Digital Design is called "multiplexing". Time is divided, and different slices of time are devoted to different tasks. How to do that for digital computers is a complex, but interesting, issue. Doing it in a "real time" environment, where tasks face deadlines, and there is a sense of time flowing against which tasks of different priorities must be done, is even more challenging. But, that's what we want to be able to do.

Some Definitions and Discussion:

If we are going to do "multitasking", it helps to define some of the terms we will be using. I'm drawing definitions from Wikipedia (readily available) and then will narrow them somewhat for our use in this class:

Task (computing)

From Wikipedia, the free encyclopedia



In computing, a **task** is a unit of execution or a unit of work. The term is ambiguous; precise alternative terms include *process*, light-weight process, *thread* (for execution), *step*, *request*, or *query* (for work). In the adjacent diagram, there are queues of incoming work to do and outgoing completed work, and a thread pool of threads to perform this work. Either the work units themselves or the threads that perform the work can be referred to as "tasks", and these can be referred to respectively as requests/responses/threads, incoming tasks/completed tasks/threads (as illustrated), or requests/responses/tasks. (The diagram shows A sample thread pool (green boxes) with task queues of waiting **tasks**(blue) and completed **tasks** (yellow), in the sense of task as "unit of work".)

I'm going to narrow this somewhat to say that a task is, yes, a single unit of work, but one that it is executed once and completes. In our context, this is like a "command" that is invoked by the user typing in a command in PuTTY. The command "dispatches" a task, the unit of work,

to carry out the "task" that needs to be done, such as toggle an LED or motor on or off, or adjust the speed of a motor, or return a reading for a photosensor. More specifically, the "task" is executed by the command callback function. A command callback function that is executing is an active task, or the "instance" of the task that is executing. A callback function that is not executing is a potential task that could be dispatched (instance created) sometime in the future.

Process (computing)

From Wikipedia, the free encyclopedia

In computing, a **process** is the instance of a computer program that is being executed by one or many threads. It contains the program code and its activity. Depending on the operating system (OS), a process may be made up of multiple threads of execution that execute instructions concurrently.[1][2]

While a computer program is a passive collection of instructions, a process is the actual execution of those instructions. Several processes may be associated with the same program; for example, opening up several instances of the same program often results in more than one process being executed.

Multitasking is a method to allow multiple processes to share processors (CPUs) and other system resources. Each CPU (core) executes a single task at a time. However, multitasking allows each processor to switch between tasks that are being executed without having to wait for each task to finish. Depending on the operating system implementation, switches could be performed when tasks perform input/output operations, when a task indicates that it can be switched, or on hardware interrupts.

A common form of multitasking is time-sharing. Time-sharing is a method to allow high responsiveness for interactive user applications. In time-sharing systems, context switches are performed rapidly, which makes it seem like multiple processes are being executed simultaneously on the same processor. This seeming execution of multiple processes simultaneously is called concurrency.

This definition of "process" is focused on general purpose computing where multiple "programs" are operating. We really have only one "program" our application, or perhaps more precisely, the executable (program) that we have downloaded and installed onto our microcontroller by "programming" (in a different sense) its FLASH memory (equivalent to ROM). So, we are going to again narrow the definition of "process" to mean a continuously operating ongoing task (or series of tasks) that does not run to completion immediately. Typically a process is periodic, or responds to situations. From what we have been doing, a "process" would be to blink an LED at a constant frequency, run a stepper motor, or maintain a clock. We have been doing these things, we just had not identified what we were doing formally as a "process" before. (In a general purpose computer, the computer may indeed be running multiple "programs" each of which has one or more processes going to do things like maintain a user interface, communicate by Ethernet with a remote site, monitor an input device like a mouse, and perhaps debug a microcontroller.)

