Dr. Dobb's Software Tools for the Professional Programmer

Concurrent Small C

Preemptive multitasking, IPC, and interrupt handling all in a small package

Andy Yuen

Andy, who holds a masters' degree in engineering from Carleton University in Ottawa, Canada, currently works in Sydney, Australia. You can reach him at andy_yuen@sydney.sterling.com.

Concurrent Small C's features make concurrent programs easy to write and understand

Concurrent Small C is a programming language specifically designed for embedded-system development. It is a superset of Small C, which was introduced by Ron Cain in "A Small C Compiler for the 8080" (DDJ, May 1980). Concurrent Small C uses five new keywords to provide direct support for preemptive multitasking, interprocess communication, and interrupt handling. CSC is unrelated to "Concurrent C," developed by Gehani and Roome; see "Concurrent C for Real-time Programming," by N.H. Gehani and W.D. Roome (DDJ, November 1989).

Small C is a self-compiler written in Small C itself-a novel idea in 1980. The compiler generates assembly code that has to be assembled and linked to produce an executable. I use Small C because of its small size, source-code availability and, most importantly, because it is thoroughly documented in James E. Hendrix's book A Small C Compiler (M&T Books, ISBN 0-13-814724-8).

CSC is based on Small C Version 2.2 Small C 2.2 is interesting in that the compiler generates intermediate codes in memory (called P-codes), which are instructions for a two-accumulator virtual machine with a stack pointer. The compiler then runs them through a peep-hole optimizer and translates them into the processor's assembler language using a translation table. Consequently, this version of the compiler generates much more efficient code than previous versions. The use of P-codes and a translation table also make it easy to modify the compiler to generate code for different processors.

Monitor as a Synchronization Mechanism

CSC's tasking/synchronization model is based on a construct called "monitor," which is both a synchronization and data-abstraction mechanism that encapsulates system resources and provides a set of operations that can be used by various tasks to manipulate the resources. (Don't confuse this monitor with the control program, also called a "monitor," in ROM on single-board computers or microcontrollers.)

You can think of a monitor as a fence around the protected resources. The only way to go past the fence is via gates called "entries." At any time, there can only be one task inside a monitor. Other tasks wanting to enter are blocked until the one inside exits voluntarily or performs a Wait or Signal operation on a condition variable. Wait and Signal are synchronization operations used inside a monitor. When a Wait is executed, the task is put in the condition variable specified and another task enters the monitor. When a Signal is called, the running task temporarily steps out of the monitor and the task waiting in the specified condition variable resumes execution inside the monitor. If the condition variable is empty when a Signal is performed, nothing happens, although the running task may still step out of the monitor temporarily (depending on the implementation).
Figure 1 illustrates a monitor with two gates, entry 1 and entry 2. Task T3 is inside the monitor. Tasks T1 and T2 are waiting to enter the monitor via entry 1. T4 and T5 are waiting on conditions C1 and C2, respectively. If T3 executes a Signal on condition C1, it steps out of the monitor and T4 resumes execution inside the monitor. T3 will be competing with T1 and T2 to enter the monitor again. If, instead, it executes a Wait on condition C2, it gets queued in C2, and T1 enters the monitor. Since only one task is allowed inside a monitor at any one time, the integrity of the monitor's resources is guaranteed.

Adding Concurrency to Small C

Small C does not require many syntax changes to evolve into a concurrent language; there are only five new keywords in CSC.
A task, handled by the keyword task, is a thread of execution that is declared using the syntax in Example 1(a), which looks much like a function declaration. priority designates the task's priority (between 1 and 32767 inclusive). The higher the value, the higher the priority. stacksize denotes the size of the task's stack in bytes. The priority and stacksize specifications are optional. Leaving out the specifications or specifying NULL tells the system to use default values; see Examples 1(b) and 1(c).
All declared tasks are started before the first executable statement of main(). Even if several tasks implement the same logic, they must all be declared explicitly. There is no shortcutall tasks are statically declared and dynamic task creation within a running task is not allowed. However, this is not as limiting as it sounds because they can all call the same function in the task body; see Example 1(d).
A monitor consists of three types of components: resources (monitor data structures), entries, and monitor-initialization routines.
The monitor-initialization routine initializes the monitor data structures. CSC calls it during system initializationbefore
starting tasks, installing interrupt handlers, and so on. The monitor-initialization routine or function is denoted by the keyword monitor. Each .c file can only contain one monitor-initialization function.
A monitor entry is denoted by the keyword entry. An entry is the gate through which a task can go inside the monitor enclosure. Entries define how tasks access and manipulate the monitor's resources. Each monitor should be put in a separate .c file so that you can use the keyword static to render the monitor data structures (resources) accessible only from within the module. If this is not enforced, some errant task may destroy the monitor data by mistake. The keyword static is not implemented in Small C, but has been added to CSC for this purpose.

