

Basic design using a real-time operating system

Reference: Simon Chapter 8

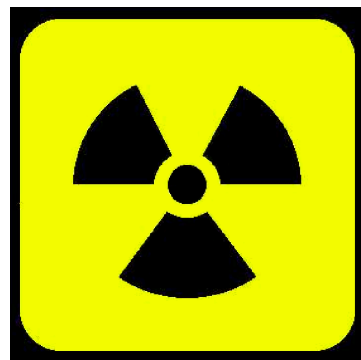
Now what?

- Over the next couple of classes we will discuss how to put all that we learned so far into useful designs for embedded-system software.
- Assumption that your system has real time constraints.
- Be aware that embedded-system software design is complex and has as many exceptions as it has rules.

Overview

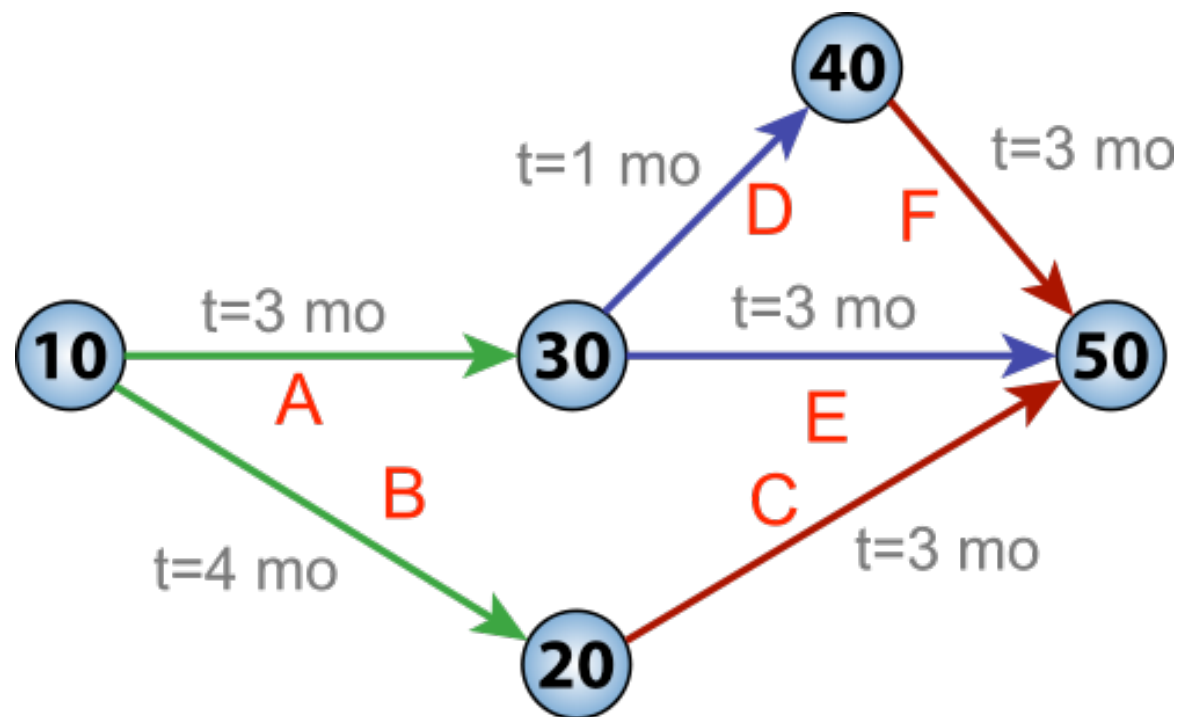
Overview

- It is harder to specify a real-time system than a desktop application.
- "What must the system do?" must also answer questions such as "How fast must it do it?"
- Example of real time systems: bar-code scanner, nuclear reactor temperature measurement system.



- You must know how **critical** each timing is.

Notions of criticality

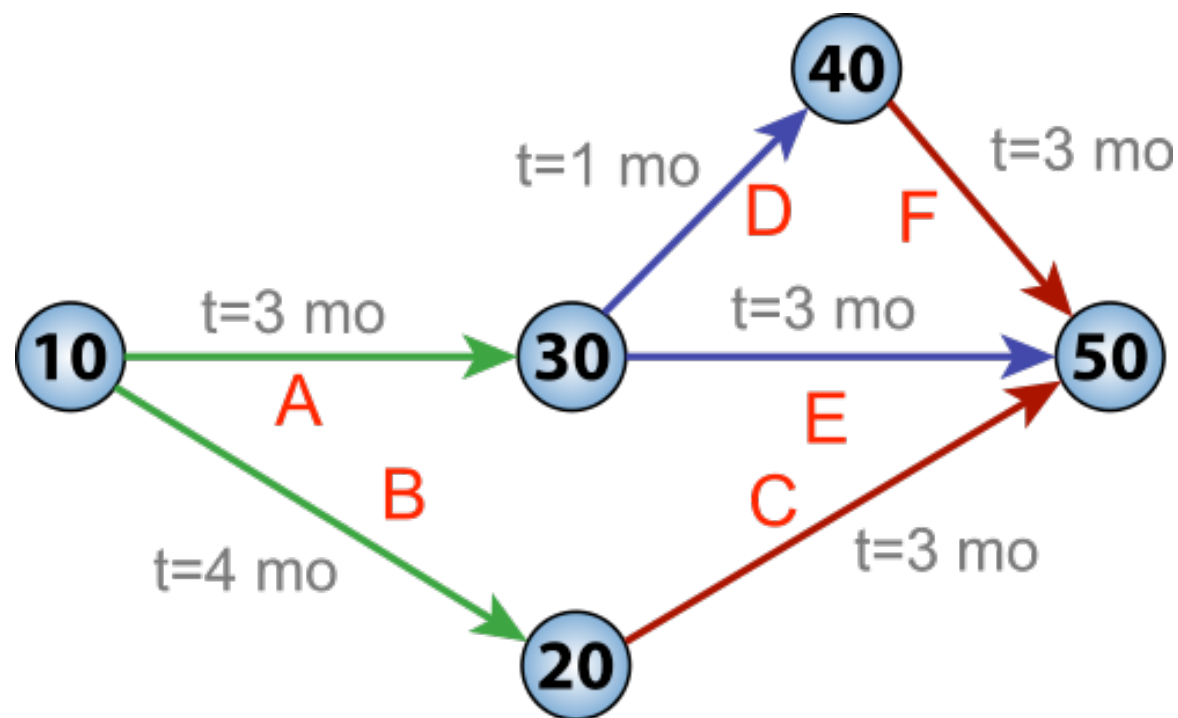


We can allocate different groups on different tasks.

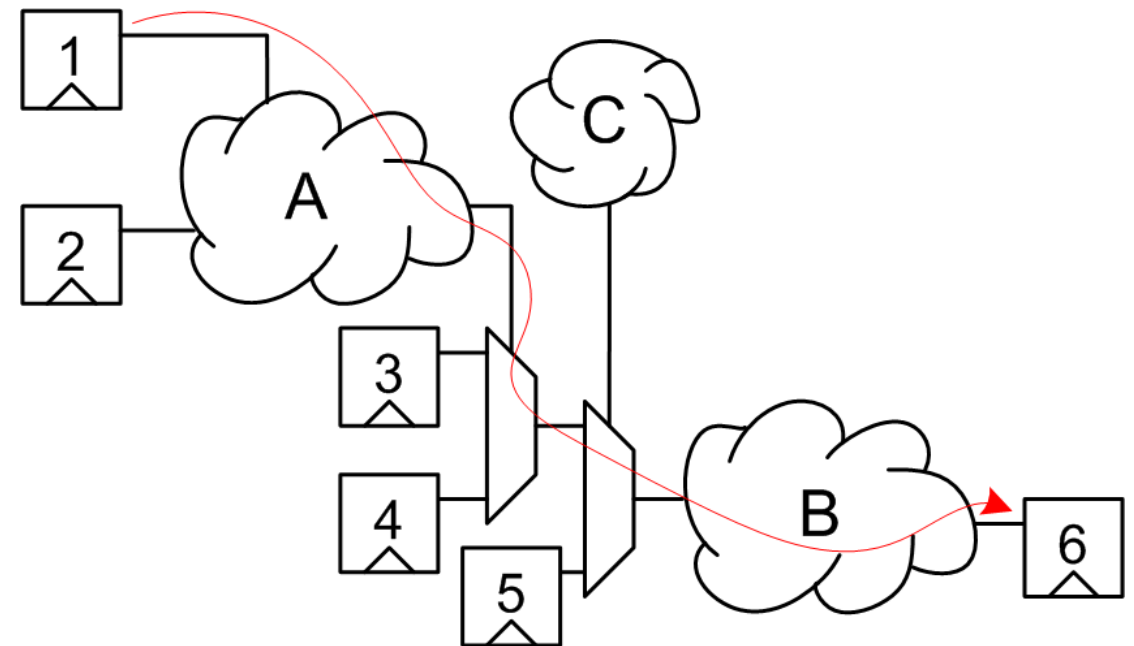
- Critical path method (for project management).
- PERT chart for a project with five milestones (10 through 50) and six activities (A through F).
- The project has two critical paths: activities B and C, or A, D, and F – giving a minimum project time of 7 months with fast tracking. Activity E is sub-critical, and has a float of 1 month.

Critical path

The **critical path** is the longest necessary path through a network of activities when respecting their interdependencies.



Project management



Electrical circuit

Hard and soft real-time systems

- It's probably okay for the cordless bar-code scanner to respond on time in 99% of the cases and be slightly too slow the other 1 percent of the time.
- However failure to respond quickly enough to reactor issues 1% of the time is unacceptable.
- Systems with absolute deadlines, such as the nuclear reactor system, are called **hard real-time systems**.
- Systems that demand good response but that allow some fudge in the deadlines are called **soft real-time systems**.



To design effectively, you must know something about the hardware!

- Suppose your system will receive data on a serial port at 9600 bits (about 1000 characters) per second: 9600 baud is 9600 bps.
- Storing in memory each individual received character causes an interrupt.
- Software design must accommodate a serial-port interrupt routine that will execute about 1000 times each second.
- How would you improve on this?

We can use direct memory access (DMA)

- If the serial port hardware can copy the received characters into memory through a **DMA** channel.
- ...and your system has no need to look at the characters immediately when they arrive.
- Then you can dispense with that interrupt routine and the problems it will cause.

What is direct memory access (DMA)?

DMA is a piece of circuitry that, without software assistance, can:

- Read data from an I/O device, such as a serial port or a network, and then write it into memory

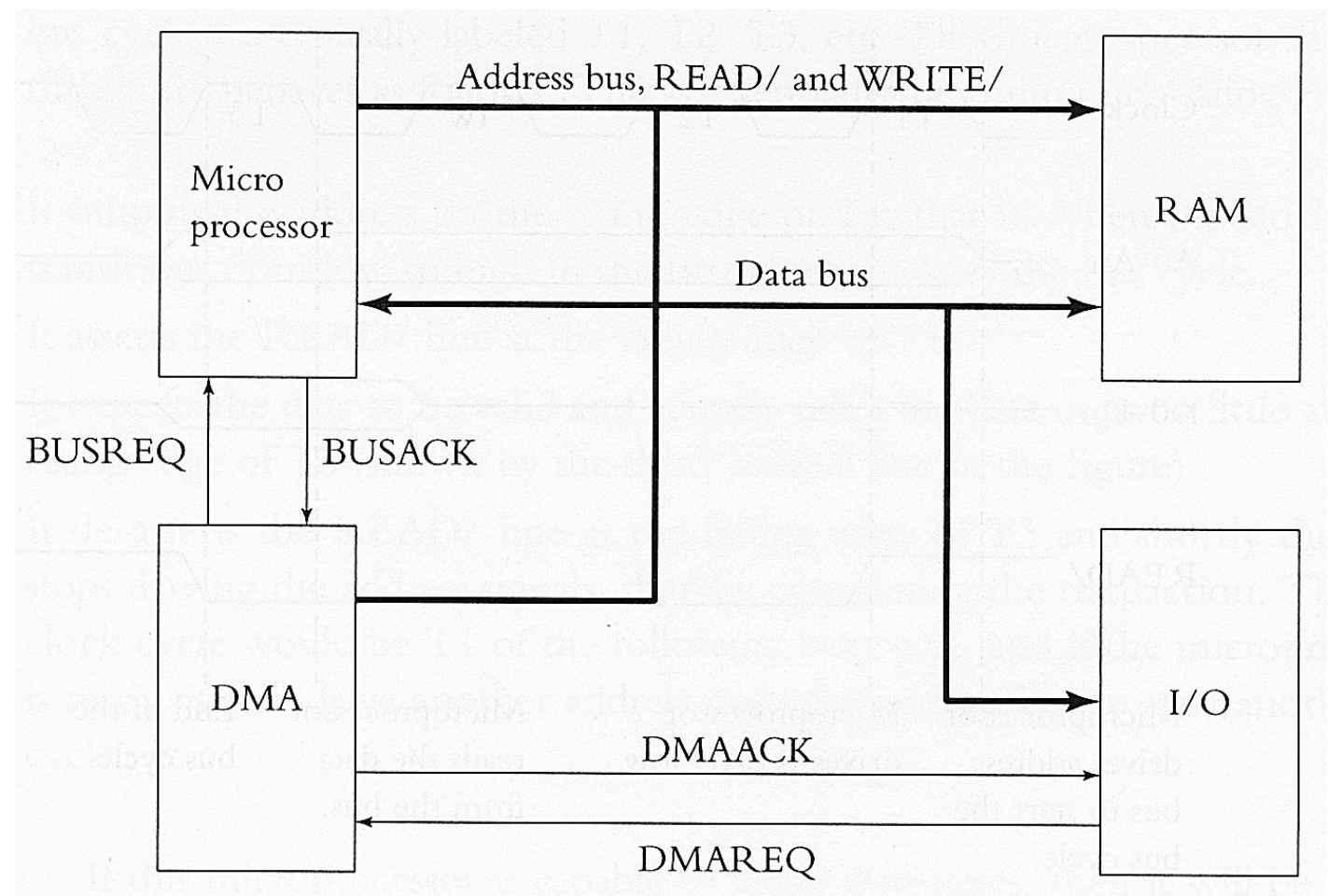
or

- Read from memory and write its contents into an I/O device

Caution: memory only has one set of address and data signals. The DMA must make sure that it is not driving those signals while the microprocessor is also driving them.

Using DMA: we want to transfer the data from the I/O into RAM

1. **DMAREQ** signal is asserted.
2. DMA circuit asserts **BUSREQ** to the microprocessor.
3. When microprocessor is ready to give up the bus, it asserts **BUSACK**.
4. DMA circuit puts address in the address bus.
5. **DMAACK** and **WRITE/** are asserted.
6. I/O places data in the data bus.