Notice the word "instance". A process may be running, or not, but it is actually the "instance" of a task that runs. In our case we won't normally be having multiple instances of the same process executing. But, suppose you have a function that starts up and runs a stepper motor. You could conceivably have two invocations of that function, one for stepper motor A attached to PTB3 and PTB4, and another for stepper motor B attached to PTB6 and PTB7. Both could use the same code (if it was general enough, and reentrant) with data within each (stored in

private or local variables) that indicated which output pins were being used for each. (Have you ever, in Windows, clicked an application twice, perhaps in annoyance of the delay coming up, and then found that you had two "instances" of that application running, when you only wanted and needed one? That's an illustration of multiple instances. You could conceivably work on two different projects at the same time using the two instances.) The word "instance" also applies to tasks. A task might be carried out by a function (like a command function) but the parameter may indicate a different way of doing the task. In theory one might have two "ADC" tasks active, one to return the value of the pot, the other the value of the photosensor. Again, you'd need to make sure the code was written to be reentrant and critical resources (the ADC) shared without one instance of the task interfering with the other. In our world that can't happen since the "terminal" cannot have more than one active task at a time. But, that task may be invoked while other tasks and processes (to run motors, blink LED's, etc.) are active.

Managing simultaneous tasks and processes is tricky. In this course, we are doing exercises to develop and understanding of some of the issues. We don't have help managing things because we are in a "bare metal" environment: we have to manage everything ourselves. That's the intent! On small systems, that's necessary because resources are of memory and processing capability are limited. But, then, so is the scope and complexity of what we are trying to do. If we were trying to do something really complicated, we'd need help: a more capable microcontroller or an even larger computer, and help with managing it. The "help with managing it" means an "Operating System".

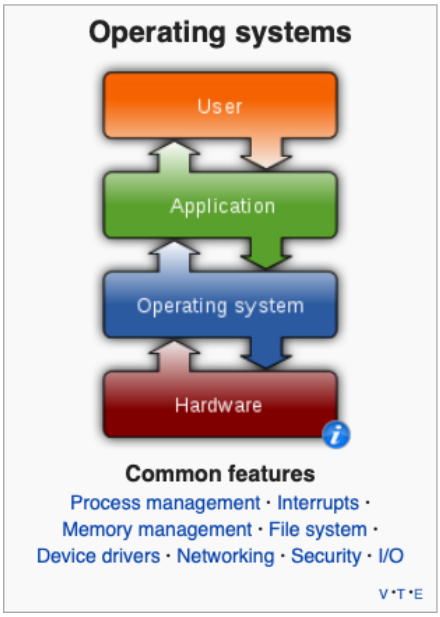
Operating system

From Wikipedia, the free encyclopedia

An **operating system** (**OS**) is system software that manages computer hardware, software resources, and provides common services for computer programs.

Time-sharing operating systems schedule tasks for efficient use of the system and may also include accounting software for cost allocation of processor time, mass storage, printing, and other resources.

For hardware functions such as input and output and memory allocation, the operating system acts as an intermediary between programs and the computer hardware,[1][2] although the application code is usually executed directly by the hardware and frequently makes system calls to an OS function or is interrupted by it. Operating systems are found on many devices that contain a computer – from cellular phones and video game consoles to web servers and supercomputers.



Our microcontrollers do not have an "Operating System". That means that our "application" must also perform the duties that an "Operating System" would on a more sophisticated computer. But, there is less to be managed. We are not maintaining a file system (mass storage). We are not printing. We are not even managing memory – everything is already allocated and installed in memory when our application begins with the calling of "main" out of reset. We do need to communicate – the serial port and its supporting functions (in SPI_Functions.c) does that. We also have "hcc_terminal" to manage the communications in order to carry out commands and print strings, one of the functions that an operating system

would normally perform. In a sense, hcc_terminal with the serial port ARE our operating system. But, that only works for commands. For managing tasks and processes when things get going, we are on our own.

Real-time operating system

From Wikipedia, the free encyclopedia

A **real-time operating system** (**RTOS**) is an operating system (OS) intended to serve real-time applications that process data as it comes in, typically without buffer delays. Processing time requirements (including any OS delay) are measured in tenths of seconds or shorter increments of time. A real-time system is a time bound system which has well defined fixed time constraints. Processing must be done within the defined constraints or the system will fail. They either are event driven or time sharing. Event driven systems switch between tasks based on their priorities while time sharing systems switch the task based on clock interrupts. Most RTOSs use a pre-emptive scheduling algorithm.