Example 2(a) shows how simple mutual exclusion can be implemented using a monitor. Every monitor must associate a consistency criterion, called an "invariant," with its critical data. The invariant guarantees that the data is accurate and up-to-date. In this example, busy is the invariant. It guarantees that the state of the resource is accurately representedeither busy or not busy.
Assume that this mutex monitor is used to manage exclusive use of, say, a printer. Multiple tasks can request the use of this printer, as in Example 2(b). The invariant busy is initialized to False (0) by declaration. The monitor-initialization function in this example does nothing useful. You may need to perform many initializing operations for a more-complicated monitor, though. For completeness, each monitor should always declare an initialization function even if it does nothing.
When the first task tries to acquire the printer, busy is False, so it sets it to True (1) and exits the monitor. It now has exclusive use of the printer until it calls release(). When a second task tries to acquire the printer, busy is True, so it waits in the condition variable cond. Note that the address of the condition variable is passed to Wait, not its value.
The monitor entry release() sets busy to False and signals the condition variable. Remember that a signal causes the signaling task to go out of the monitor so that the awakened task can proceed. Consequently, the awakened task is assured that busy is accurate and up-to-date.
The mutex monitor can easily be generalized into providing multiple resources, as in Example 2(c).
Since interrupt handling is an important part of any embedded system, a concurrent language must provide interrupt support; hence CSC's interrupt keyword. CSC's interrupt-support mechanism is powerful, yet simple to use. A task synchronizes its operation with an interrupt-service routine (ISR), declared using the keyword interrupt, by using the function call StartIO(intrnum, command, &result);. In this case, intrnum is the interrupt number, command is a function to be invoked by CSC before putting the task in the condition variable (supplied by CSC) associated with the interrupt, and result is the variable to receive the result of the I/O operation. command is a user-written function to initiate an I/O operation. The function must return an integer value: zero if the command is successful, nonzero otherwise. If the command is successful, the task will be put to sleep awaiting notification of the completion of the I/O operation from the ISR. If the command fails, StartIO returns immediately with the error code of the command saved in result. A NULL may be specified in place of a command, in which case, the task is put to sleep immediately. Example 3 declares an ISR, where intrnum is the interrupt number, which must be a numeral, not a variable or expression. The ISR must return a value: zero if the I/O operation is not yet complete, or nonzero if the operation has completed or failed. A nonzero value causes the CSC run time to Signal the waiting task. When the task resumes, result contains the
value returned by the ISR. StartIO and interrupt functions are designed for writing device handlers.
The complete list of kernel functions is summarized in Table 1.

Implementation

The Small C compiler generates code that uses separate code (CS) and data/stack (DS=SS) segments on an 80x86. The compiler is organized into four modules: cc1.c,

cc2.c, cc3.c, and cc4.c. The cc1.c module handles overall program flow and parsing; cc2.c, input preprocessing (such as macro expansion); cc3.c, expression analysis; and cc4.c, code generation and optimization.
Changes are made to the parser in cc1.c to recognize and generate code for the new language features. Whenever the parser encounters the keywords monitor, task, and interrupt, it saves that function's address and parameters, if any, in separate named segments (one for each kind) for use by _cscinit to set up the environment during system initialization. I've added three new P-codes and their translation table entries to cc.h and cc4.c for generating native assembler code for these new features. These P-codes are:

- CSCINIT, which generates a call to the function _cscini in the CSC run time to execute all monitor-initialization functions, create all declared tasks, install system (timer) and user-defined interrupt handlers, create an idle task, convert main() into a task, and go into multitasking mode. If the user does not declare a task, the system stays in single-tasking mode, which means that the system-timer interrupt handler will not be installed and a call to Delay will return immediately. In single-tasking mode, CSC is 100-percent compatible with Small C.