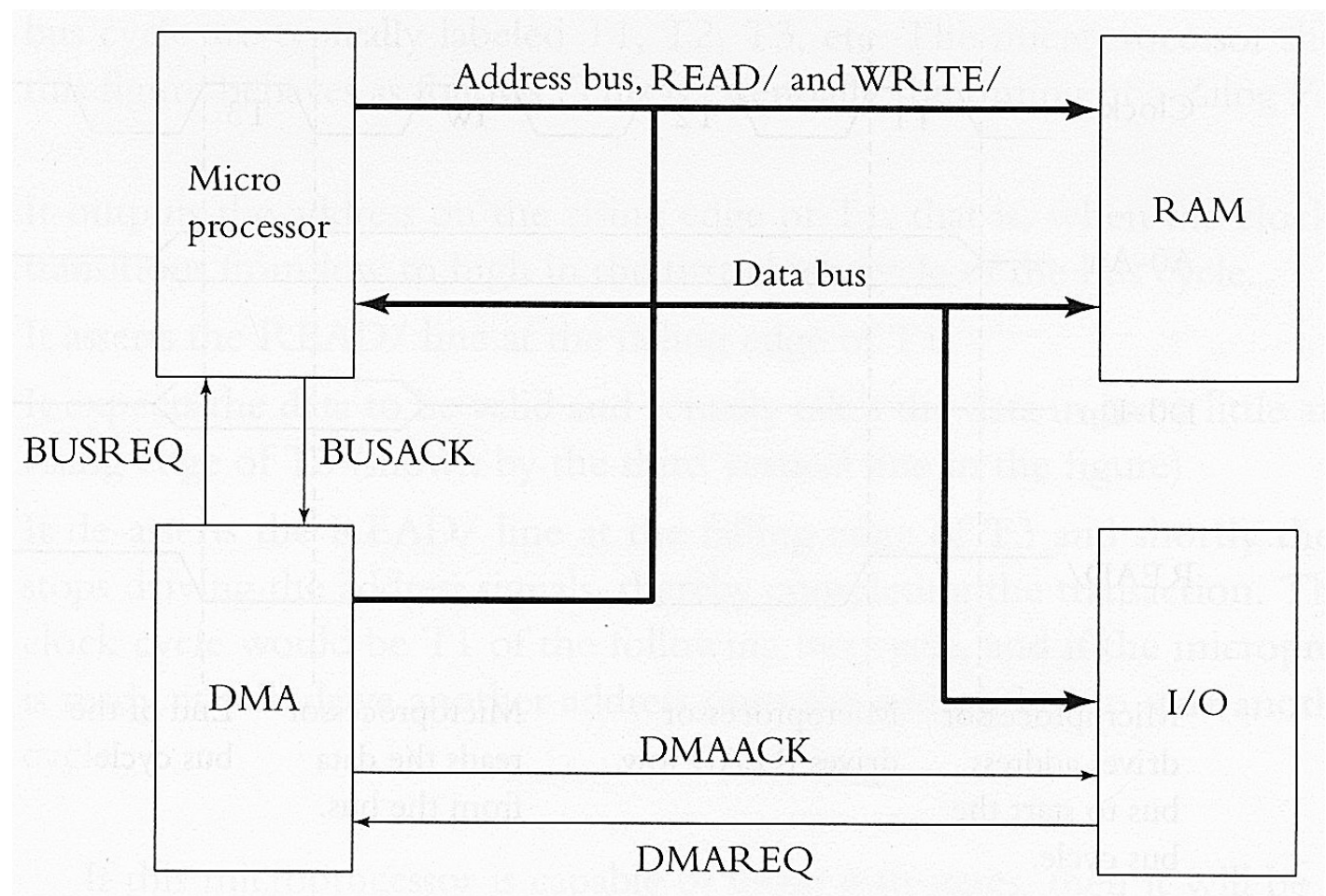


What's next?

Now that data has been written to RAM, the DMA circuit releases:

- DMAACK
- Address bus
- BUSREQ

The microprocessor releases BUSACK and microprocessor execution resumes.



Microprocessor speed

- You must have some feel for the microprocessor speed.
- Knowing which computations will take long enough to affect other deadlines is a necessary design consideration.
- "Can our microprocessor execute the serial-port interrupt routine 1000 times per second and still have any time left over for other processing?" is a question that needs an answer.
- Unfortunately, only experience and experimentation can help you with this.

Principles

Interrupts are driving force of embedded systems

- Embedded systems commonly have nothing to do until something requires a response:
- Either the passage of time OR some external event.
- The driving force of embedded systems are interrupts:
 - ▶ External events generally cause interrupts.
 - ▶ We can make the passage of time cause interrupts (with hardware timers).

Common design technique: system is commonly on the blocked state.

An embedded system design technique is to have RTOS tasks spend most of the time in the blocked state waiting for:

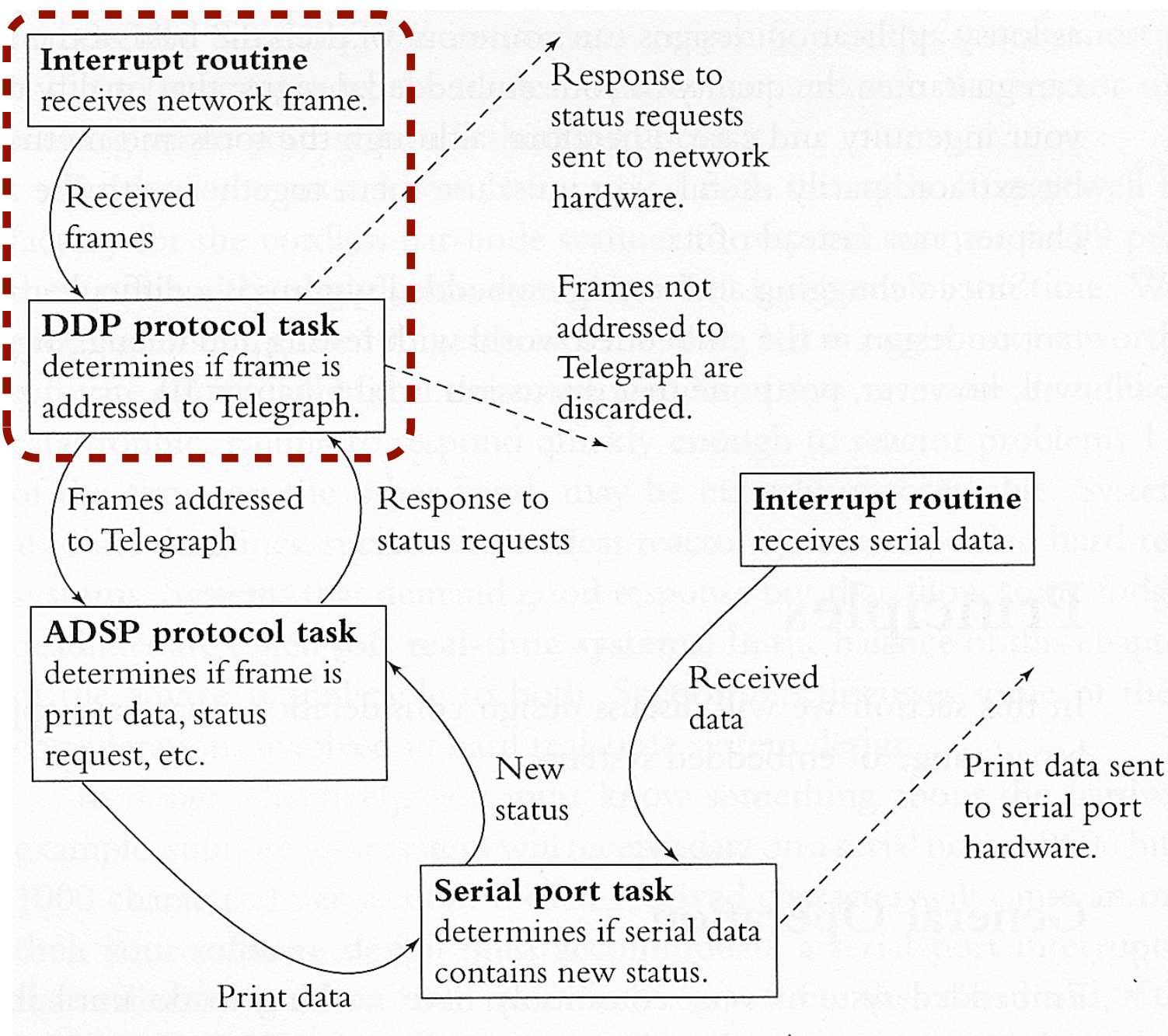
- ▶ an interrupt routine
- ▶ another task to send a message
- ▶ another task to cause an event
- ▶ another task to free a semaphore

Any of these operations will tell the tasks that there is something to do.

Interrupts are fundamental

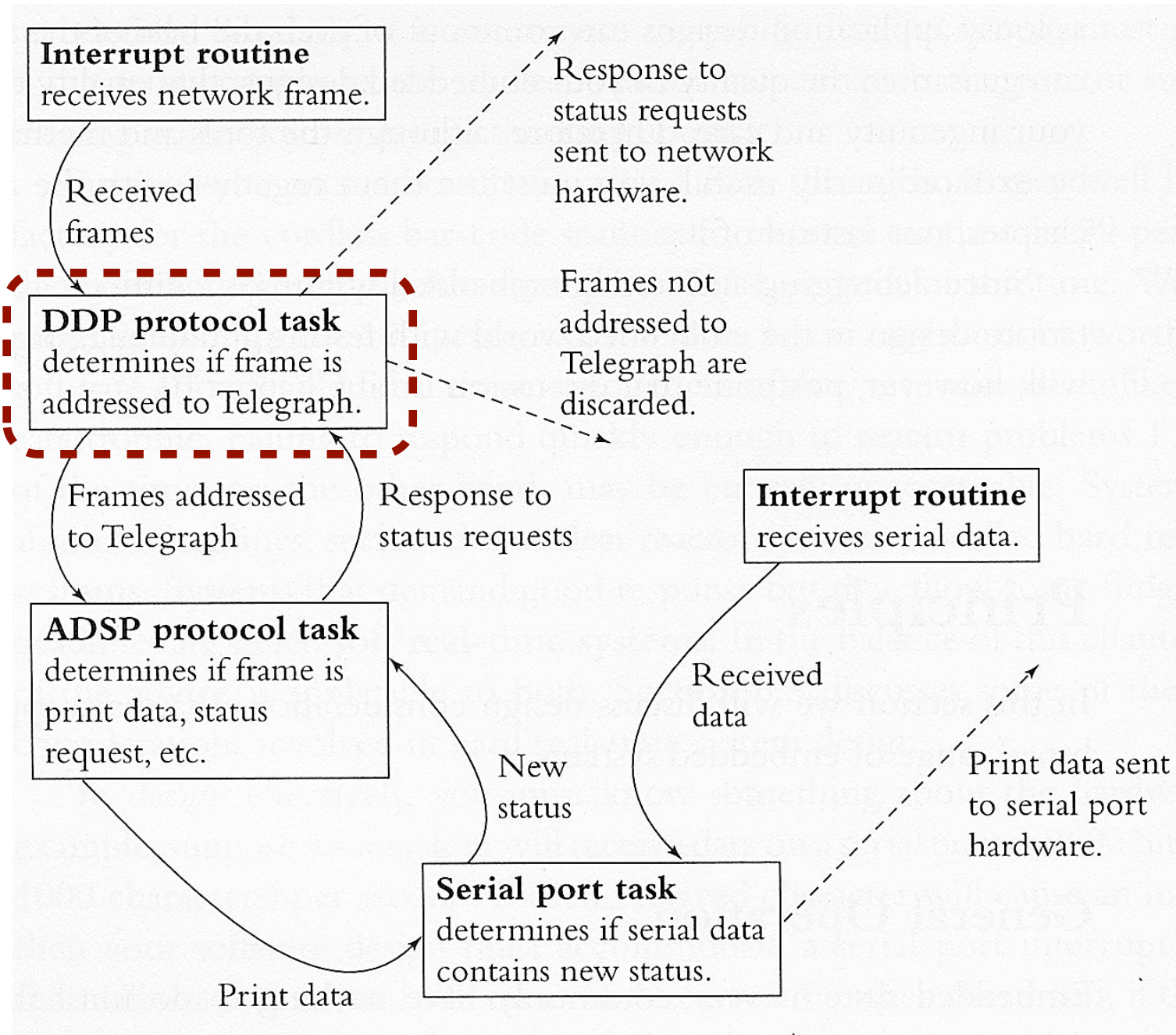
- When an interrupt occurs, the interrupt routine uses the RTOS services to signal one or more of the tasks.
- Each does its work and each may then signal other tasks.
- Each interrupt can create a cascade of signals and task activity.

Simplified version of what happens inside a telegraph system



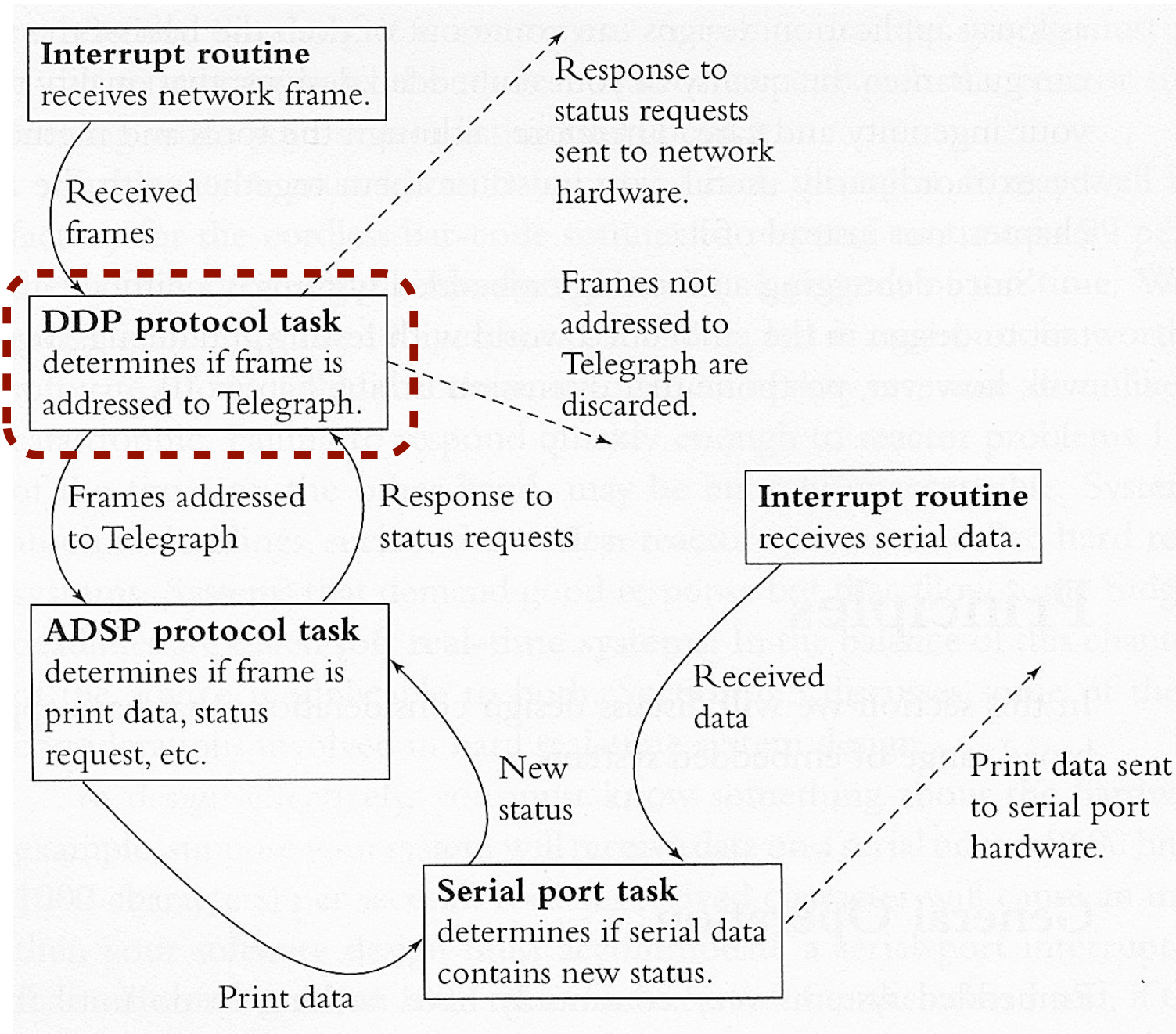
- When the system receives a network frame, the hardware interrupts.
- A network frame is a data packet that includes frame synchronization that allows the receiver to detect the beginning and end of the packet in the stream of symbols or bits.