A key characteristic of an RTOS is the level of its consistency concerning the amount of time it takes to accept and complete an application's task; the variability is 'jitter'.[1] A 'hard' real-time operating system has less jitter than a 'soft' real-time operating system. The chief design goal is not high throughput, but rather a guarantee of a soft or hard performance category. An RTOS that can usually or generally meet a deadline is a soft real-time OS, but if it can meet a deadline deterministically it is a hard real-time OS.[2]

An RTOS has an advanced algorithm for scheduling. Scheduler flexibility enables a wider, computer-system orchestration of process priorities, but a real-time OS is more frequently dedicated to a narrow set of applications. Key factors in a real-time OS are minimal interrupt latency and minimal thread switching latency; a real-time OS is valued more for how quickly or how predictably it can respond than for the amount of work it can perform in a given period of time.[3]

If we are doing a complex application on a sophisticated computer, we would use a Real Time Operating System (RTOS) to help us manage things. In a normal operating system like Windows, precise timing is not particularly important. So the user has to wait a while. So what? No problem. On the other hand, a computer managing the control system for an aircraft had better not just do things when it feels like it. That is a “hard” real-time application! The physics of the real world will not wait. There are consequences that matter. Even in a relatively simple application, like a printer, moving the print head precisely in time is needed to make sure the ink or toner lands in the right place on the paper.

Our World:

Our MC9HCS08SH8 microcontroller is too small and limited to be running an RTOS. But we want to do things the RTOS way. We want to be able to control tasks and processes with precision. That is, we are going to do many of the same things that an RTOS would have to do to manage multitasking and multiple processes. This is really what the rest of the course is primarily about. Our tools are some way of dispatching singular “tasks” that the user initiates (the terminal will do that for us) and a way to maintain processes that have to keep operating. Some processes may be maintained by hardware (such as the TPM timers running a stepper at a constant speed). Others will need to be activated periodically (the clock that maintains time of day). At the heart of it all, we need some way to dispatch tasks, including the tasks that keep processes going. Just as was described for operating systems above, some tasks can be dispatched periodically (as time sharing) and others in response to events (interrupts). We need to do all this in a way that is sensitive to the relative priority of the tasks.

If We Stick to Polling:

Let's imagine a world (a processor) in which interrupts don't exist. How would we manage a variety of tasks and processes in which various things are going on with different priorities and durations? We'll use tasks and processes with which we are familiar:

1) The terminal process, which actually dispatches tasks to do different things.
2) A clock process that needs to update variables gHours, gMinutes, and gSeconds.
3) Monitor the pushbuttons to notice if somebody pushes one
4) Run a stepper motor. Assume the stepper has to advance every x milliseconds.
5) Blink and LED slowly (using a timing loop) – represents computing intensive task.

The most straightforward thing to do is to put all of these in the main loop, as below:

```
while(1){                  /* start main loop */
      terminal_process(); /* receive and dispatch commands */
      s1=PTAD_PTAD2;
      s2=PTAD_PTAD3;
      if(s1!=olds1||s2!=olds2){ /* monitor switches */
            olds1=s1;
            olds2=s2;
            s[0]=s1+'0';
            s[1]=s2+'0';
            s[2]='\0';
            print("switches=");
            print(s);}
      if(RTCCNT!=oldRTCCNT){ /Clock: *RTC counts 0-99 each sec */
            oldRTCCNT=RTCCNT;
            gMilliSeconds = gMilliSeconds+10;
            if(gMilliSeconds>=1000){gMilliSeconds=0; gSeconds++;}
            if(gSeconds>=60){gSeconds=0; gMinutes++;}
            if(gMinutes>=60){gMinutes=0; gHours++;}}
      if(gMilliseconds!=oldmilliseconds){ /* stepper motor driver */
            oldmilliseconds=gMilliseconds;
            stepperCount++;
            if(stepperCount>=stepperPeriod){
                  stepperCount=0;
                  stepperIndex++;
                  if(stepperIndex>=4)stepperIndex=0;
                  stepperoutput=outData[stepperIndex];
                  PTBD=PTBD|stepperoutput<<6;}}
      /* process to blink the LED slowly */
      for(i=0;i<blinklimit;i++)__RESET_WATCHDOG;
      PTBD_PTBD5=0;
      for(i=0;i<blinklimit;i++)__RESET_WATCHDOG;
      PTBD_PTBD5=1;
}                                  /* end main loop */
```