- ENTERMON is the request to enter a monitor. This is simply translated into a disable-interrupt instruction to guarantee that only one task can be inside a monitor at any one time.

- EXITMON exits a monitor. This is translated into an enable-interrupt instruction.

Modifying the compiler to generate code to support the new language features is straightforward. The difficulty is in the initial design of the language features and the system kernel. For portability, the kernel is mostly written in CSC. Only the tricky stack manipulation stuff is written in assembler, using the #asm/#endasm directives. Also, all function names starting with an underscore (_) are meant to be used internally only.
Each task created by the kernel has a task descriptor and a stack associated with it. The task descriptor keeps the task's priority, the number of clock ticks to delay (if the Delay function is called), the task's stack pointer, and a pointer to the next task descriptor. The values of the registers (or task context) are saved in the task's stack during task switching. There is only one running task at any one time. The pointer to its descriptor is kept in running. All ready-to-run tasks' descriptors are chained together according to their priority. The pointer to the head of the chain is kept in ready. In multitasking mode, on every timer interrupt, the running task is put on hold in the ready queue and the first ready task is reactivated.
When a monitor entry executes a Wait(&cond), the running task's context is saved and its descriptor is queued according to its priority in the condition variable cond, which is, in fact, a queue. The first task queued in ready is restarted and becomes the running task.
When a Signal(&cond) is called from within a monitor entry, the running task's context is again saved, and its descriptor is queued in ready. However, instead of restarting a task from the ready queue, it restarts the first one in the condition queue.
The kernel also provides a Delay function for tasks to suspend execution for a specified number of clock ticks. This capability could have been implemented as a monitor outside the kernel, but I implemented it inside the kernel for convenience. The kernel uses a delta list to keep track of the order in which the tasks are to be reactivated. For example, if the tasks T1, T2, and T3 request delays of 2, 6, and 9 timer ticks, the delta values saved in their descriptors are 2, 4, and 3, respectively. If two of them request the same delay as in 2, 2, and 5, then their delta values are 2, 0, and 3, respectively. The pointer to the first task in the delta list is kept in delta.

Interrupt handling is the most difficult, yet interesting, part of the kernel. Instead of generating special code to save all registers on entry and to restore all registers on exit for every user-defined ISR, and hooking it to the specified interrupt vector, CSC actually uses a common interrupt handler for all interrupts. For each user-defined ISR, an interrupt descriptor is built that contains fields for the interrupt number, an inter-segment call instruction to the common interrupt code (80x86 parlance), a pointer to the user ISR, a condition variable, an error variable, and a pointer to the next interrupt descriptor. Figure 2 shows how an interrupt is serviced.
The interrupt vector is set pointing to its descriptor's inter-segment call instruction. When an interrupt occurs, it vectors to this call instruction, which invokes the common ISR. The common ISR saves the task context and calls the user-defined interrupt handler. If all interrupts call a common ISR, how does it tell which user function to call? The trick is in the call instruction, which means that the return address on the stack contains the address immediately following the call instruction in the interrupt descriptor. By design, this is the address to the pointer to the user function. Using this address, other fields in the interrupt descriptor can be accessed as well. If a user-defined ISR returns a nonzero value, the common ISR moves the running tasks to the ready queue and wakes up the task waiting in the interrupt descriptor's condition variable. If a zero is returned, this step is skipped. _dodelta, which updates the delta list and always returns a 1, is declared by the system as the user-defined function for the timer interrupt. The common ISR wakes up the first task in the ready queue instead.
StartIO starts an I/O operation and waits in the interrupt descriptor's condition variable. StartIO and user-defined ISRs are the mechanism in CSC for writing device handlers.