DDP protocol task



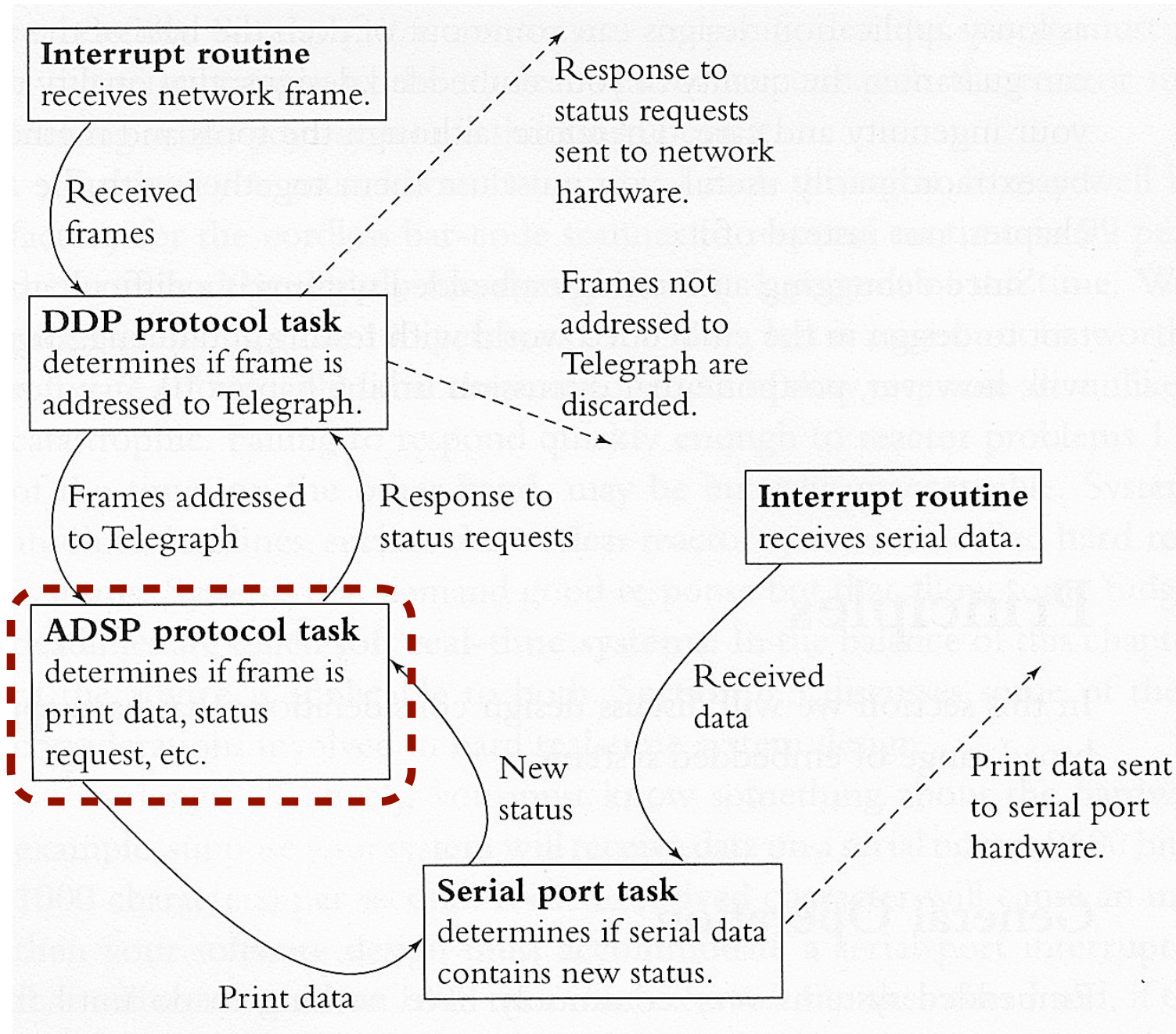
- The interrupt routine resets the hardware and then passes a message containing the received frame to the DDP protocol task (outdated!).
- Datagram Delivery Protocol (DDP) main responsibility is for socket-to-socket delivery of datagrams over an AppleTalk network.
- In simpler terms, its a program that routes data from point A to B.

After the interrupt, DDP protocol is no longer blocked



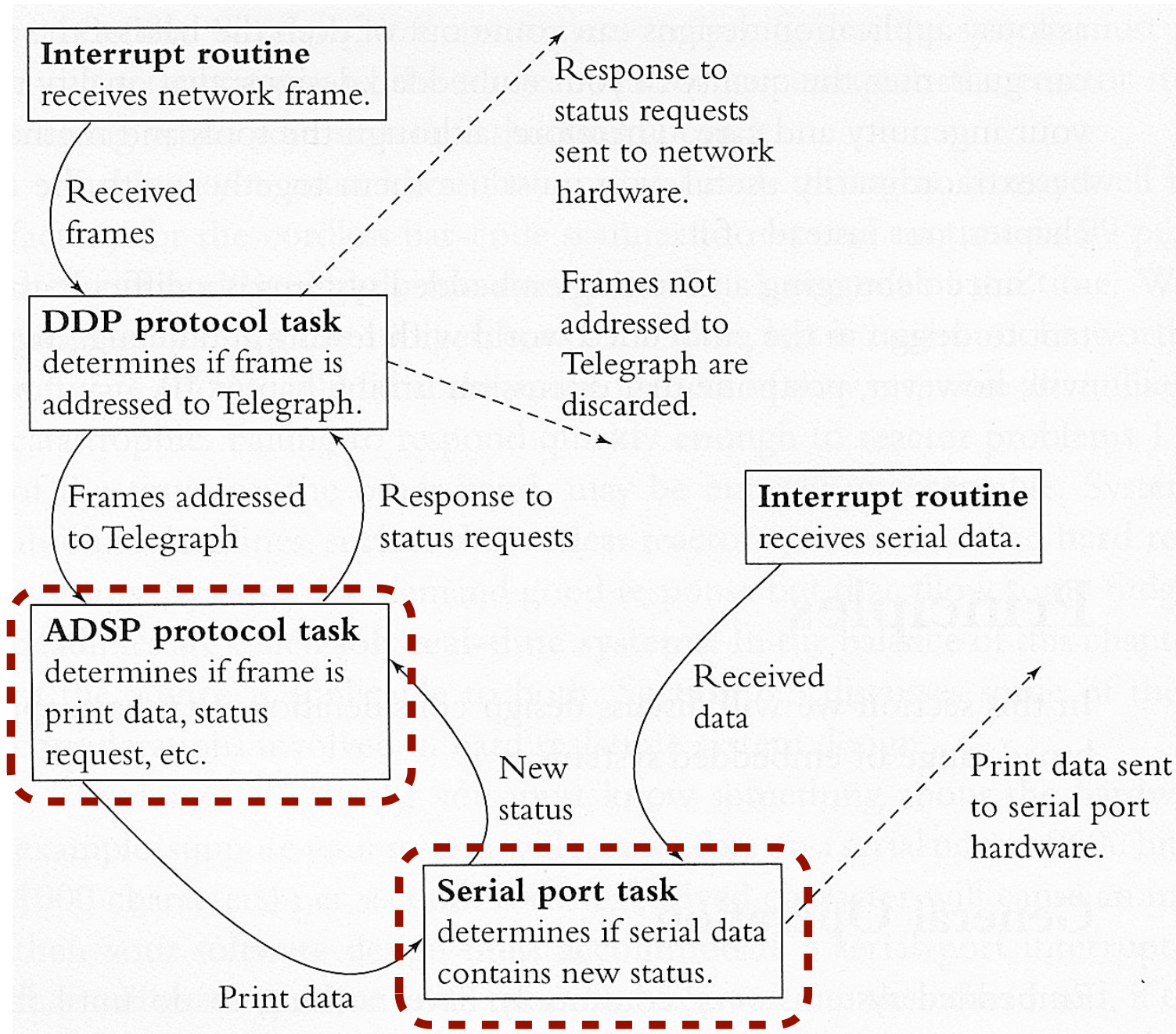
- The DDP protocol task was blocked waiting for a message
- When this message arrives, the task wakes up and, among many other things...
- ... Determines if the frame was intended for the Telegraph or if it was sent to some other network station and received by mistake.

Unblocking the ADSP protocol which then reads the data packet



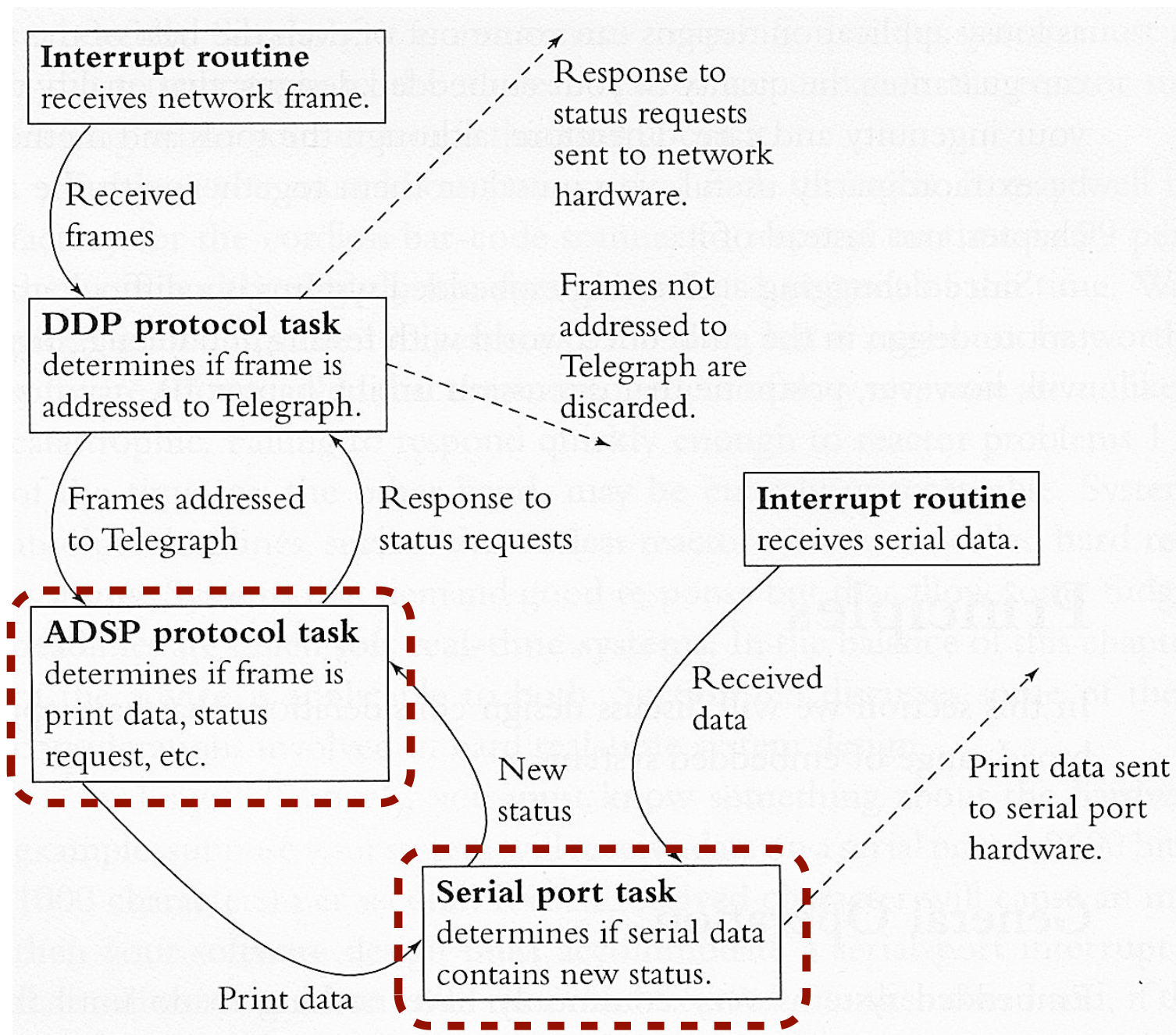
- If the frame was intended for the Telegraph, the DDP protocol task sends a message containing the received frame to the ADSP protocol task.
- This message unblocks the ADSP protocol task, which determines the contents of the received frame.

Unblocking the ADSP protocol which then reads the data packet



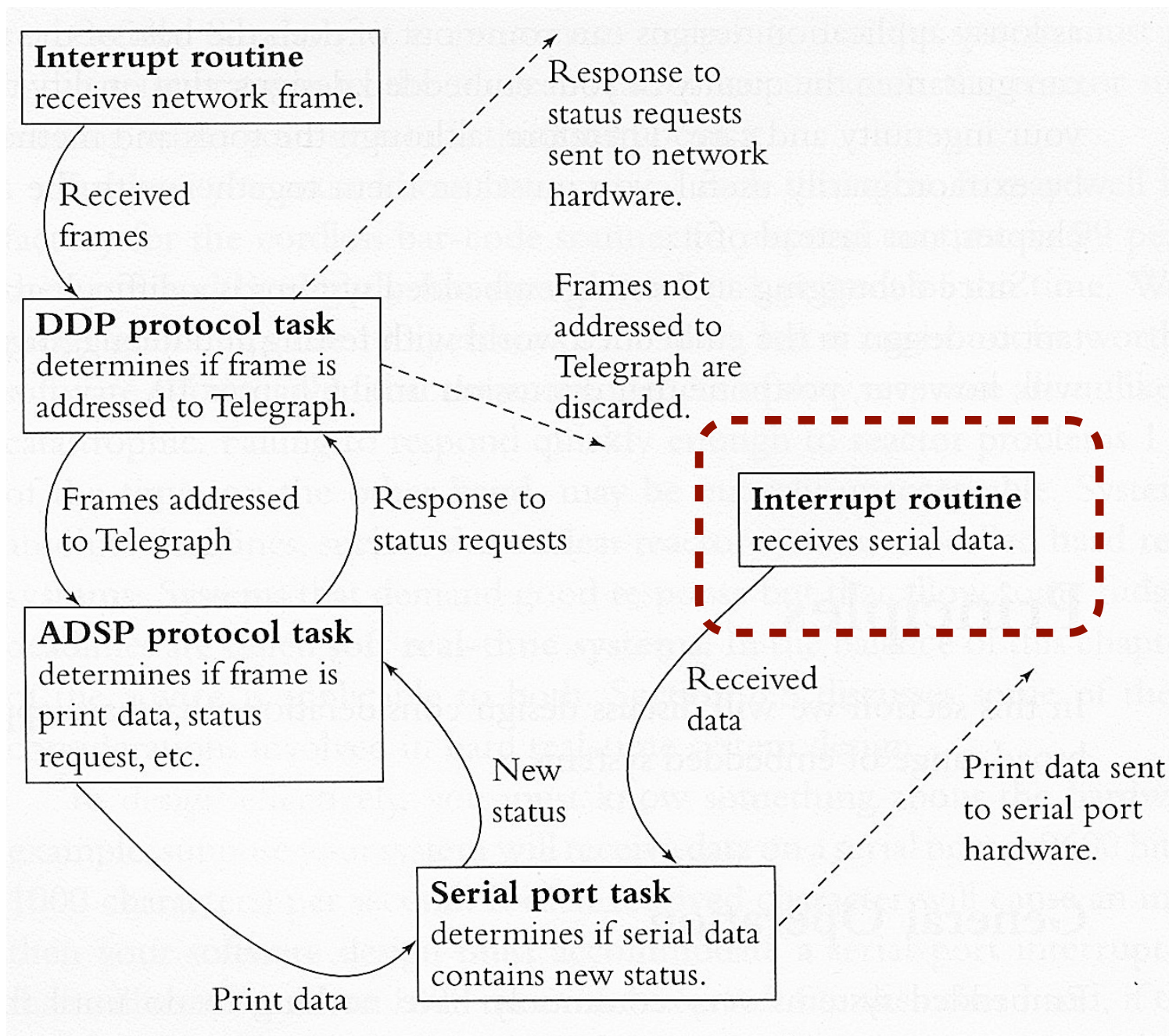
- If the frame contains print data...
- ...the ADSP protocol task sends a message containing the data to the serial-port task
- ...which sends the data to the serial port hardware and through it to the printer.