We have seen the problem with this. With the slow "blink the LED slowly" process dominates everything. We won't "poll" the clock often enough to keep up with the 10mS updates needed. The stepper will slow to a crawl. We'll miss characters in the terminal. What to do? Well, one thing we could do is to reorganize the main loop to put the highest priority tasks at the top, and do a "continue" after each so that we wait on the lowest priority (blink the LED at PTB5) until the end. Reorganizing the main loop looks like:

```
while(1){                  /* start main loop */
      if(RTCCNT!=oldRTCCNT){ /Clock: *RTC counts 0-99 each sec */
            oldRTCCNT=RTCCNT;
            gMilliSeconds = gMilliSeconds+10;
            if(gMilliSeconds>=1000){gMilliSeconds=0; gSeconds++;}
            if(gSeconds>=60){gSeconds=0; gMinutes++;}
            if(gMinutes>=60){gMinutes=0; gHours++;
            continue;}}
      if(gMilliseconds!=oldmilliseconds){ /* stepper motor driver */
            oldmilliseconds=gMilliseconds;
            stepperCount++;
            if(stepperCount>=stepperPeriod){
                  stepperCount=0;
                  stepperIndex++;
                  if(stepperIndex>=4)stepperIndex=0;
                  stepperoutput=outData[stepperIndex];
                  PTBD=PTBD|stepperoutput<<6;}}
      terminal_process(); /* receive and dispatch commands */
      s1=PTAD_PTAD2;
      s2=PTAD_PTAD3;
      if(s1!=olds1||s2!=olds2){ /* monitor switches */
            olds1=s1;
            olds2=s2;
            s[0]=s1+'0';
            s[1]=s2+'0';
            s[2]='\0';
            print("switches=");
            print(s);
            continue;}
      /* process to blink the LED slowly */
      for(i=0;i<blinklimit;i++)__RESET_WATCHDOG;
      PTBD_PTBD5=0;
      for(i=0;i<blinklimit;i++)__RESET_WATCHDOG;
      PTBD_PTBD5=1;
}                                  /* end main loop */
```

We show here that the clock advance task has been made the highest priority, followed by the stepper, the terminal, and the pushbuttons. We'd really like terminal_process to return a 1 if it does something, so that we could put an "if" to continue after it, as we did for those other higher priority tasks. Now, we always return to the top after doing something (except terminal_process, since it doesn't tell us if it did something). The higher priority tasks have the first opportunities to do something each pass through the loop, and we check all the other high priority tasks after doing something before we consider the lower priority tasks.

But a fundamental problem remains. Once we get into the LED timing loops, we are stuck there. There is no way to get back to the high priority tasks while this keeps going. If we can "break up" the processing of this slow task, into many smaller, quicker tasks, we can fix the problem. Otherwise, we can't. Not without interrupts. In this case, we actually can break up the timing loops. But we have to "rewrite" the process into a series of smaller tasks, and that makes the process less clear to the observer and more complicated. The relationship between the control variable "blinklimit" and the blink period will have to be recalibrated. In a more

complicated computational process like DSP, this gets “hard”. But, here’s what we’d do for this simple case. Yes, the algorithm is changes but we can still get the same effect.:

```
        /* process to blink the LED slowly */
        i++;
        if(i>=blinklimit){i=0; onoff=(onoff+1)&1;}
        PTBD_PTBD5=onoff;
        __RESET_WATCHDOG;
```

With Interrupts:

Suppose we cannot, or it would be difficult and undesirable, to break up a slow expensive process into smaller tasks that can be processed sequentially. Well, we could conceivably put the main loop inside the timing loops (like we did for __RESET_WATCHDOG). But that’s awkward and undesirable too. What really solves this problem is interrupts. Interrupts run at a “higher level” of priority. That is, an interrupt can cause processing in the main loop to be set aside while a higher priority task is taken care of. Indeed, for this example, we would move everything except the low priority LED blinking out of the main loop:

```
    while(1){                   /* start main loop */
        /* process to blink the LED slowly */
        for(i=0;i<blinklimit;i++)__RESET_WATCHDOG;
        PTBD_PTBD5=0;
        for(i=0;i<blinklimit;i++)__RESET_WATCHDOG;
        PTBD_PTBD5=1;
    }                                   /* end main loop */
```