A Concurrent Program Example

To demonstrate how you might use CSC, imagine an embedded system (with remote-console support) designed to solve the classic dining-philosopher problem: There are N philosophers sitting at a round table. A bowl of Chinese food is put in the middle. A total of N chopsticks are placed on the table, one between adjacent philosophers. They alternate between eating and contemplating the meaning of life. To eat, a philosopher must pick up two chopsticks, one from each side. After eating, he puts the chopsticks back where they came from. The problem is preventing deadlock or indefinite postponement. A deadlock results, for example, if every philosopher picks up and retains the chopstick on his left; in this case, no one will be able to obtain both chopsticks. Indefinite postponement occurs if philosopher #1 always eats when philosopher #3 is thinking, and vice versa, in such case, philosopher #2 will never be able to pick up both chopsticks to eat.
The example program consists of four files: philos.c, mchopstk.c, mconsole.c and asyn8250.c (Listings One through Four, respectively; listings begin on page 94). philos.c contains the main program and task declaration for each philosopher (a total of five). Each task consists of only one statement: philosopher(n), which implements the behavior of a philosopher. It calls the chopsticks monitor entries, GetChopsticks() and PutChopsticks(), to acquire and release the chopsticks, respectively. It delays a random amount of time (between 1 and 200 timer ticks) after each monitor-entry call, and sends a message to the console monitor to describe what it is doing.

mchopstk.c implements the chopsticks monitor, which provides the logic to solve the dining-philosopher problem. The array chopsticks represents the number of chopsticks (0, 1, or 2) available to each philosopher. Each philosopher has its own condition variable to wait in waiting[n]. Also, each can determine its neighbor by looking at the array myleft[n] and myright[n]. Inside the GetChopsticks() entry, philosopher n is allowed to pick up chopsticks only if chopsticks[n] equals 2; that is, both chopsticks are available. Otherwise it waits in waiting[n]. After the philosopher picks up the chopsticks, it decreases its left and right neighbor's chopstick count by 1 and exits the monitor. When a philosopher releases its chopsticks, it calls the entry PutChopsticks(), which increases its neighbors' chopstick counts by 1 and wakes up one or both of them if their chopstick count now equals 2.

mconsole.c implements the console monitor. Each philosopher outputs a message to the console describing its current activity. Since the console can only output one message at a time, a monitor coordinates the use of the console. A philosopher prints a message by calling the monitor entry Send(). On entry, Send examines the invariant busy to see if the console is busy (1). If so it waits in the condition variable conready. Otherwise, it copies the message to the monitor's buffer, sets busy to 1 and Signals the condition variable msgavail, where the console is waiting. The console gets work by calling the monitor entry GetMsg(). On entry, GetMsg() checks if busy is set. If it is not set, there is nothing to do, and it waits in the condition variable msgavail. Otherwise, it copies the message to the caller's buffer, sets busy to 0, and Signals conready to wake up a waiting philosopher. This monitor provides a single buffer for all philosopher messages and forces a philosopher to wait when the buffer is not available. Modifying the monitor to provide multiple buffers is not difficult and is left as an exercise for the reader.

asyn8250.c is the most interesting module. It implements a link to a remote site by using the PC's asynchronous adapter. This module demonstrates how the xmitter task coordinates with the ISR asynsend() to transmit messages over the link. xmitter initializes the asynchronous adapter (setting the baud rate, data size, number of stop bits, and so on) and calls the console monitor entry GetMsg() to get a message to send. It displays the message on the screen and calls StartIO() to initiate the transmission. StartIO specifies that the startx command be invoked before waiting in its interrupt condition variable. startx transmits the first character of the message. When the adapter is ready to transmit the next character, it generates an interrupt that passes control to asynsend(), which sends the next character, when the end of the message is reached, it returns a nonzero value, which causes the CSC common interrupt handler to wake up the xmitter task.

To write device handlers in CSC, devote an I/O task (such as xmitter in the example) to each device and use StartIO to interact with its ISR to service the device.