Unblocking the ADSP protocol which then reads the data packet



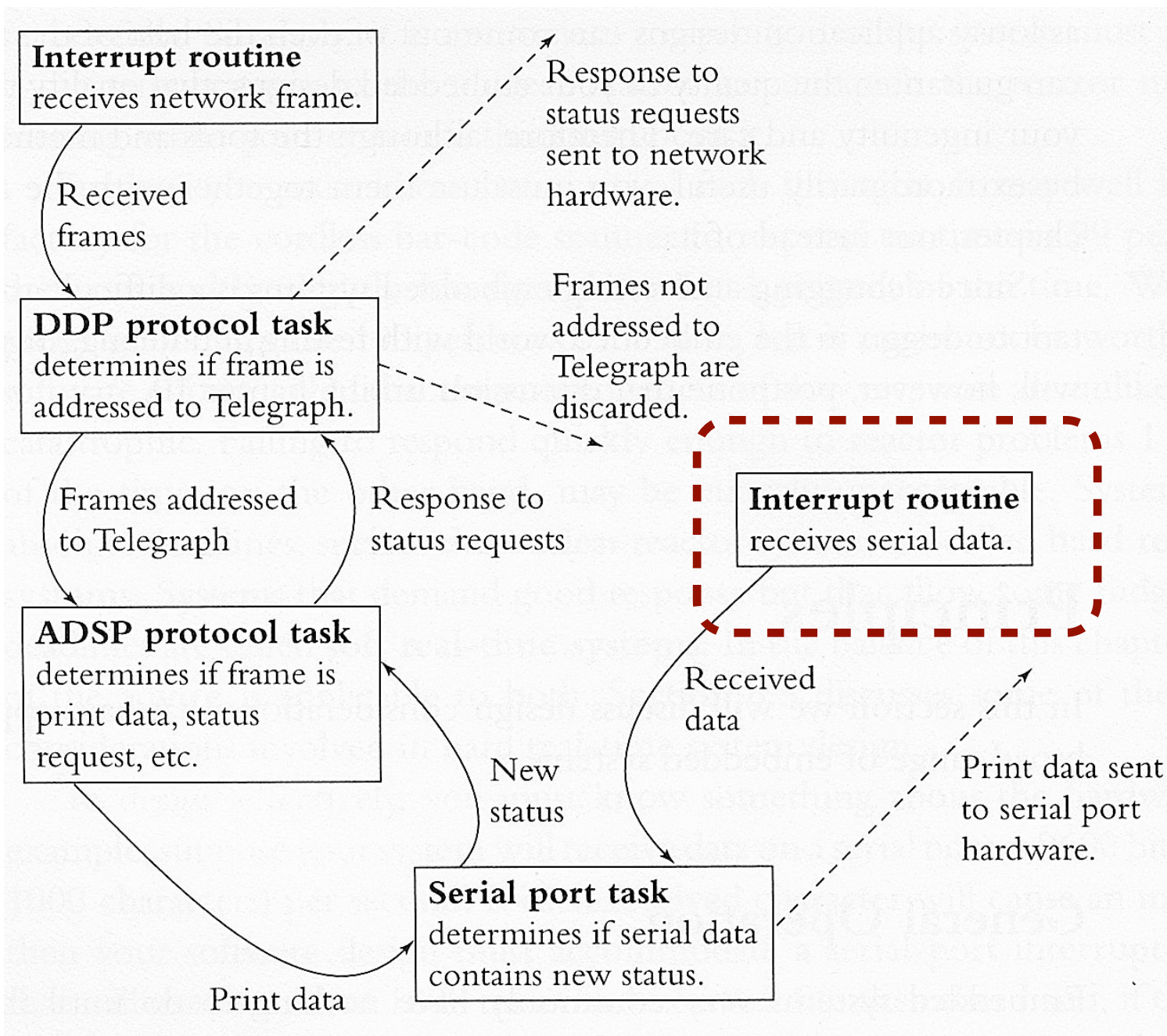
- If the frame contains a request for printer status.
- ... the ADSP protocol task constructs a response frame and sends it to the DDP protocol task to be sent on the network.

Receiving serial data from the printer



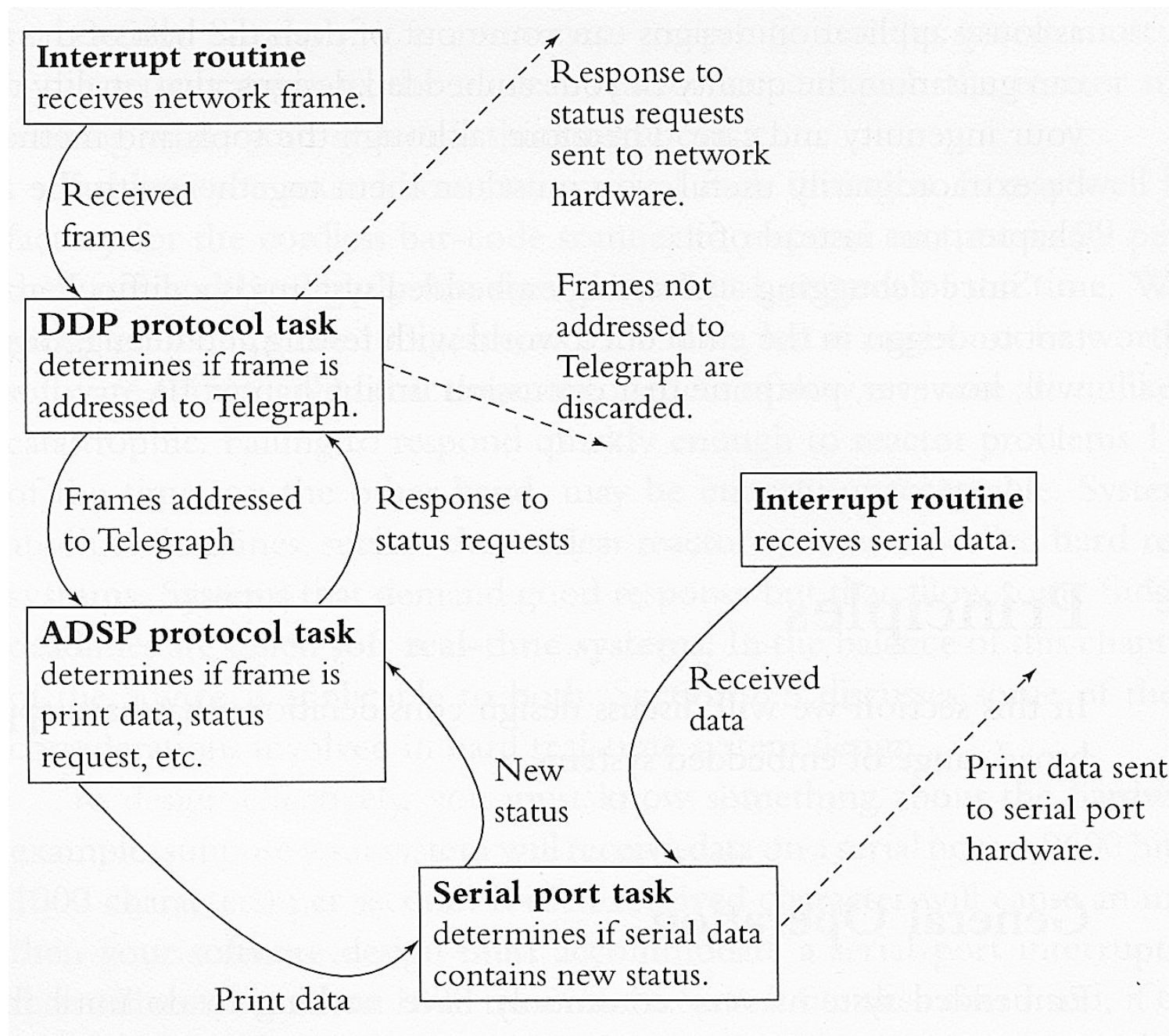
- When the system receives serial data from the printer, the interrupt routine resets the hardware and forwards the data in a message to the serial port task.
- If that data contains printer status, the serial port task forwards the status to the ADSP protocol task.

Receiving serial data from the printer



- The ADSP protocol task stores the status and uses it when responding to later status requests from the network.
- Each time the system receives a network frame or serial port data, an interrupt routine sends a message to one of the tasks, which initiates a chain of events that eventually causes an appropriate response to the received data.

No interrupts



- When no frames or data are arriving, there are no interrupts, and the three tasks in the system remain idle, waiting to receive messages.

Write short interrupt routines

- In general you will be better off if you write short interrupt routines rather than long ones. Why?
- First, since even the lowest-priority interrupt routine is executed in preference to the highest-priority task code, writing longer interrupt routines translates directly into slower task-code response.
- Second, interrupt routines tend to be more bug-prone and harder to debug than task code.

Events and deadlines

- Most **events** require various responses from your software: for example, the system must reset port hardware, save received data, reset the interrupt controller, analyze received data, formulate a response, and so on.
- The **deadlines** for these responses may be quite different.
- Although it may be necessary to reset the port hardware and interrupt controller and to save data immediately, the data analysis and the response are often not nearly as urgent.

System example

Example

Suppose we are writing the software for a system with the following characteristics:

- ▶ The system must respond to commands coming from a serial port. These, always end with a carriage return.
- ▶ Commands arrive one at a time; the next command will not arrive until the system responds to the previous one.
- ▶ The serial port hardware can only store one received character at a time, and characters may arrive quickly.
- ▶ The system can respond to commands relatively slowly.

Wretched way to do it: do all work inside an ISR

- Do all of the work (include processing the commands and sending a command reply) inside an interrupt routine that receives characters.
- That interrupt routine will be long and complex and difficult to debug, and it will slow response for every operation the system does in task code.

Still a bad way to do it: forward every character in a RTOS message

- Interrupt routine that simply forwards every character in an RTOS message to a command parsing task.
- The interrupt routine will be short.
- A practical disadvantage is that the interrupt routine will send a lot of messages to the command parsing task.
- Unfortunately putting messages onto an RTOS queue is not instantaneous.

Possible design compromise

- Interrupt routine that saves the received characters in a buffer and watches for the carriage return that ends each command.
- When the carriage return arrives, the interrupt routine sends a single message to the command parsing task, which reads the characters out of the buffer.

interrupt routine
`vGetCommandCharacter`
stores the incoming
characters in
`a_chCommandBuffer` and
checks each incoming
character for a carriage
return.

carriage return has
arrived... so post that
command into a mailbox.

```
#define SIZEOF_CMD_BUFFER 200
char a_chCommandBuffer[SIZEOF_CMD_BUFFER];
#define MSG_EMPTY ((char *) 0)
char *mboxCommand = MSG_EMPTY;
#define MSG_COMMAND_ARRIVED ((char *) 1)

void interrupt vGetCommandCharacter (void)
{
    static char *p_chCommandBufferTail = a_chCommandBuffer;
    int iError;
    *p_chCommandBufferTail = !!Read rec. character from hardware;
    if (*p_chCommandBufferTail == '\r')
        sc_post (&mboxCommand, MSG_COMMAND_ARRIVED, &iError);
    //Advance the tail pointer and wrap if necessary
    ++p_chCommandBufferTail;
    if (p_chCommandBufferTail == &a_chCommandBuffer[SIZEOF_CMD_BUFFER])
        p_chCommandBufferTail = a_chCommandBuffer;
    !!Reset the hardware as necessary.
}

void vInterpretCommandTask (void)
{
    static char *p_chCommandBufferHead = a_chCommandBuffer;
    int iError;
    while (TRUE)
    {
        //Wait for the next command to arrive.
        scPend (&mboxCommand, WAIT_FOREVER, &iError);
        //We have a command.
        !!Interpret the command at p_chCommandBufferHead
        !!Advance p_chCommandBufferHead past carriage return
    }
}
```

vInterpretCommandTask,
waits on the mailbox;
when it receives a
message, it reads the
characters of the current
command from
a_chCommandBuffer

```
#define SIZEOF_CMD_BUFFER 200
char a_chCommandBuffer[SIZEOF_CMD_BUFFER];
#define MSG_EMPTY ((char *) 0)
char *mboxCommand = MSG_EMPTY;
#define MSG_COMMAND_ARRIVED ((char *) 1)

void interrupt vGetCommandCharacter (void)
{
    static char *p_chCommandBufferTail = a_chCommandBuffer;
    int iError;
    *p_chCommandBufferTail = !!Read rec. character from hardware;
    if (*p_chCommandBufferTail == '\r')
        sc_post (&mboxCommand, MSG_COMMAND_ARRIVED, &iError);
    //Advance the tail pointer and wrap if necessary
    ++p_chCommandBufferTail;
    if (p_chCommandBufferTail==&a_chCommandBuffer[SIZEOF_CMD_BUFFER])
        p_chCommandBufferTail = a_chCommandBuffer;
    !!Reset the hardware as necessary.
}

void vInterpretCommandTask (void)
{
    static char *p_chCommandBufferHead = a_chCommandBuffer;
    int iError;
    while (TRUE)
    {
        //Wait for the next command to arrive.
        sc_pend (&mboxCommand, WAIT_FOREVER, &iError);
        //We have a command.
        !!Interpret the command at p_chCommandBufferHead
        !!Advance p_chCommandBufferHead past carriage return
    }
}
```

**sc_post and sc_pend
functions are **non-
blocking** from the
VRTX system**

```
#define SIZEOF_CMD_BUFFER 200
char a_chCommandBuffer[SIZEOF_CMD_BUFFER];
#define MSG_EMPTY ((char *) 0)
char *mboxCommand = MSG_EMPTY;
#define MSG_COMMAND_ARRIVED ((char *) 1)

void interrupt vGetCommandCharacter (void)
{
    static char *p_chCommandBufferTail = a_chCommandBuffer;
    int iError;
    *p_chCommandBufferTail = !!Read rec. character from hardware;
    if (*p_chCommandBufferTail == '\r')
        sc_post (&mboxCommand, MSG_COMMAND_ARRIVED, &iError);
    //Advance the tail pointer and wrap if necessary
    ++p_chCommandBufferTail;
    if (p_chCommandBufferTail==&a_chCommandBuffer[SIZEOF_CMD_BUFFER])
        p_chCommandBufferTail = a_chCommandBuffer;
    !!Reset the hardware as necessary.
}

void vInterpretCommandTask (void)
{
    static char *p_chCommandBufferHead = a_chCommandBuffer;
    int iError;
    while (TRUE)
    {
        //Wait for the next command to arrive.
        sc_pend (&mboxCommand, WAIT_FOREVER, &iError);
        //We have a command.
        !!Interpret the command at p_chCommandBufferHead
        !!Advance p_chCommandBufferHead past carriage return
    }
}
```

How many tasks?

- One of the first problems in an embedded-system design is to divide your system's work into RTOS tasks.
- Obvious question is "Am I better off with more tasks or with fewer tasks?"
- What are the advantages and disadvantages of having more tasks?

Advantages of having lots of tasks

- With more tasks you have **better control** of the relative response times of the different parts of your system's work.
- With more tasks your system can be somewhat **more modular**. Using a separate task for each device allows for cleaner code.
- With more tasks you can sometimes **encapsulate data** more effectively. Only the code in that task needs access to the variables.

Disadvantages of having lots of tasks

- With more tasks you are likely to have **more data shared** among two or more tasks... so more microprocessor time is lost handling the semaphores.
- With more tasks you are likely to have **more requirements** to pass messages from one task to another through pipes, mailboxes, queues, and so on. This will also translate into more microprocessor time and more chances for bugs.
- Each task requires a stack; therefore, with more tasks you need **more memory**.

Even more disadvantages of having more tasks

- Each time the RTOS switches tasks, a certain amount of microprocessor **time evaporates** saving the context of the task that is stopping and restoring the context of the task that is about to run.
- More tasks probably means more calls to the RTOS. RTOS vendors promote their products by telling you how fast they can switch tasks, put messages into mailboxes, set events, and so on. Your system **runs slower** if it calls many RTOS functions
- ... Other things being equal, use as few tasks as you can; add more tasks to your design only for clear reasons.