The other tasks would be called out if Interrupt Service Routines (for RTC,RDRF):

```
interrupt 25 void rtc_isr(void){ /Clock: *RTC ints each 10 msec */
    char s1,s2, stepperoutput, s[3];
    RTCSC_RTIF=1;  /* reset flag */
    gMilliSeconds = gMilliSeconds+10;
    if(gMilliSeconds>=1000){gMilliSeconds=0; gSeconds++;}
    if(gSeconds>=60){gSeconds=0; gMinutes++;}
    if(gMinutes>=60){gMinutes=0; gHours++;}
    stepperCount++;
    if(stepperCount>=stepperPeriod){
        stepperCount=0;
        stepperIndex++;
        if(stepperIndex>=4)stepperIndex=0;
        stepperoutput=outData[stepperIndex];
        PTBD=PTBD|stepperoutput<<6;}}
        s1=PTAD_PTAD2;
        s2=PTAD_PTAD3;
    if(s1!=oldS1||s2!=olds2){ /* monitor switches */
        olds1=s1;
        olds2=s2;
        s[0]=s1+'0';
        s[1]=s2+'0';
        s[2]='\0';
        print("switches=");
        print(s);}}
```

```
interrupt 17 void rtdf_isr(void){
    terminal_process();} /* receive and dispatch commands */
```

Notice that this makes the "blink LED" free running while no other tasks are active; it actually gets more time since no time needs to be wasted polling other processes to see if they are ready to run. The stepper, clock, and switches are now "time share" tasks that get checked every 10 mSec. If that high a frequency was not needed, they could be checked less often. If a higher frequency of attention is needed, a TPM timer could be used for that purpose (with its separate interrupt), or the primary RTF clock could be made to run faster. BUT, we need to be sure that whatever is dispatched on every RTC call can complete before the next one. That should not be a problem for clock updates and advancing the stepper. The steps taken are finite and "in-line" (no looping).

But the switch checking is different. We need to print whenever the switches change. How long does that take? Let's see. We need to print 11 characters. At 9600 bps that should only take 11.45 mSec. Hmmm. That means that the clock's next interrupt will come before the print finishes up! Well, so the clock will be 1.45 msec late. And the stepper would hesitate that 1.45 msec as well. Is that a problem? Probably not. We will not have missed anything, just will be slightly late with that update. But suppose the bit rate is 4800 bps? Then we have a problem! We will miss an interrupt for the clock. It's as if 10 mSec vanishes. The stepper will slightly slow down.

(Actually for the 9600 bps case we might be OK since the first character gets loaded into the serial port shift register and the second can be accepted immediately after, so we really only need 1.04 msec for the print – almost not a problem at all. The rest of the ISR code will only need a several microseconds.)

Still problems:

However. (There always seems to be a "however, doesn't there?) There is still a problem. It is that "print" is a "blocking" function. Just as our blinking LED process was originally. When we were in the middle of blinking the LED, we couldn't do anything else. We were "blocked.": In the case of print, print calls "putchar" (actually the function `TERMIO_PutChar`, which was passed in to hcc_terminal when we did initialization). TERMIO_PutChar code is this:

```
/*void TERMIO_PutChar(char send)*/
int TERMIO_PutChar(char send){
  int dummy;
  while(!SCIS1_TDRE);/* wait until TDRE=1 */
  dummy = SCIS1;
  SCID  = send;
  dummy=send;
  return dummy; /*added line*/
}//end SPI_PutChar
```

Notice the while loop. This waits until the TDRE flag goes up, meaning that we can pack another character into the Transmit Data register. That blocks us from doing anything else. Because putchar is blocking, and because print calls putchar, print is blocking.

Now, consider what terminal process does. It dispatches command tasks, the command callback functions. Suppose somebody enters the command "help". What comes out is typically hundreds of characters. All those characters have to be funneled through the "blocking" putchar function. We are stuck doing printing, because terminal_process, which called print, is itself now called out of an ISR, and while that ISR is going, nothing can interrupt it. ISR's can preemptively interrupt things in the main loop, but (on this machine) they cannot interrupt each other! As it happens, the serial port is a higher level interrupt (lower numbered) than the RTC. On this machine that means, when interrupts are again enabled, if both an RTC interrupt and an RDRF interrupt are pending, the RDRF interrupt gets to go first. So, let's suppose that help command prints 200 characters. That takes about 208 msec. That's got serious consequences for our clock and the stepper. Something has to be done.