Conclusion

Concurrent Small C's features make concurrent programs easy to write and understand. Still, it could be improved. Exception handling and task termination are not addressed in the current version. Also, the P-codes ENTERMON and EXITMON (which enforce mutual, exclusive access to monitor resources) are implemented simply by disabling and enabling interrupts, respectively. This affects the interrupt latency if a lot of work is done inside the monitor. It would be better to use simple binary semaphores to provide the mutual exclusion, so that monitor-entry processing is no longer time critical. Also, the object file containing main() must be the last

object file specified in the link command, because of the way the named segment tables for recording the monitor, task, and interrupt function addresses and parameters (for use by _cscinit) are terminated when main() is encountered. Although minor, if overlooked, this may cause unnecessary difficulty and frustration in debugging concurrent programs.

Despite these shortcomings, standardizing the tasking and synchronization model in the language makes porting an embedded system from one processor to another often a simple recompile using the CSC compiler and the C library for the target platform. Although only the 80x86 version was available at this writing, a project is ready to go (pending donations of hardware by vendors) to port Concurrent Small C to popular microcontrollers such as the 6811 and 8051.

Figure 1: Diagram of a monitor.

Figure 2: Interrupt processing using a common ISR.

Example 1: (a) Declaring a task; (b) task abc uses the default priority (64) and stack size (512 bytes);

(c) task xyz uses the default priority and a stack size of 384 bytes; (d) creating similar tasks.

(a)

```
task taskname([priority] [,  stacksize])
{
    ...
}
```

(b)

```
task abc()
{
    ...
}
```

(c)

```
task xyz(NULL, 384)
{
    ...
}
```

(d)

```
common()
{
    while (1)
        {
        ...
        }
}
task task1()
{
```

```
    common();
}
    ...
task taskn()
{
    common();
}
```

Example 2: (a) Using a monitor to implement a simple mutual exclusion; (b) multiple tasks can request exclusive use of the printer; (c) generalizing the mutex monitor.

(a)

```
#include <stdio.h>

/* monitor data */
static CONDITION cond;
static int busy;

void monitor printer()
{
    printf("Initializing mutex monitor\n"); /
}
void entry acquire()
{
    if (busy)
        Wait(&cond);
    busy = 1;
}
void entry release()
{
    busy = 0;
    Signal(&cond);
}
```

(b)

```
/* get exclusive use of the printer */
acquire();
    /* start print job */
/* release the printer */
release();
```

(c)

```
#include <stdio.h>

/* monitor data */
#define MAXRESOURCES    10

static CONDITION cond;
static int count;

void monitor printer()
{
    printf("Initializing resource   monitor\n"); /
}

void entry acquire()
{      if (count == 0)
```

```
        Wait(&cond);
    count--;
}

void entry release()
{
    count++;
    Signal(&cond);
}
```

Example 3: Declaring an ISR.

```
interrupt isr(inttnum)
{
    ...
}
```

Table 1: Useful kernel functions.

Function	Description
enable()	Enables interrupt.
disable()	Disables interrupt.
Delay(n) int n;	Delays for n timer ticks (works only when at least one task has been declared, otherwise return immediately with a value of 1).
inpbyte(port) int port;	Returns a byte from the specified port.
outpbyte(port, value) int port; int value;	Outputs a byte value to the specified port.
SetPrior(prior) int prior;	Sets the running task's priority.
GetPrior()	Returns the running task's priority.
StartIO(intr, command, result) int intr; int (*command)(); int *result;	Executes command and waits for the interrupt-service routine for intr to wake it up. On return, result contains the I/O operation's return code.
Yield()	Gives up the CPU to another ready task.
Signal(cond) CONDITION *cond;	Wakes up a task waiting in the specified condition variable (not to be used outside a monitor-entry function).
Wait(cond) CONDITION *cond;	Goes to sleep in the condition variable cond until a Signal(&cond) is executed (not to be used outside a monitor-entry function).