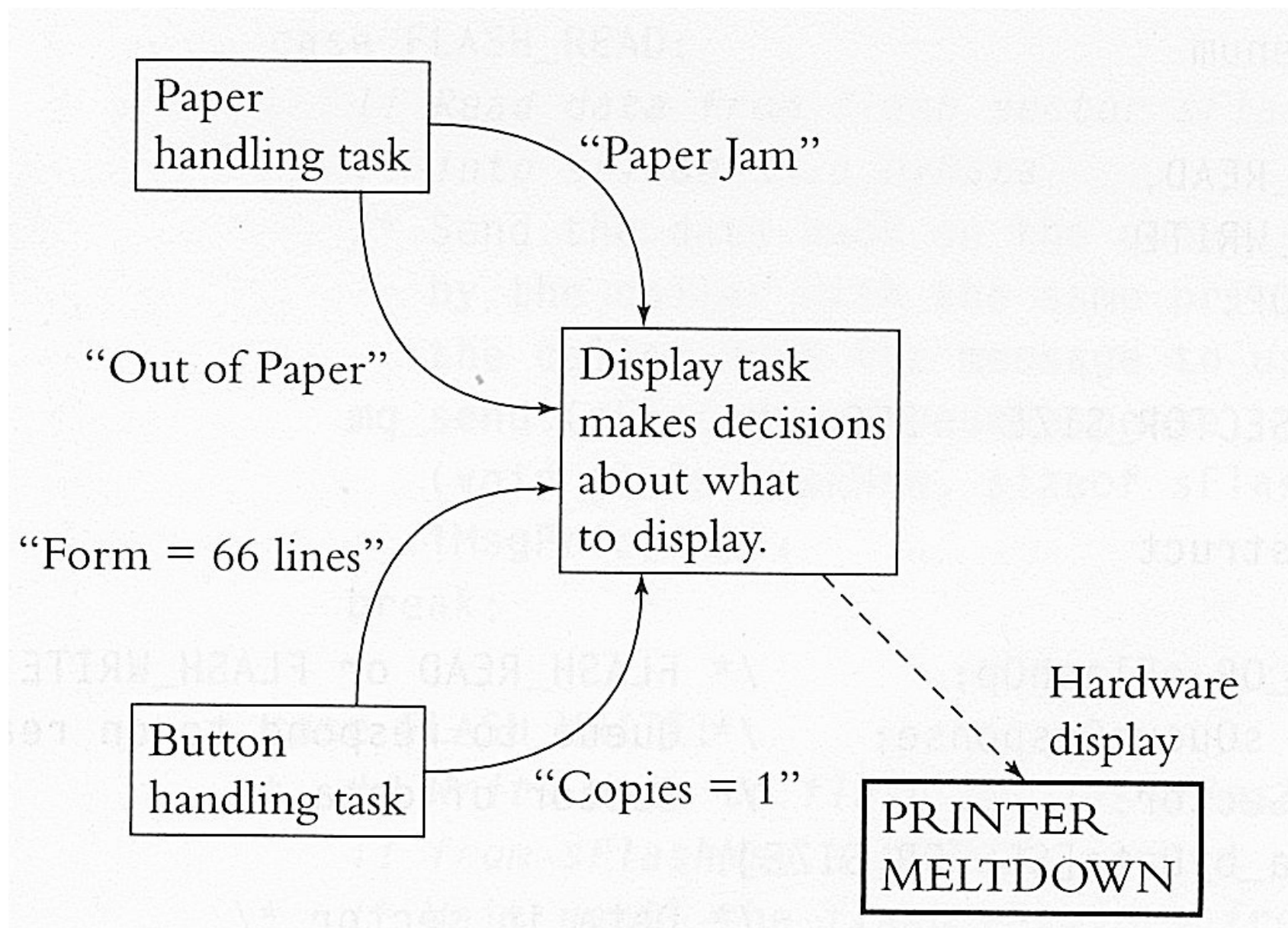
You need tasks for priority

- Let's examine some situations in which it makes sense to add more tasks to your system design.
- First, the obvious **advantage of the RTOS architecture over the others is the improved control of task code response.**
- One obvious reason for having multiple tasks is to be able to assign higher priorities to parts of the work with tighter response time requirements.

You need tasks for encapsulation

- It often makes sense to have a separate task to deal with hardware shared by different parts of the system.
- For example, the printer's display is shared by buttons and the printer mechanism.
 - ▶ A single task that controls the hardware display can solve these problems.
 - ▶ When other tasks in the system have information to display, they will send messages to the display task.
 - ▶ The RTOS will ensure that messages sent to the display task are queued properly.

A separate task helps control shared hardware



Another encapsulation example

If various parts of a system need to store data in a flash memory, a single task responsible for dealing with the flash memory hardware can simplify your system.

```

void vHandleFlashTask (void)
{
    //Handle of our input queue
    mdt_q sQueueOurs;
    //Message telling us what to do.
    FLASH_MSG sFlashMsg;
    //Priority of received message
    int iMsgPriority;
    sQueueOurs = mq_open("FLASH", O_RDONLY, 0, NULL);

    while (TRUE)
    {
        //Get the next request.
        mq_receive (sQueueOurs, (void *) &sFlashMsg,
            sizeof sFlashMsg, &iMsgPriority);
        switch (sFlashMsg.eFlashOp)
        {
            case FLASH_READ:
                !!Read data from flash sector sFlashMsg.iSector
                !! into sFlashMsg.a_byData
                //Send the data back on the queue specified
                //by the caller with the same priority as
                //the caller sent the message to us.

                mq_send (sFlashMsg.sQueueResponse,
                    (void *) &sFlashMsg, sizeof sFlashMsg,
                    iMsgPriority);

                break;

            case FLASH_WRITE:
                !!Write data to flash sector sFlashMsg.iSector
                !!from sFlashMsg.a_byData
                //Wait until the flash recovers from writing.
                nanosleep (!! Amount of time needed for flash);
                break;
        }
    }
}

```

```

typedef enum { FLASH_READ, FLASH_WRITE } FLASH_OP;

#define SECTOR_SIZE 256
typedef struct{
    //FLASH_READ or FLASH_WRITE
    FLASH_OP eFlashOp;
    //Queue to respond to on reads
    mdt_q sQueueResponse;
    //Sector of data
    int iSector;
    //Data in sector
    BYTE a_byData[SECTOR_SIZE];
} FLASH_MSG;

void vInitFlash (void){
    //This function must be called before any other,
    //preferably in the startup code.
    //Create a queue called 'FLASH' for input
    //to this task
    mq_open ("FLASH", O_CREAT, 0, NULL);
}

void vTaskA (void){
    //Handle of flash task input queue
    mdt_q sQueueFlash;
    //Message to the flash routine.
    FLASH_MSG sFlashMsg;
    (...)
    //We need to write data to the flash
    //Set up the data in the message structure
    !!Write data to sFlashMsg.a_byData
    sFlashMsg.iSector = FLASH_SECTOR_FOR_TASK_A;
    sFlashMsg.eFlashOp = FLASH_WRITE;
    //Open queue and snd the message with priority 5
    sQueueFlash = mq_open ("FLASH", O_WRONLY, 0,
        NULL);

    mq_send (sQueueFlash, (void *) &sFlashMsg,
        sizeof sFlashMsg, 5);

    mq_close (sQueueFlash);
    (...)
}

```



```

void vHandleFlashTask (void)
{
    //Handle of our input queue
    mdt_q sQueueOurs;
    //Message telling us what to do.
    FLASH_MSG sFlashMsg;
    //Priority of received message
    int iMsgPriority;
    sQueueOurs = mq_open("FLASH", O_RDONLY, 0, NULL);

    while (TRUE)
    {
        //Get the next request.
        mq_receive (sQueueOurs, (void *) &sFlashMsg,
            sizeof sFlashMsg, &iMsgPriority);
        switch (sFlashMsg.eFlashOp)
        {
            case FLASH_READ:
                !!Read data from flash sector sFlashMsg.iSector
                !! into sFlashMsg.a_byData
                //Send the data back on the queue specified
                //by the caller with the same priority as
                //the caller sent the message to us.

                mq_send (sFlashMsg.sQueueResponse,
                    (void *) &sFlashMsg, sizeof sFlashMsg,
                    iMsgPriority);

                break;

            case FLASH_WRITE:
                !!Write data to flash sector sFlashMsg.iSector
                !!from sFlashMsg.a_byData
                //Wait until the flash recovers from writing.
                nanosleep (!! Amount of time needed for flash);
                break;
        }
    }
}

```

```

typedef enum { FLASH_READ, FLASH_WRITE } FLASH_OP;

#de
typ
/
F
/
m
/
i
/
B
} FLASH_MSG;

```

- Any other task in the system wanting to write to the flash sends a message containing a **FLASH_MSG** structure to **vHandleFlashTask**.

```

void vInitFlash (void){
    //This function must be called before any other,
    //preferably in the startup code.
    //Create a queue called 'FLASH' for input
    //to this task
    mq__open ("FLASH", O_CREAT, 0, NULL);
}

void vTaskA (void){
    //Handle of flash task input queue
    mdt_q sQueueFlash;
    //Message to the flash routine.
    FLASH_MSG sFlashMsg;
    (...)
    //We need to write data to the flash
    //Set up the data in the message structure
    !!Write data to sFlashMsg.a_byData
    sFlashMsg.iSector = FLASH_SECTOR_FOR_TASK_A;
    sFlashMsg.eFlashOp = FLASH_WRITE;
    //Open queue and snd the message with priority 5
    sQueueFlash = mq_open ("FLASH", O_WRONLY, 0,
        NULL);

    mq_send (sQueueFlash, (void *) &sFlashMsg,
        sizeof sFlashMsg, 5);

    mq_close (sQueueFlash);
    (...)
}

```



```

void vHandleFlashTask (void)
{
    //Handle of our input queue
    mdt_q sQueueOurs;
    //Message telling us what to do.
    FLASH_MSG sFlashMsg;
    //Priority of received message
    int iMsgPriority;
    sQueueOurs = mq_open("FLASH", O_RDONLY, 0, NULL);

    while (TRUE)
    {
        //by the caller with the same priority as
        //the caller sent the message to us.

        mq_send (sFlashMsg.sQueueResponse,
                (void *) &sFlashMsg, sizeof sFlashMsg,
                iMsgPriority);

        break;

        case FLASH_WRITE:
            !!Write data to flash sector sFlashMsg.iSector
            !!from sFlashMsg.a_byData
            //Wait until the flash recovers from writing.
            nanosleep (!! Amount of time needed for flash);
            break;
    }
}

```

• Any other task in the system wanting to write to the flash sends a message containing a **FLASH_MSG** structure to **vHandleFlashTask**.

```

typedef enum { FLASH_READ, FLASH_WRITE } FLASH_OP;

#define SECTOR_SIZE 256
typedef struct{
    //FLASH_READ or FLASH_WRITE
    FLASH_OP eFlashOp;
    //Queue to respond to on reads
    mdt_q sQueueResponse;
    //Sector of data
    int iSector;
    //Data in sector
    BYTE a_byData[SECTOR_SIZE];
} FLASH_MSG;

void vInitFlash (void){
    //This function must be called before any other,
    //preferably in the startup code.
    //Create a queue called 'FLASH' for input
    //to this task
    mq_open ("FLASH", O_CREAT, 0, NULL);
}

void vTaskA (void){
    //Handle of flash task input queue
    mdt_q sQueueFlash;
    //Message to the flash routine.
    FLASH_MSG sFlashMsg;
    (...)
    //We need to write data to the flash
    //Set up the data in the message structure
    !!Write data to sFlashMsg.a_byData
    sFlashMsg.iSector = FLASH_SECTOR_FOR_TASK_A;
    sFlashMsg.eFlashOp = FLASH_WRITE;
    //Open queue and snd the message with priority 5
    sQueueFlash = mq_open ("FLASH", O_WRONLY, 0,
                            NULL);

    mq_send (sQueueFlash, (void *) &sFlashMsg,
            sizeof sFlashMsg, 5);

    mq_close (sQueueFlash);
    (...)
}

```

```

void vHandleFlashTask (void)
{
    //Handle of our input queue
    mdt_q sQueueOurs;
    //Message telling us what to do.
    FLASH_MSG sFlashMsg;
    //Priority of received message
    int iMsgPriority;
    sQueueOurs = mg_open("FLASH", O_RDONLY, 0, NULL);

    while (TRUE)
    {
        //Get the next request.
        mq_receive (sQueueOurs, (void *) &sFlashMsg,
        sizeof sFlashMsg, &iMsgPriority);
        switch (sFlashMsg.eFlashOp)
        {
            case FLASH_READ:
                !!Read data from flash sector sFlashMsg.iSector
                !! into sFlashMsg.a_byData
                //Send the data back on the queue specified
                //by the caller with the same priority as
                //the caller sent the message to us.

                mq_send (sFlashMsg.sQueueResponse,
                        (void *) &sFlashMsg, sizeof sFlashMsg,
                        iMsgPriority);

                break;

            case FLASH_WRITE:
                !!Write data to flash sector sFlashMsg.iSector
                !!from sFlashMsg.a_byData
                //Wait until the flash recovers from writing.
                nanosleep (!! Amount of time needed for flash);
                break;
        }
    }
}

```

```

typedef enum { FLASH_READ, FLASH_WRITE } FLASH_OP;

#define ...
typedef ...
...
FLASH_MSG;

```

- The vHandleFlashTask task copies the contents of a `_byData` in the `FLASH_MSG` structure into the sector indicated by `iSector`.

```

void ...
...
FLASH_MSG sFlashMsg;

```

- Any task wishing to read from the flash sends a message to vHandleFlashTask containing a `FLASH_MSG` structure with `eFlashOp` set to `FLASH_READ`.

```

//Message to the flash routine.
FLASH_MSG sFlashMsg;
...
mq_send (sQueueFlash, (void *) &sFlashMsg,
        sizeof sFlashMsg, 5);
mq_close (sQueueFlash);
(...)

```

- The vHandleFlashTask task will mail the data from the flash back to the queue specified by the `sQueueResponse` element.

Other tasks you might or might not need

- Good idea to have many small tasks, so that each task is simple. Keep in mind that the time your system spends switching tasks will eat into your throughput.
- Good idea: Have separate tasks for work that needs to be done in response to separate stimulus.
- However, if Task1 and Task2 share data or must communicate with one another, the problems that arise from that may make your code more complicated.

```
void task1 (void)
{
    while (TRUE)
    {
        !!Wait for stimulus 1
        !!Deal with stimulus 1
    }
}

void task2 (void)
{
    while (TRUE)
    {
        !!Wait for stimulus 2
        !!Deal with stimulus 2
    }
}
```

Recommended task structure

- You should use this task structure most of the time.
- The task remains in an infinite loop, waiting for an RTOS signal that there is something for it to do.
- That signal is most commonly in the form of a message from a queue.
- This task declares its own private data.
- **Advantages?**

```
!!Private static data is declared here

void vTaskA (void)
{
    !!More private data declared here,
    !!either static or on the stack
    !!Initialization code, if needed.

    while (FOREVER)
    {
        !!Wait for a system signal
        !!...(event, queue message, etc.)
        switch (!!type of signal)
        {
            case !! signal type 1:
                (...)
                break;
            case !! signal type 2
                (...)
                break;
            (...)
        }
    }
}
```


Advantages of this recommended task structure

- Advantages:

- ▶ The task blocks in only one place. When another task puts a request on this task's queue, this task is not off waiting for some other event that may or may not happen in a timely fashion.
- ▶ When there is nothing for this task to do, its input queue will be empty, and the task will block and use up no microprocessor time.

```
!!Private static data is declared here
```

```
void vTaskA (void)
```

```
{
```

```
    !!More private data declared here,  
    !!either static or on the stack  
    !!Initialization code, if needed.
```

```
while (FOREVER)
```

```
{
```

```
    !!Wait for a system signal  
    !!...(event, queue message, etc.)  
    switch (!!type of signal)
```

```
{
```

```
    case !! signal type 1:  
        (...)
```

```
    break;
```

```
    case !! signal type 2:  
        (...)
```

```
    break;
```

```
    (...)
```

```
}
```

```
}
```

```
}
```

More advantages of this recommended task structure

- This task does not have public data that other tasks can share; other tasks that wish to see or change its private data write requests into the queue, and this task handles them.
- There is no concern that other tasks using the data use semaphores properly; there is no shared data, and there are no semaphores.

```
!!Private static data is declared here

void vTaskA (void)
{
    !!More private data declared here,
    !!either static or on the stack
    !!Initialization code, if needed.

    while (FOREVER)
    {
        !!Wait for a system signal
        !!...(event, queue message, etc.)
        switch (!!type of signal)
        {
            case !! signal type 1:
                (...)
                break;
            case !! signal type 2
                (...)
                break;
            (...)
        }
    }
}
```

Avoid creating and destroying tasks

- Every RTOS allows you to create tasks as the system is starting.
- Most RTOSs also allow you to create and destroy tasks while the system is running.
- First, the functions that create and destroy tasks are typically the **most time-consuming** functions in the RTOS.
- Second, whereas creating a task is a relatively reliable operation, it can be **difficult to destroy a task** without leaving little pieces lying around to cause bugs.
- The alternative to creating and destroying tasks is to create all of the tasks you'll need at system startup. Later, if a task has nothing to do, it can block for as long as necessary on its input queue.

Timings of an RTOS on a 20 MHz Intel 80386

Service	Time
Get a semaphore	10 microseconds (μsec)
Release a semaphore	6—38 μsec
Switch tasks	17-35 μsec
Write to a queue	49-68 μsec
Read from a queue	12-38 μsec
Create a task	158 μsec
Destroy a task	36—57 μsec

Consider turning time-slicing off

- RTOS scheduler always runs the highest-priority ready task.
- Consider two or more ready tasks have the same priority and no other ready task has a higher priority. RTOSs normally time-slice among those tasks, giving the microprocessor to each tasks for a short period of time
- RTOSs also allow you to turn this option off.
- **Time-slicing** causes more task switches and therefore cuts throughput.
- Unless you can pinpoint a reason that it will be useful in your system, you're probably better off without it.