Fix proposal #1: Enable interrupts:

This is the simplest fix. Interrupts are inhibited by an ISR because the hardware sets the interrupt flag (I) in the Processor Status Word (PSW). The PSW is a register inside the CPU of the microcontroller that has bits to represent whether the most recent instruction resulted in a zero (Z), negative (N) or carry (C) as well as a few other things needed for normal code execution, especially conditional branches. Setting I in the PSW inhibits further interrupts. Flags are raised, but nobody answers.

But, we can clear the I bit in the PSW! As the last thing we do in terminal_process, before calling the command callback function, we could enable interrupts. See the excerpt from terminal_process shown below:

```
/* Identify command. */
x=find_command(cmd_line+start);
/* Command not found. */
if (x == -1)
{
  print("Unknown command!\r\n");
}
else
{
  EnableInterrupts; /* enable interrupts */
  (*cmds[x]->func)(cmd_line+end+1);
}
cmd_line_ndx=0;
print_prompt();
```

Having done this, the terminal_process no longer enjoys its previously privileged status, and can be interrupted just like the main loop. If the RTC goes off before the printing for the "help" command is done, it does so, gets its job done, then exits back into the terminal_process called help task to resume printing. The RTC could go off several times before the print job is done. So, this fixes the problem. Bu thtere are a couple of additional "howevers" lurking.

First, suppose someone happened to push or release a pushbutton when the "help' command is printing. The RTC's call to "print" will likely be a second, and recursive, call to print, and to putchar. At best, the print about the buttons will be inserted into the middle of the print job for "help". That's undesirable. It's likely characters may be lost. Even worse.

An additional consequence is that stack usage increases. Each time an ISR is called, stuff about what is happening in the interrupted task has to go on the stack. Local variables are already on the stack; that's not a problem. But the Program Counter (PC) that points to the next instruction that would have been executed needs to go on the stack, as well as the PSW that's keeping track of carries and zeros and negatives for branching, and the A, H and X registers. Figure 5-1 in the SH8 Manual has a diagram. That's 5 bytes. Now, in addition, you have any local variables for the ISR and anything that the ISR might call. If the an ISR is interrupting another ISR that has cleared its interrupt inhibit flag, that happens again. The stack climbs. This machine has only a limited amount of RAM. Tha stack has to be big enough for a worst case: the terminal_process ISR is interrupted at the worst possible time when you are deep inside a command callback function that has used print (or worse, printf!).

Worse can happen. The RTC interrupt calls print too, to respond to switch pushings. Suppose we provided the same relief for that ISR as well; if we need to do a print we enable interrupts before we do so. Now, suppose that these print calls get held up – either they are inherently too slow, or they are competing with a print job from terminal_process. The next RTC interrupt could be called before the earlier one is finished! RTC's should not be reentrant, and should not need to be. If this kind of thing cascades, ISR gets piled on top of ISR on the stack until the whole thing wipes out all of memory, and VERY BAD THINGS happen! (This is where you hope the dog goes off and saves you by resetting the machine.)

In general, clearing the interrupt flag before you exit the ISR is a dangerous thing to do, and should be considered very, very carefully. There may be times and places where it makes sense. But look for other alternatives first.

Fix #2: Make print non-blocking:

Yes! This is what we'd like to do, but how? Characters can be sent out only at a limited speed as set by the serial port. How can print return until they have all been sent? The answer is to have the print function stash toe characters somewhere until they are ready to be sent. That allows print to return as soon as it puts the characters somewhere; they can then be sent out at leisure. The place where we put the characters is called a "buffer".

In effect, the "print" process (which we will call it now, instead of just a task) is broken up into two pieces. There is a process that fills the buffer with characters. We will think of that as the "print" task and still call it a task. Then there is the process that empties the buffer by sending out the individual characters one at a time. We will be doing this as an exercise (to be written up as soon as I can get it done).

The catch is, of course, that the individual characters to be printed have to be stored in a buffer until they can be sent out, and that buffer necessarily has to be in RAM. RAM is limited. If someone asks for the "help" command, that may be a few hundred characters. Is your buffer big enough for that? We have 512 bytes of RAM, but ¼ of that is in zero page (which we could use for our global variables). We need a stack. How much is left for the buffer? In practice, you just might be able to have a buffer as big as 256 bytes. (We usually use powers of 2.) How to actually do the buffered print will be put in a separate document.