Listing One

```
#include <stdio.h>
unsigned int seed = 127;

/* generate pseudo-random number between 1-200 inclusive */
rnd()
{
   seed *= 177;
   if (!seed)
      seed = 127;
   return seed % 200 + 1;
}
/* logic for a dining philosopher */
void philosopher(id)
int id;
{
   char buffer[128];
   int n;
   while (1)
   {
      /* acquire chopsticks */
      GetChopsticks(id);
      /* display message and eat */
      n = rnd();
         sprintf(buffer, "Philosopher #%d eating for %d ticks :-)\n",id,n);
      Send(buffer);   /* pass to console task */
      Delay(n);       /* simulate eating */
      /* release chopsticks */
      PutChopsticks(id);
      /* display message and think */
      n = rnd();
        sprintf(buffer, "Philosopher #%d thinking for %d ticks :-(\n",id,n);
      Send(buffer);   /* pass to console task */
      Delay(n);       /* simulate thinking */
   }
}
/* task declarations */
void task p0()
{
   philosopher(0);
}
void task p1()
{
   philosopher(1);
}
void task p2()
{
   philosopher(2);
}
void task p3()
{
   philosopher(3);
}
/* main program: note that the function philosopher is used as
   the body of a number of tasks including main() */
main()
```

```
{
   /* all declared tasks start before this. main() is converted into a
      task by Concurrent Small C and becomes the fifth philosopher */
   philosopher(4);
}
```

Listing Two

```
#include <stdio.h>

/* data for the chopstick monitor */
static int chopsticks[] = {2, 2, 2, 2, 2};  /* chopstick counts */
static int myright[] = {1, 2, 3, 4, 0};     /* right neighbor identity */
static int myleft[] = {4, 0, 1, 2, 3};      /* left neighbor identity */
static CONDITION waiting[5];                /* condition variables */

/* initialization for chopstick monitor */
void monitor mchopstk()
{
   /* demonstrates that this is being called during initialization */
   printf("Initializing chopsticks monitor...\n");
}
/* monitor entry to get both chopsticks */
void entry GetChopsticks(id)
int id;
{
   /* wait until both chopsticks available */
   if (chopsticks[id] != 2)
      Wait(&waiting[id]);
   /* decrement neighbors' chopstick counts */
   chopsticks[myright[id]]--;
   chopsticks[myleft[id]]--;
}
/* monitor entry to release the 2 chopsticks */
void entry PutChopsticks(id)
int id;
{
   /* increment neighbors' chopstick counts and wake
      them up if they now have both chopsticks */
   if (++chopsticks[myright[id]] == 2)
      Signal(&waiting[myright[id]]);
   if (++chopsticks[myleft[id]] == 2)
      Signal(&waiting[myleft[id]]);
}
```

Listing Three

```
#include <stdio.h>

/* monitor data */
static CONDITION conready;  /* console ready condition variable */
static CONDITION msgavail;  /* message available condition variable */
static int busy;            /* buffer inuse invariant */
static char buffer[128];    /* monitor's message buffer */

/* console monitor initialization */
void monitor mconsole()
{
   printf("Initializing console monitor...\n");
```

```
}
/* monitor entry to send message to console */
void entry Send(msg)
char *msg;
{
   /* wait till buffer is not in use */
   if (busy)
      Wait(&conready);
   /* copy message to local buffer and wake up console */
   strcpy(buffer, msg);
   busy = 1;
   Signal(&msgavail);
}
/* monitor entry for console to get message */
entry GetMsg(msg)
char *msg;
{
   /* wait till message is available */
   if (!busy)
      Wait(&msgavail);
   /* copy message to caller's buffer and wake up any task
      waiting to deliver a message */
   strcpy(msg, buffer);
   busy = 0;
   Signal(&conready);
}
```