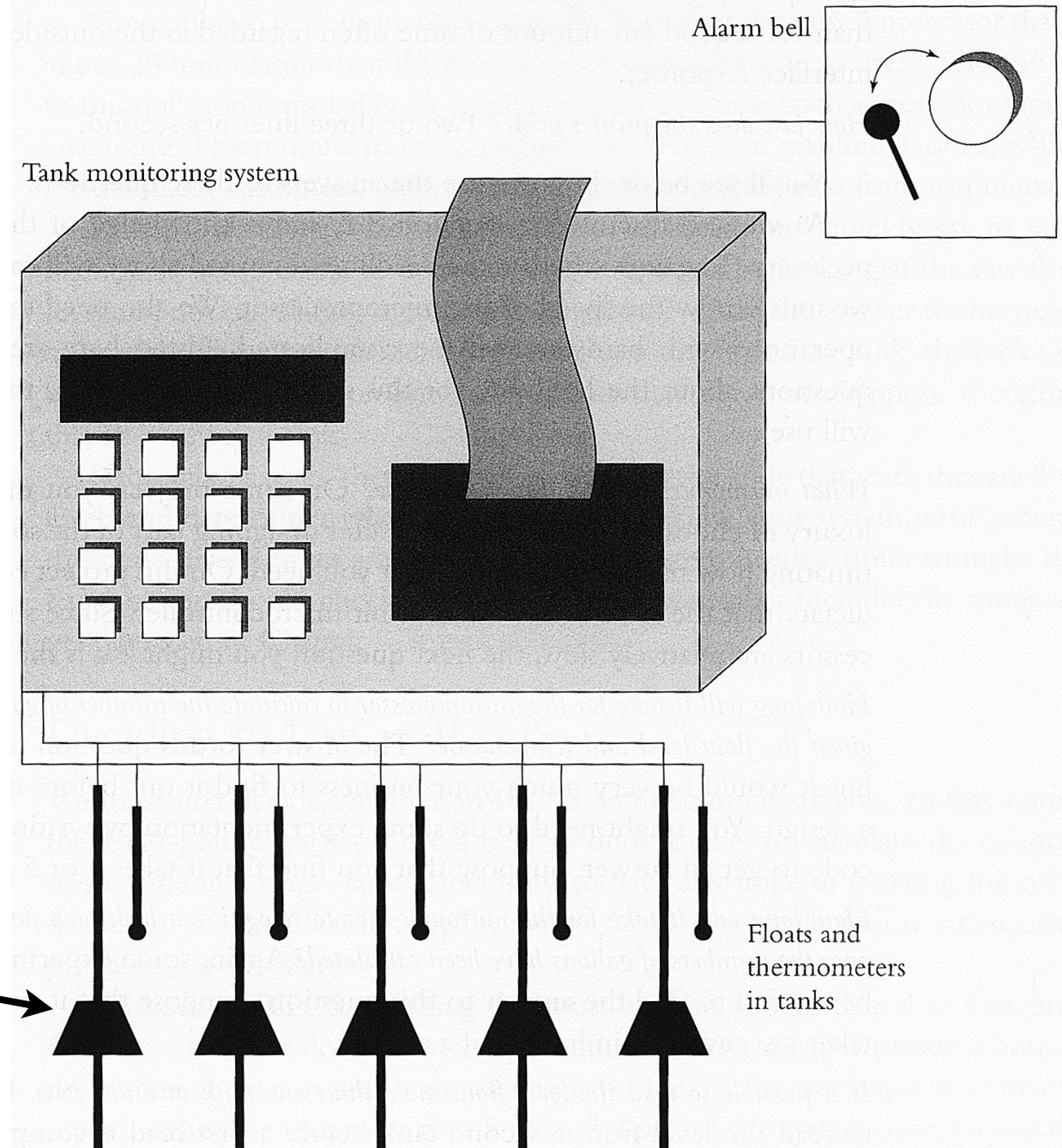
Consider restricting your use of the RTOS

- Most RTOSs, even fairly small ones, offer more services than you are likely to need on any given project.
- Many RTOSs allow you to configure them and to remove any services that you do not use
- You can save memory space by figuring out a subset of the RTOS features that is sufficient for your system and using only that.
- Many embedded-system designers prefer to put a shell around the RTOS and have all of the rest of their code call the shell rather than directly call the RTOS. It makes the code more portable from one RTOS to another, because only the shell needs be rewritten.

An example: lets design an embedded system

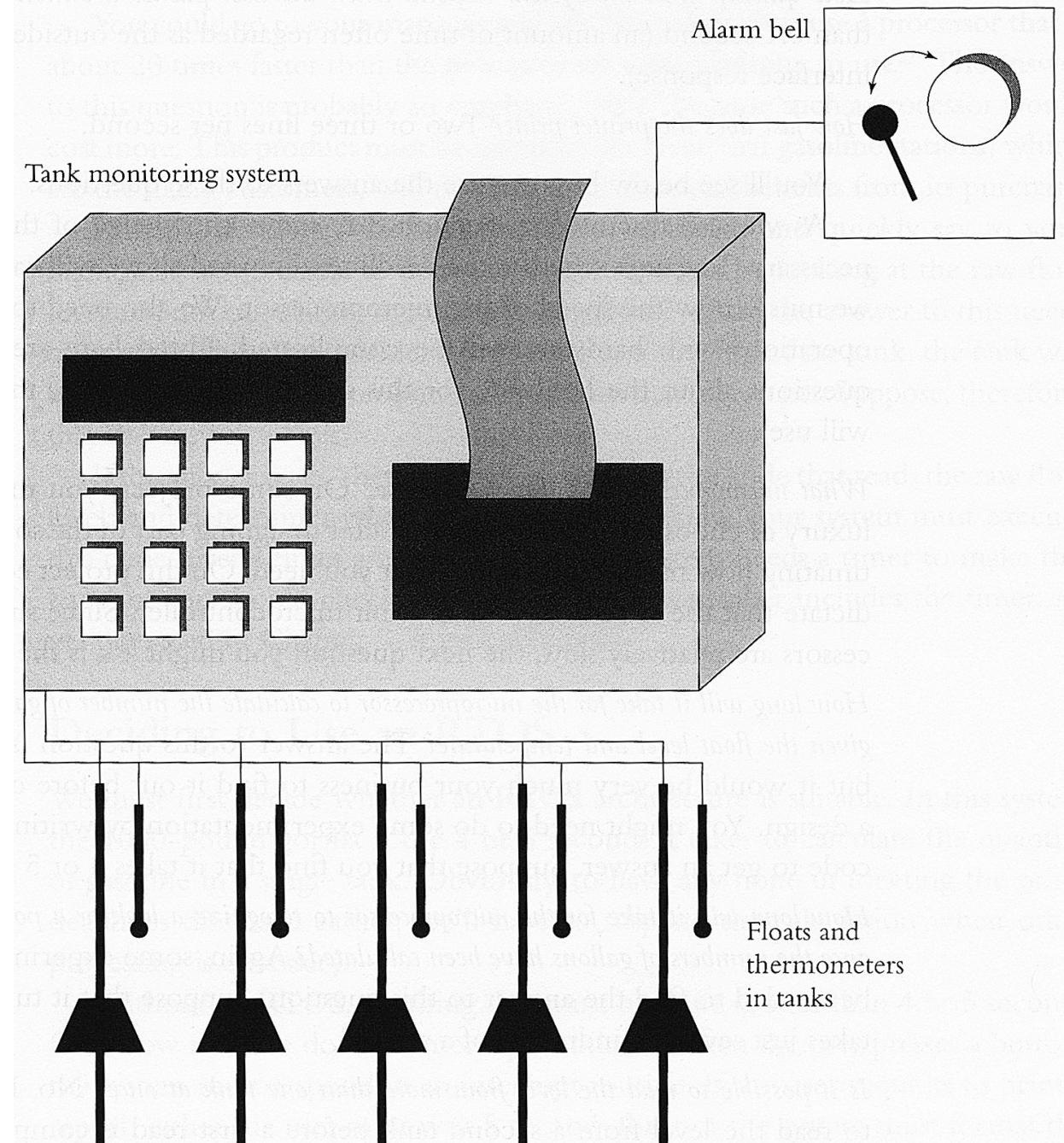
Tank monitoring system

- The purpose of this example is to show you the considerations that go into the process.
- This system monitors up to eight underground tanks by reading thermometers and the levels of floats.



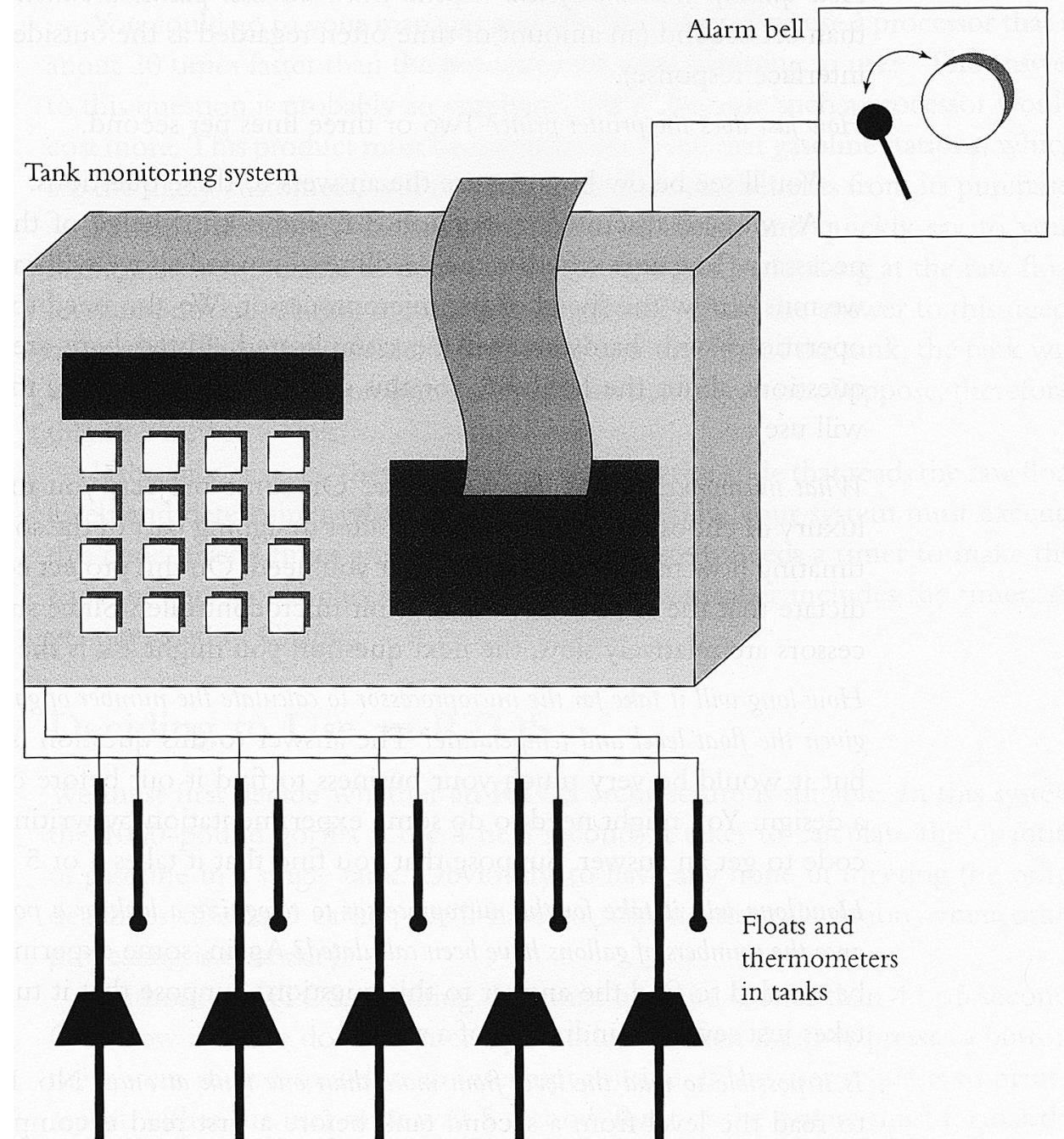
Functionality of the tank monitoring system

- When the hardware has obtained a new float reading, it interrupts; the microprocessor can read the level.
- The microprocessor can read the temperature in any tank at any time.
- The system must pay special attention to tanks in which the level is rising rapidly and set off the alarm if such a tank gets close to full and the level is still rising.



Interface for the tank monitoring system

- The UI consists of a keypad, a liquid crystal display, and a thermal printer.
- With the keypad, the user can tell the system to display various information such as the levels in the tanks
- The system will override the user's display preference and show messages if it detects a leak or a overflow.
- The system also has a connector to a loud alarm bell.
- The printer can accept one line of a report at a time.



Some initial questions

- When the float in one tank is rising rapidly, how often do we need to read it? *Often.*
- How quickly must the system respond when the user pushes a button? *In no more than 0.1 second.*
- How fast does the printer print? *Two or three lines per sec.*
- How long does it take to calculate the quantity of gasoline in a tank? *4 or 5 seconds.*
- What microprocessor will this system use? *Cost constraints dictate that the system run on an 8-bit micro-controller.*

More questions

- How long will it take for the microprocessor to calculate the number of gallons in a tank, given the float level and temperature? *Figure it out before designing system.*
- How long will it take for the microprocessor to recognize a leak or a potential overflow? *Figure it out before designing system.*
- Is it possible to read the level from more than one tank at once? *No. In fact, trying to read the level from a second tank before a first read is complete will mess up your results.*
- How difficult is it for software to turn the alarm bell on and off?

Deciding to use an RTOS

- Decide whether an RTOS architecture is suitable.
- For any hope of meeting the other deadlines discussed earlier, we'll have to suspend the calculation when other processing is necessary.
- Can you build a system that does all this work in interrupt routines? *Probably yes.*
- Will it be easy to build a system that does all this work in interrupt routines? *Probably not.*
- Using an RTOS looks like a better solution in this case

Divide the work of the system into individual tasks

- We need a level calculation task that takes as input the levels and temperatures in the tanks, calculates how much gasoline is in each tank, and perhaps detects leaks by looking at previous gasoline levels.
- Since this takes 4 or 5 seconds for each tank, and since other things must happen more quickly than that, this is the classic RTOS situation calling for a separate, low-priority task.
- One-task-per-tank plan creates code problems.
- Disadvantage of the one-task-for-all-tanks is that the task must have code to figure out which tank to deal with next.

Overflow-detection must be separate from the level-calculation

- Overflow detection must happen at a higher priority than the level calculation and leak detection processes; therefore, it must be in a separate task.
- Both the level calculation task and the overflow detection task must read from the float hardware; must make sure that they do not fight over it.
- You could use a semaphore to ensure that only one task tries to read from the floats at one time.
- You could set up a separate float hardware task and have the other tasks queue messages to that task requesting service.

Button handling task

- Since some commands require button presses, so we need a state machine to keep track of the buttons already pressed.
- Can be done in an interrupt routine... but it will be long and complicated.
- Various tasks have messages to be displayed: For example, the level calculation task (when it detects a leak), the overflow detection task, and the button handling task.
- We need a separate display task. For example, to handle the situation when the user presses a button right after a leak.

A Semaphore can't protect the display properly

```
{
    (...)
    if (!!Leak detected)
    {
        TakeSemaphore (SEMAPHORE_DISPLAY);
        !! Write "LEAK!!!" to display
        ReleaseSemaphore (SEMAPHORE_DISPLAY);
    }
    (...)
}

void vButtonHandlingTask (void)
{
    (...)
    if (!! Button just pressed necessitates a prompt)
    {
        TakeSemaphore (SEMAPHORE_DISPLAY);
        !!Write "Press next button" to display
        ReleaseSemaphore (SEMAPHORE_DISPLAY);
    }
    (...)
}
```

Alarm bell

- The alarm bell is another piece of shared hardware.
- The level calculation and overflow detection tasks can turn it on, and the button task can turn it OFF.
- Turning the bell ON and OFF is **atomic**.
- If the system discovers a second leak or an overflow right after the user turns OFF the bell, it should turn the bell back on again to call attention to the second problem.
- It probably makes sense to let any task turn the bell ON or OFF directly.
- A separate alarm bell task is not useful.