The print task fills characters into the buffer than exist immediately. It is no longer blocking. So, print calls don't take significant time, and most of the problems timing problems that we were worried about earlier go away. The process of emptying the print buffer is a process that used the serial port TDRE interrupt. As long as there are characters in the buffer, we enable the TDRE interrupt. Each time it goes off, we stuff another character into the Transmit Data Register of the serial port, turning off the TDRE flag until it empties again. When we pick up the last character and sent it the TDRE interrupt is turned off until someone puts more stuff into the buffer.

The same concept can be used for other "blocking" tasks. For example, we wait for the ADC to finish converting an analog value before we pick it up to print or do something else. Starting the ADC is not synchronous; it is not blocking. It is waiting for COCO that is "blocking". Instead of waiting for COCO, we can turn on the interrupt flag when starting the ADC, and after having started the conversion, exit. Then, when the flag goes off signaling that the conversion is complete, an ISR picks up the data and does whatever needs to be done with it. In effect, this is the same kind of solution as described for printing. The task is divided into two parts, one that starts things and one that finishes the job. In fact, this kind of strategy can be expressed in other ways as well.

Fix #3 Make print non-blocking, but store pointers, not characters:

This method of managing printing uses the same idea as above, but instead of storing the individual characters to be printed, the buffer stores pointers to the strings. This way much less RAM is used, just two bytes per character string to be printed. But, as usual, there's a catch. This method works well for constant strings stored in ROM. Things like the "help" command texts (which are in constant command structures). But if you use this to print something in RAM, it is very, very dangerous. You can't print a string in a local variable, because by the time you get around to actually printing,. The variable and its contents are gone. You have a pointer to something that no longer exists, and there's no telling what is stored there now. You could mitigate this by requiring that strings from RAM be from static or global variables, but how are you going to ensure that those don't get changed between when the string is sent to print and when the characters actually get sent? Not simple. Dangerous even, unless you separate out RAM in global memory space for every different string that you might print, and that's going to chew up a lot of RAM.

It's conceivable that you could use a combination of techniques, with a character buffer for variable (RAM) string prints and a pointer buffer for constant (ROM) prints. But then the two print functions have to be carefully integrated so that the buffer emptying process that puts the characters into the serial port knows how to do that correctly. That's tricky.

Fix #4: Leave print blocking, but have calls to it made only in the main loop:

This approach requires us to make a rule: "Never call print out of an ISR!" That means terminal_process needs to be fundamentally changed. Instead of calling command callback functions itself, it needs to put commands to be called somewhere. Then, another process we'll call the "dispatcher", called out of the main loop, actually does the call. In between? A buffer.

Assuming that we would only be invoking one command at a time, the buffer need only be a pointer in global memory to the next command needing to be executed. And, a string for the parameter string to be passed to that command. (Conceivably the current command buffer could just be made a global accessible directly to command callback functions.) The presence of the command needing to be executed would be signaled by a nonzero pointer. When the command callback is executed, the pointer is changed back to being a zero.

The obvious problem we are back to is that if we are in the middle of those long expensive processes we don't want to break up (the led blinking), the command won't actually execute until we get to the next pass of the main loop. But unlike the previous situation, we will be able to type in characters rapidly since terminal process is not blocked, only the execution of the command callback function.

Similar things can be done for other kinds of tasks. For example, the pushbuttons. Instead of printing, the ISR would signal by setting a few characters or a flag indicating the buttons were changed and a new remark needed to be printed. Then, a task would be initiated out of the main loop to do the actual print.

Conclusion:

Managing multitasking is a complicated business, especially in the presence of real time constraints, limited memory resources, and inherently slow or blocking tasks and processes. It is hoped that this discussion and the attendant exercises will give you a practical appreciation for these issues. This certainly is not a comprehensive treatment of the topic. In practice, for critical real time computer operations, assuming adequately capable processors, an RTOS is used to give guaranteed performance. Hopefully this material will help give an appreciation for that they have to do. Many topics have not been covered here, with perhaps "scheduling" in general among the most important.