Listing Four

```
#include <stdio.h>

/* 8259 interrupt controller interrupt mask register port# */
#define MASK8259    0x21

/* async 8250 constants */
/* these parameters are for com2 */
#define IOBASE      0x2F0
#define INTVEC      0x0B
#define DEVMASKBIT  0x08

/* 8250 control register port number */
#define RLINECTR    (IOBASE + 0x0B)
#define RLBAUD      (IOBASE + 0x08)
#define RHBAUD      (IOBASE + 0x09)
#define RLINESTAT   (IOBASE + 0x0D)
#define RRECV       (IOBASE + 0x08)
#define RXMIT       (IOBASE + 0x08)
#define RINTEN      (IOBASE + 0x09)
#define RMODEMC     (IOBASE + 0x0C)
#define RINTID      (IOBASE + 0x0A)
#define RMODEMSTAT  (IOBASE + 0x0E)

#define TXREADY     0x20    /* ready to transmit */
#define SETDIVISOR  0x80    /* set divisor */
#define L2400       0x30    /* low value for clock divisor */
#define H2400       0x00    /* high value for clock divisor */
#define DATA8       0x03    /* 8-bit data */
```

```
#define DCD         0x80    /* data carrier detect */
#define DSR         0x20    /* data set ready */
#define CTS         0x10    /* clear to send */
#define DTR         0x01    /* data terminal ready */
#define RTS         0x02    /* request to send */
#define OUT2        0x08    /* output 2 signal */
#define TXEMPTY     0x02    /* Tx holding register empty interrupt */

#define COMPLETED   0xFF    /* user-defined completion return code */

static int pos;         /* position of char in msg to send */
static char msg[128];   /* message to send */
static int linefeed;    /* time to send a linefeed */

/* send a char over the asyn link */
sendchar(ch)
char ch;
{
   /* check for modem status */
   if (inpbyte(RMODEMSTAT) & (DCD | DSR | CTS)) {
      /* wait till Tx is ready and send */
      while ((inpbyte(RLINESTAT) & TXREADY) != TXREADY);
      outpbyte(RXMIT, ch);
      return 0;
      }
   return 1;
}
/* initialize the 8250 chip */
void Init8250()
{
   /* set baud rate at 2400, no parity, 8-bit data */
   outpbyte(RLINECTR, SETDIVISOR);
   outpbyte(RLBAUD, L2400);
   outpbyte(RHBAUD, H2400);
   outpbyte(RLINECTR, DATA8);

   /* set modem control */
   outpbyte(RMODEMC, (DTR | RTS | OUT2));

   /* enable 8250 Tx holding register empty interrupt */
   outpbyte(RINTEN, TXEMPTY);

   /* enable 8259 interrupt controller for com2 */
   outpbyte(MASK8259, inpbyte(MASK8259) & ~DEVMASKBIT);
}
/* interrupt service routine to send a message. Note: a non-zero return value
    wakes up xmitter task which initiated transmission by calling StartIO */
interrupt asynsend(INTVEC)
{
   char ch;
   int retcode;
   if (linefeed) {
      /* time to send a linefeed */
      linefeed = 0;
      retcode = sendchar(LF);
      }
   else if ((ch = msg[++pos]) != 0) {
      /* translate LF to CR followed by LF */
```

```
        if (ch == LF)
            {
            retcode = sendchar(CR);
            linefeed = 1;
            }
        else
            /* send char as is */
            retcode = sendchar(ch);
        }
    else
        return COMPLETED;
    return retcode;
}
/* start interrupt-driven message transmission */
startx()
{
    pos = 0;
    return sendchar(msg[0]);
}
/* xmitter task declaration: note use of high task priority */
void task xmitter(256)
{
    int status;
    Init8250(); /* init 8250 */
    while (1) {
        /* get message and print it */
        GetMsg(msg);
        printf(msg);
        /* start i/o operation to send message to remote console.
            Only return when operation is completed or failed */
        StartIO(INTVEC, startx, &status);
        if (status != COMPLETED)
            printf("Async transmit error: %d\n", status);
        }
}
```