Printer

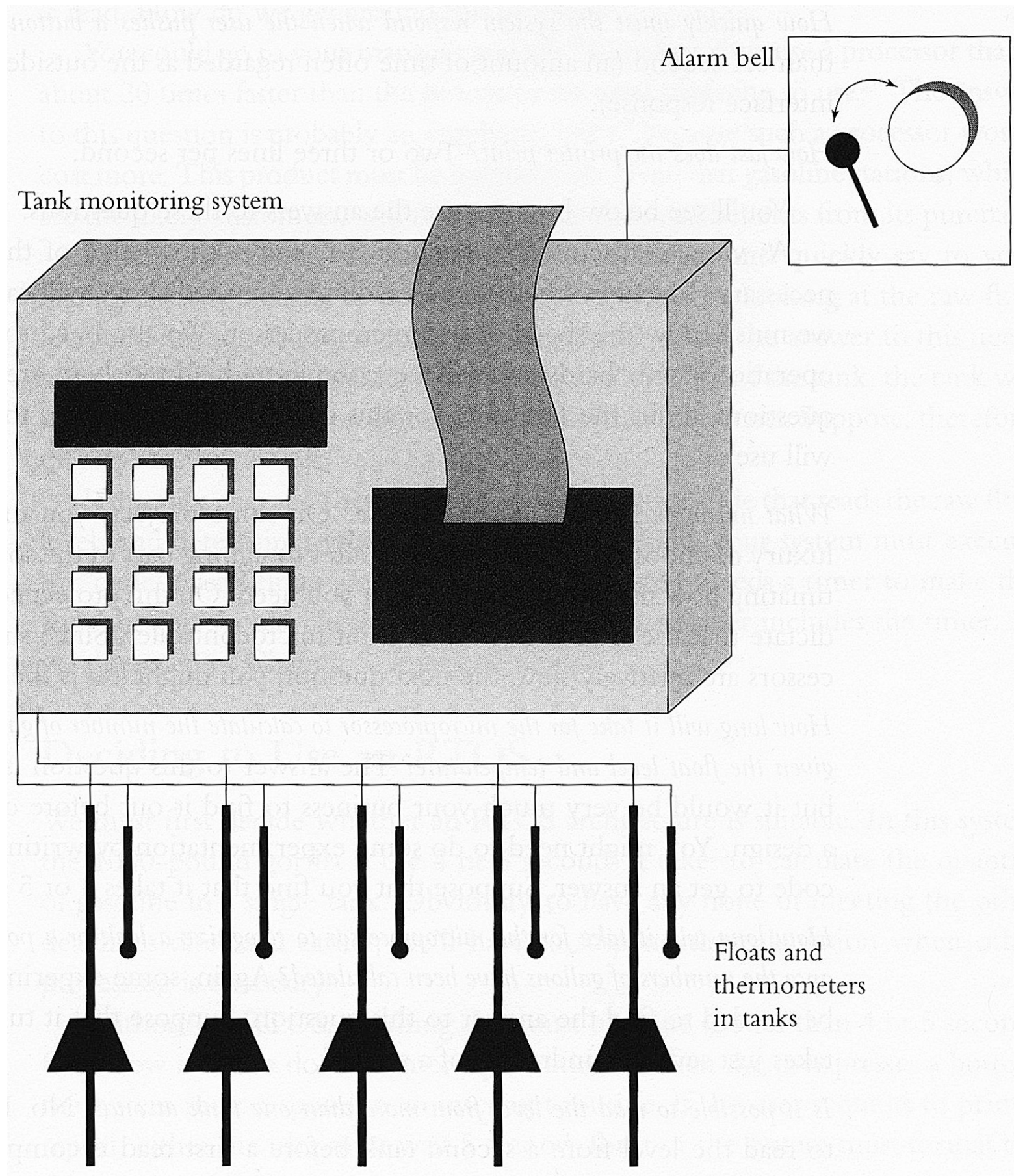
- Since the printer interrupts after printing each line, we can write an interrupt routine to send successive lines of each report to the printer.
- First, if reports might take more than one-tenth of a second to format, then the formatting process must be in a task with lower priority than the button handling task so as not to interfere with the required button response.
- Second, the complication of maintaining a print queue may make a separate task easier to deal with.

Starting the system

- Interrupt routines start sending signals through the system, telling tasks to do their work.
- Whenever the user presses a button, the button hardware interrupts the microprocessor.
- The button interrupt routine can send a message to the button handling task, which can interpret the commands and then forward messages on to the display task and the printer task as necessary.

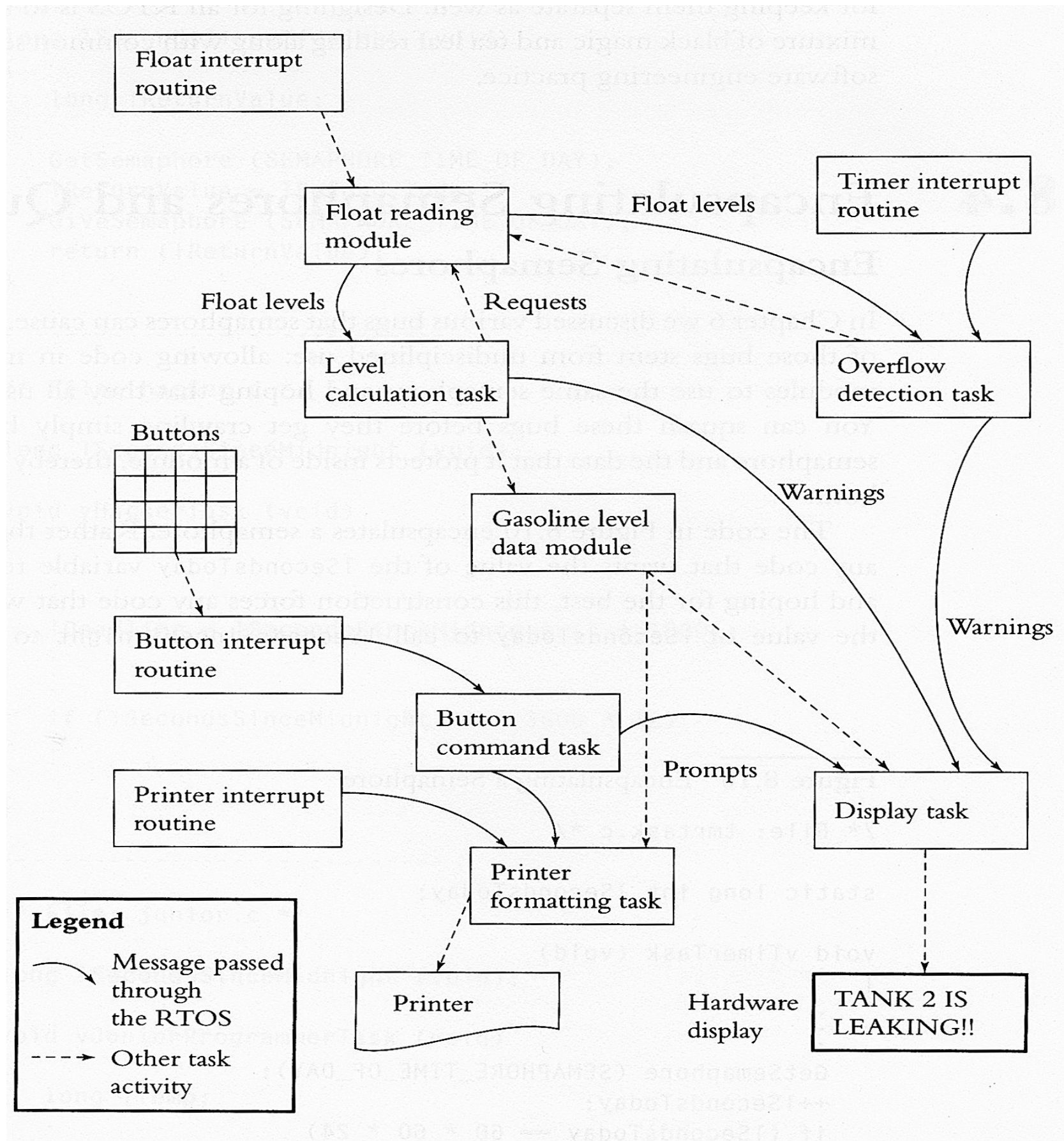
Shared data problems

- The gasoline levels data is shared by several tasks: the level calculation task (leak detection), the display task (show data to user), and the print formatting task (prints data)
- Should we protect the data with a semaphore or should we create a separate task responsible for keeping the data consistent for the other tasks?
- "What is the longest that any one task will hold on to the semaphore?" (about 1 ms)
- "Can every other task wait that long?" (Yes)
- So we do not need an additional task...



Tasks in the underground tank system

Task	Priority	Reason for Creating This Task
Level calculation task	Low	Other processing is much higher priority than this calculation, and this calculation is a microprocessor hog.
Overflow detection task	High	This task determines whether there is an overflow; it is important that this task operate quickly.
Button handling task	High	This task controls the state machine that operates the user interface, relieving the button interrupt routine of that complication, but still responding quickly.
Display task	High	Since various other tasks use the display, this task makes sure that they do not fight over it.
Print formatting task	Med	Print formatting might take long enough that it interferes with the required response to the buttons. Also, it may be simpler to handle the print queue in a separate task.



Encapsulating semaphores and queues

Review of semaphore problems

- Forgetting to take the semaphore: Semaphores only work if every task that accesses the shared data, uses the semaphore.
- Forgetting to release the semaphore: If any task fails to release the semaphore, then every other task that ever uses the semaphore will sooner or later be blocked.
- Taking the wrong semaphore: If you are using multiple semaphores, then taking the wrong one is bad.
- Holding a semaphore for too long: Whenever one task takes a semaphore, every other task that subsequently wants that semaphore has to wait until the semaphore is released.

Encapsulating semaphores

- Most of semaphore bugs stem from undisciplined use.
- For example ... allowing code in many different modules to use the same semaphore and hoping that they all use it correctly.
- You can squash these bugs by hiding the semaphore and the data that it protects inside of a module, thereby encapsulating both.

Code that encapsulates a semaphore

```
/* File: tmrtask.c */
static long int lSecondsToday;

void vTimerTask (void)
{
    (...)
    GetSemaphore (SEMAPHORE_TIME_OF_DAY);
    ++lSecondsToday;
    if (lSecondsToday == 60*60*24) lSecondsToday = 0L;
    GiveSemaphore (SEMAPHORE_TIME_OF_DAY);
    (...)
}

long lSecondsSinceMidnight (void)
{
    long lReturnValue;
    GetSemaphore (SEMAPHORE_TIME_OF_DAY);
    lReturnValue = lSecondsToday;
    GiveSemaphore (SEMAPHORE_TIME_OF_DAY);
    return (lReturnValue);
}
```

```
/* File: hacker.c */
long lSecondsSinceMidnight (void);

void vHackerTask (void)
{
    (...)
    lDeadline = lSecondsSinceMidnight () + 1800L;
    (...)
    if (lSecondsSinceMidnight () > 3600 * 12)
        (...)
}

/* File: junior.c */
long lSecondsSinceMidnight (void);

void vJuniorProgrammerTask (void)
{
    long lTemp;
    (...)
    lTemp = lSecondsSinceMidnight ();
    for (1 = lTemp; 1 < lTemp + 10; ++1)
        (...)
}
```

We don't want tasks to access the variable `lSecondsToday` directly.
If you want the value of `lSecondsToday` you must use `lSecondsSinceMidnight`.

Code that encapsulates a semaphore

```
/* File: tmrtask.c */
static long int lSecondsToday;

void vTimerTask (void)
{
    (...)
    GetSemaphore (SEMAPHORE_TIME_OF_DAY);
    ++lSecondsToday;
    if (lSecondsToday == 60*60*24) lSecondsToday = 0L;
    GiveSemaphore (SEMAPHORE_TIME_OF_DAY);
    (...)
}

long lSecondsSinceMidnight (void)
{
    long lReturnValue;
    GetSemaphore (SEMAPHORE_TIME_OF_DAY);
    lReturnValue = lSecondsToday;
    GiveSemaphore (SEMAPHORE_TIME_OF_DAY);
    return (lReturnValue);
}
```

```
/* File: hacker.c */
long lSecondsSinceMidnight (void);

void vHackerTask (void)
{
    (...)
    lDeadline = lSecondsSinceMidnight () + 1800L;
    (...)
    if (lSecondsSinceMidnight () > 3600 * 12)
        (...)
}

/* File: junior.c */
long lSecondsSinceMidnight (void);

void vJuniorProgrammerTask (void)
{
    long lTemp;
    (...)
    lTemp = lSecondsSinceMidnight ();
    for (l = lTemp; l < lTemp + 10; ++l)
        (...)
}
```

Since `lSecondsSinceMidnight` uses the semaphore correctly, no bugs will be caused.

C Review: literal constants

```
/* File: tmrtask.c */
static long int lSecondsToday;

void vTimerTask (void)
{
    (...)
    GetSemaphore (SEMAPHORE_TIME_OF_DAY);
    ++lSecondsToday;
    if (lSecondsToday == 60*60*24) lSecondsToday = 0L;
    GiveSemaphore (SEMAPHORE_TIME_OF_DAY);
    (...)
}

long lSecondsSinceMidnight (void)
{
    long lReturnValue;
    GetSemaphore (SEMAPHORE_TIME_OF_DAY);
    lReturnValue = lSecondsToday;
    GiveSemaphore (SEMAPHORE_TIME_OF_DAY);
    return (lReturnValue);
}
```

```
/* File: hacker.c */
long lSecondsSinceMidnight (void);

void vHackerTask (void)
{
    (...)
    lDeadline = lSecondsSinceMidnight () + 1800L;
    (...)
    if (lSecondsSinceMidnight () > 3600 * 12)
    (...)
}

/* File: junior.c */
long lSecondsSinceMidnight (void);

void vJuniorProgrammerTask (void)
{
    long lTemp;
    (...)
    lTemp = lSecondsSinceMidnight ();
    for (l = lTemp; l < lTemp + 10; ++l)
    (...)
}
```

You specify the value of a type of constant at compile time, so you don't have to perform any time intensive (background) data-conversion during run-time.

Wretched alternative: invitation to semaphore and shared data bugs

```
/* File: tmrtask.c */
long int lSecondsToday;

void vTimerTask (void)
{
    (...)
    GetSemaphore (SEMAPHORE_TIME_OF_DAY);
    ++lSecondsToday;
    if (lSecondsToday == 60 * 60 * 24)
        lSecondsToday = 0L;
    GiveSemaphore (SEMAPHORE_TIME_OF_DAY);
    (...)
}
```

```
/* File: junior.c */
extern long int lSecondsToday;
void vJuniorProgrammerTask (void)
{
    (...)
    for (i=lSecondsToday; i<lSecondsToday+10; ++i)
        (...)
}

/* File: hacker.c */
extern long int lSecondsToday;
void vHackerTask (void)
{
    (...)
    iDeadline = lSecondsToday + 1800L;
    (...)
    if (lSecondsToday > 3600 * 12)
        (...)
}
```

C-Review: external variable

```
/* File: tmrtask.c */
long int lSecondsToday;

void vTimerTask (void)
{
    (...)
    GetSemaphore (SEMAPHORE_TIME_OF_DAY);
    ++lSecondsToday;
    if (lSecondsToday == 60 * 60 * 24)
        lSecondsToday = 0L;
    GiveSemaphore (SEMAPHORE_TIME_OF_DAY);
    (...)
}
```

```
/* File: iunior.c */
extern long int lSecondsToday;
void vJuniorProgrammerTask (void)
{
    (...)
    for (i=lSecondsToday; i<lSecondsToday+10; ++i)
        (...)
}

/* File: hacker.c */
extern long int lSecondsToday;
void vHackerTask (void)
{
    (...)
    iDeadline = lSecondsToday + 1800L;
    (...)
    if (lSecondsToday > 3600 * 12)
        (...)
}
```

Using the shared variable without semaphore protection.

External variable in C

- An external variable is a variable defined outside any function block.
- On the other hand, a local (automatic) variable is a variable defined inside a function block.
- The extern keyword means "declare without defining".

File 1:

```
int GlobalVariable;           // implicit definition
void SomeFunction();          // function prototype (declaration)

int main() {
    GlobalVariable = 1;
    SomeFunction();
    return 0;
}
```

File 2:

```
extern int GlobalVariable;     // explicit declaration

void SomeFunction() {          // function header (definition)
    ++GlobalVariable;
}
```

External variable in C

File 1:

```
int GlobalVariable;           // implicit definition
void SomeFunction();         // function prototype (declaration)

int main() {
    GlobalVariable = 1;
    SomeFunction();
    return 0;
}
```

File 2:

```
extern int GlobalVariable;    // explicit declaration

void SomeFunction() {        // function header (definition)
    ++GlobalVariable;
}
```

Remember the difference between definition and declaration.

- The variable GlobalVariable is **defined** in File 1. In order to utilize the same variable in File 2, it must be declared.
- Regardless of the number of files, a global variable is only defined once.
- If the program is in several source files, and a variable is defined in file 1 and used in file 2 and file 3, then extern **declarations** are needed in file 2 and file 3 to connect the occurrences of the variable.

Hard Real-Time Scheduling Considerations

Hard real-time scheduling considerations

- The obvious issue that arises in hard real-time systems is that you must somehow guarantee that the system will meet the hard deadlines.
- The ability to meet hard deadlines comes from writing fast code
- To write some frequently called subroutine in assembly language.
- If you can characterize your tasks, then the studies can help you determine if your system will meet its deadlines.

Saving Memory Space

- Embedded systems often have limited memory.
- RTOS: each task needs memory space for its stack.
- The first method for determining how much stack-space a task needs is to examine your code.
- The second method is experimental. Fill each stack with some recognizable data pattern at startup, run the system for a period of time

...A few ways to save code space

- Make sure that you aren't using two functions to do the same thing.
- Check that your development tools aren't sabotaging you.
- Configure your RTOS to contain only those functions that you need
- Look at the assembly language listings created by your cross-compiler to see if certain of your C statements translate into huge numbers of instructions.

Look at the assembly code...

```
struct sMyStruct a_sMyData[3];
struct sMyStruct *p_sMyData;
int i;
```

```
/* Method 1 for initializing data */
a_sMyData[0].iMember = 0;
a_sMyData[1].iMember = 5;
a_sMyData[2].iMember = 10;
```

```
/* Method 2 for initializing data */
for (i = 0; i < 3; ++i)
    a_sMyData[i].iMember = 5 * i;
```

```
/* Method 3 for initializing data */
i = 0;
p_sMyData = a_sMyData;
do
{
    p_sMyData->iMember = i;
    i += 5;
    ++p_sMyData;
} while (i < 10);
```

- Each of the methods does the same thing.
- Initializes the iMember of the a_sMyData array of structures.
- Which one contains the largest number of instructions?

Method #1: Reasonable amount of instructions

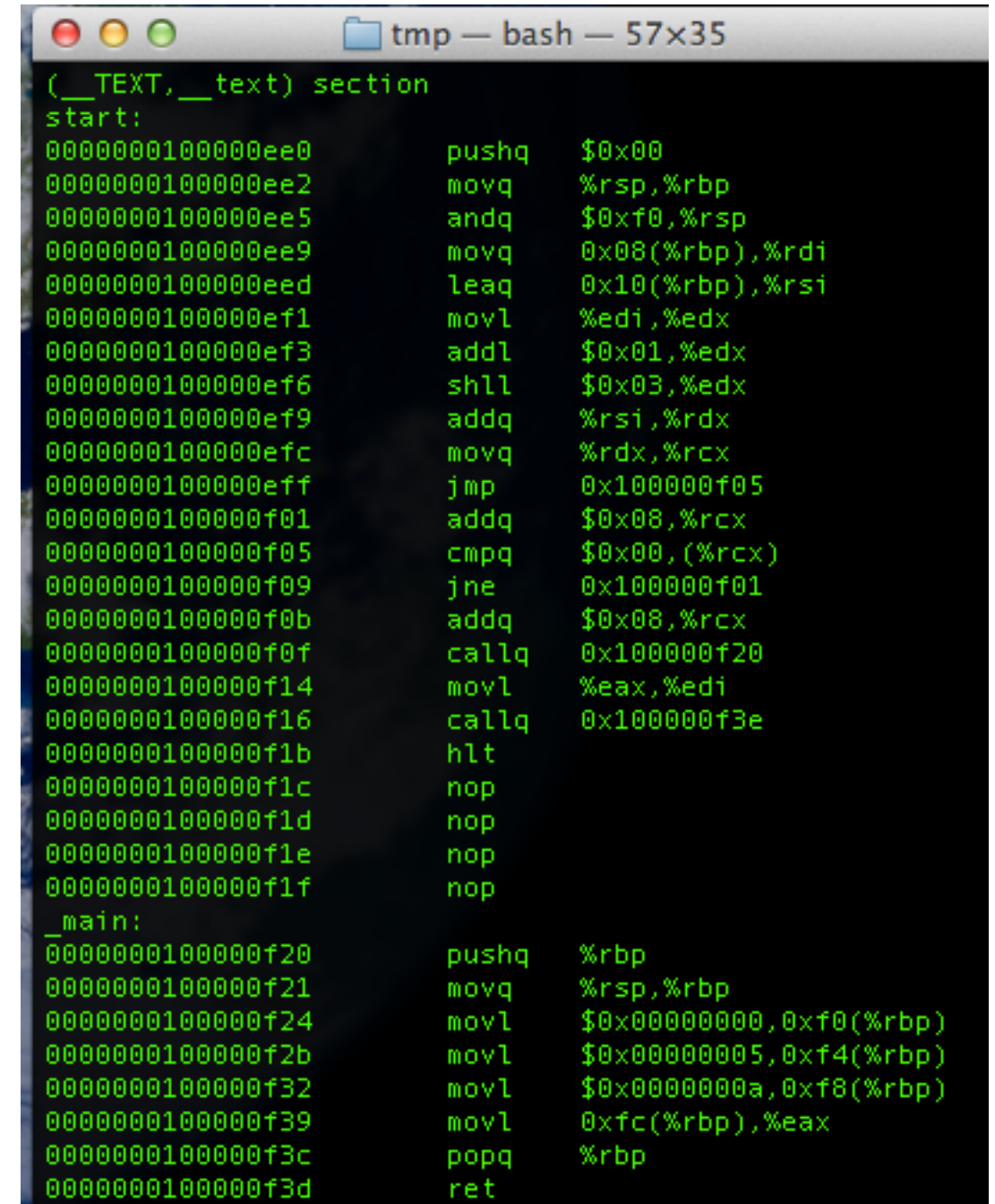
```
#include <stdio.h>

struct sMyStruct
{
    int iMember;
};

int main()
{
    struct sMyStruct a_sMyData[3];
    struct sMyStruct *p_sMyData;
    int i;
    a_sMyData[0].iMember = 0;
    a_sMyData[1].iMember = 5;
    a_sMyData[2].iMember = 10;
}
```

ASM dump with :

`gcc test.c ; otool -tv a.out`



```
(__TEXT,__text) section
start:
00000000100000ee0      pushq    $0x00
00000000100000ee2      movq     %rsp,%rbp
00000000100000ee5      andq     $0xf0,%rsp
00000000100000ee9      movq     0x08(%rbp),%rdi
00000000100000eed      leaq     0x10(%rbp),%rsi
00000000100000ef1      movl     %edi,%edx
00000000100000ef3      addl     $0x01,%edx
00000000100000ef6      shll     $0x03,%edx
00000000100000ef9      addq     %rsi,%rdx
00000000100000efc      movq     %rdx,%rcx
00000000100000eff      jmp      0x100000f05
00000000100000f01      addq     $0x08,%rcx
00000000100000f05      cmpq     $0x00,(%rcx)
00000000100000f09      jne      0x100000f01
00000000100000f0b      addq     $0x08,%rcx
00000000100000f0f      callq    0x100000f20
00000000100000f14      movl     %eax,%edi
00000000100000f16      callq    0x100000f3e
00000000100000f1b      hlt
00000000100000f1c      nop
00000000100000f1d      nop
00000000100000f1e      nop
00000000100000f1f      nop
_main:
00000000100000f20      pushq    %rbp
00000000100000f21      movq     %rsp,%rbp
00000000100000f24      movl     $0x00000000,0xf0(%rbp)
00000000100000f2b      movl     $0x00000005,0xf4(%rbp)
00000000100000f32      movl     $0x0000000a,0xf8(%rbp)
00000000100000f39      movl     0xfc(%rbp),%eax
00000000100000f3c      popq     %rbp
00000000100000f3d      ret
```

Method #3: Requires a lot more instructions!

```
#include <stdio.h>

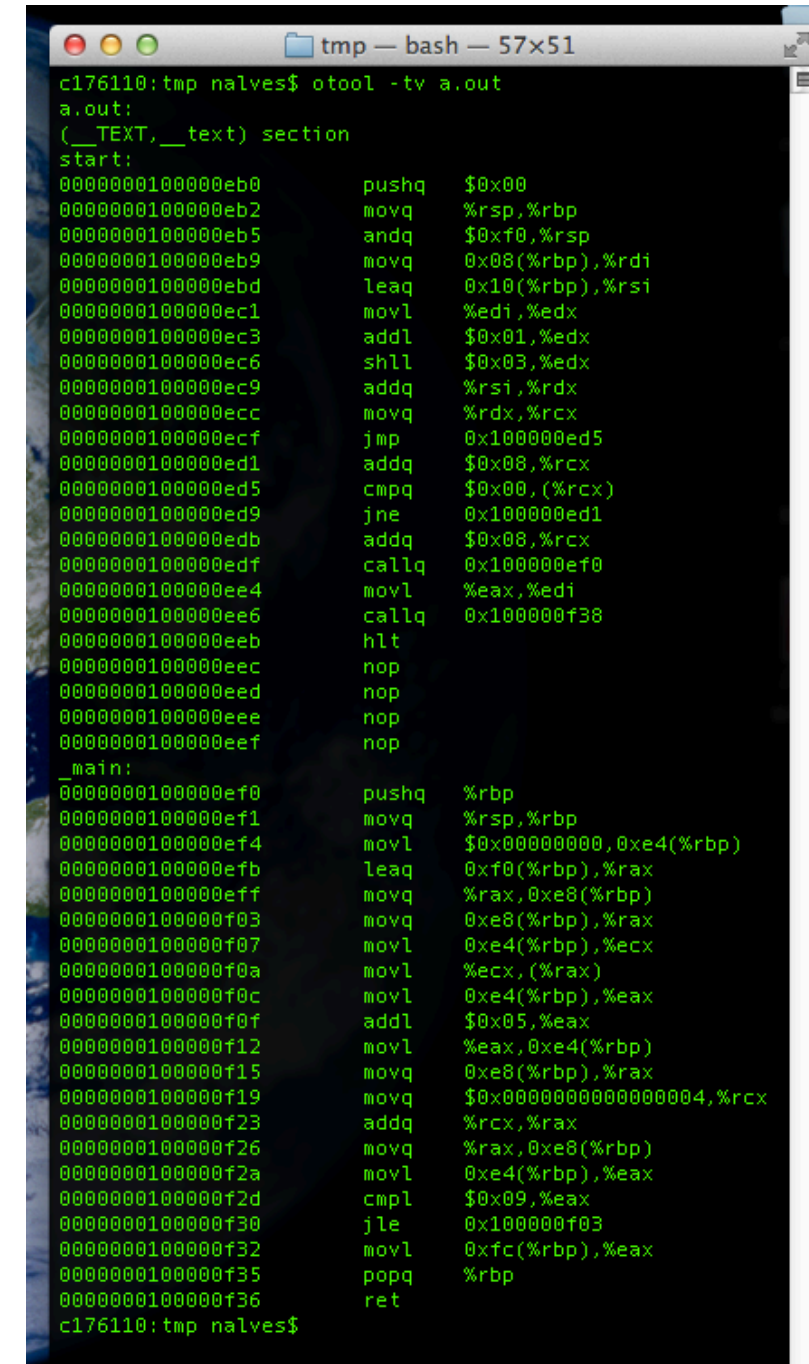
struct sMyStruct
{
    int iMember;
};

int main()
{
    struct sMyStruct a_sMyData[3];
    struct sMyStruct *p_sMyData;
    int i;

    i = 0;
    p_sMyData = a_sMyData;
    do
    {
        p_sMyData->iMember = i;
        i += 5;
        ++p_sMyData;
    } while (i < 10);
}
```

ASM dump with :

gcc test.c ; otool -tv a.out



```
c176110:tmp nalves$ otool -tv a.out
a.out:
(__TEXT,__text) section
start:
0000000010000eb0      pushq   $0x00
0000000010000eb2      movq    %rsp,%rbp
0000000010000eb5      andq    $0xf0,%rsp
0000000010000eb9      movq    0x08(%rbp),%rdi
0000000010000ebd      leaq    0x10(%rbp),%rsi
0000000010000ec1      movl    %edi,%edx
0000000010000ec3      addl    $0x01,%edx
0000000010000ec6      shll    $0x03,%edx
0000000010000ec9      addq    %rsi,%rdx
0000000010000ecc      movq    %rdx,%rcx
0000000010000ecf      jmp     0x10000ed5
0000000010000ed1      addq    $0x08,%rcx
0000000010000ed5      cmpq    $0x00,(%rcx)
0000000010000ed9      jne     0x10000ed1
0000000010000edb      addq    $0x08,%rcx
0000000010000edf      callq   0x10000ef0
0000000010000ee4      movl    %eax,%edi
0000000010000ee6      callq   0x10000f38
0000000010000eeb      hlt
0000000010000eec      nop
0000000010000eed      nop
0000000010000eee      nop
0000000010000eef      nop
_main:
0000000010000ef0      pushq   %rbp
0000000010000ef1      movq    %rsp,%rbp
0000000010000ef4      movl    $0x00000000,0x04(%rbp)
0000000010000efb      leaq    0xf0(%rbp),%rax
0000000010000eff      movq    %rax,0xe8(%rbp)
0000000010000f03      movq    0xe8(%rbp),%rax
0000000010000f07      movl    0xe4(%rbp),%ecx
0000000010000f0a      movl    %ecx,(%rax)
0000000010000f0c      movl    0xe4(%rbp),%eax
0000000010000f0f      addl    $0x05,%eax
0000000010000f12      movl    %eax,0xe4(%rbp)
0000000010000f15      movq    0xe8(%rbp),%rax
0000000010000f19      movq    $0x0000000000000004,%rcx
0000000010000f23      addq    %rcx,%rax
0000000010000f26      movq    %rax,0xe8(%rbp)
0000000010000f2a      movl    0xe4(%rbp),%eax
0000000010000f2d      cmpl    $0x09,%eax
0000000010000f30      jle     0x10000f03
0000000010000f32      movl    0xfc(%rbp),%eax
0000000010000f35      popq    %rbp
0000000010000f36      ret
c176110:tmp nalves$
```

Consider using static variables instead of variables on the stack

```
void vFixStructureCompact (struct sMyStruct *p_sMyData)
{
    static struct sMyStruct sLocalData;
    static int i, j, k;
    //Copy the struct in p_sMyData to sLocalData
    memcpy (&sLocalData, p_sMyData, sizeof sLocalData);

    !!Do all sorts of work in structure sLocalData, using
    !! i, j, and k as scratch variables.

    //Copy the data back to p_sMyData
    memcpy (p_sMyData, &sLocalData, sizeof sLocalData);
}
```

Consider using char variables instead of int variables

Every bit counts.

```
int i ;
struct sMyStruct sMyData[23];
(...)

for (i = 0; i < 23; ++i)
    sMyData[i].charStructMember = -1 * i;
```

```
char ch;
struct sMyStruct sMyData[23];
(...)
for (ch = 0; ch < 23; ++ch)
    sMyData[ch].charStructMember = -1 * ch;
```


Save space by going old school

- If all else fails, you can usually save a lot of space by writing your code in assembly language.
- Before doing this, try writing a few pieces of code in assembly to get a feel for how much space you might save (and how much work it will be to write and to maintain).

Saving power

- Some embedded systems run on battery power, and for these systems, battery life is often a big issue.
- The primary method for preserving battery power is to turn off parts or all of the system whenever possible.
- Most embedded-system microprocessors have at least one power-saving mode; many have several.
- The modes have names such as sleep mode, low-power mode, idle mode, standby mode, and so on.
- A very common power-saving mode is one in which the microprocessor stops executing instructions, stops any built-in peripherals, and stops its clock circuit.

Typical power saving modes

- Another typical power-saving mode is one in which the microprocessor stops executing instructions but the on-board peripherals continue to operate.
- Any interrupt starts the microprocessor up again.
- No special hardware is required
- Use this power-saving mode even while other things are going on.
- For example, a built-in DMA channel can continue to send data to a UART, the timers will continue to run,