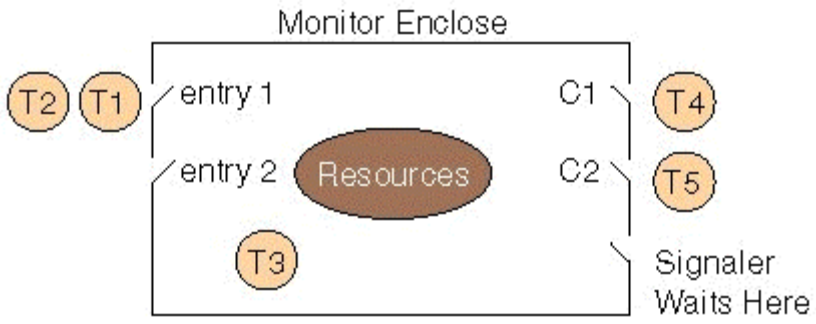
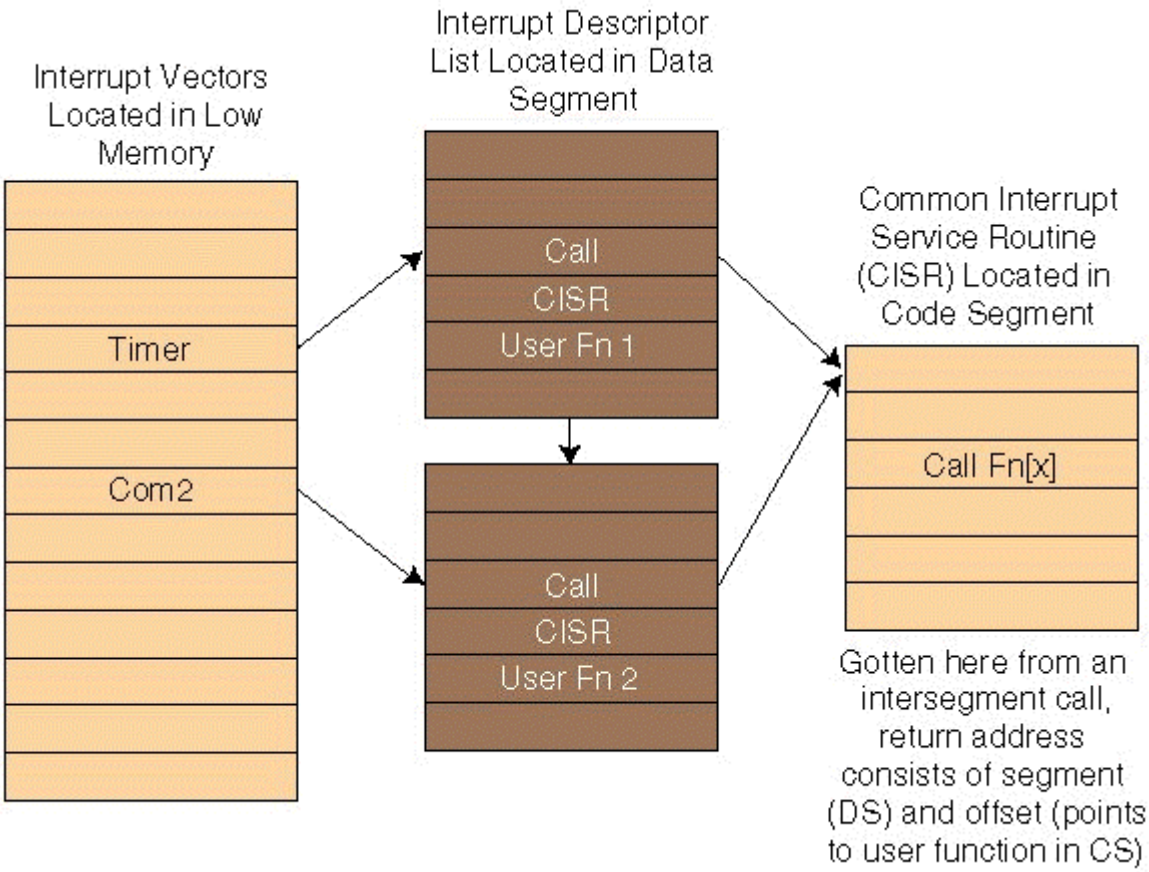


Figure 1.

Figure 2.


Interrupt Vectors Located in Low Memory
Timer
Com2
Interrupt Descriptor List Located in Data Segment
Call
CISR
User Fn 1
Call
CISR
User Fn 2
Common Interrupt Service Routine (CISR) Located in Code Segment
Call Fn[x]
Gotten here from an intersegment call, return address consists of segment (DS) and offset (points to user function in CS)