Real-Time Operating Systems (Part III)

Embedded Software Design

熊博安

國立中正大學資訊工程研究所

pahsiung@cs.ccu.edu.tw

Textbook: Embedded Software Primer, David E. Simon, Addison Wesley

Contents

- Principles of Embedded Software Design
  - How many tasks?
  - Task Structure
- Tank Monitoring System Example
- Encapsulating Semaphores & Queues
- Saving Memory Space
- Saving Power

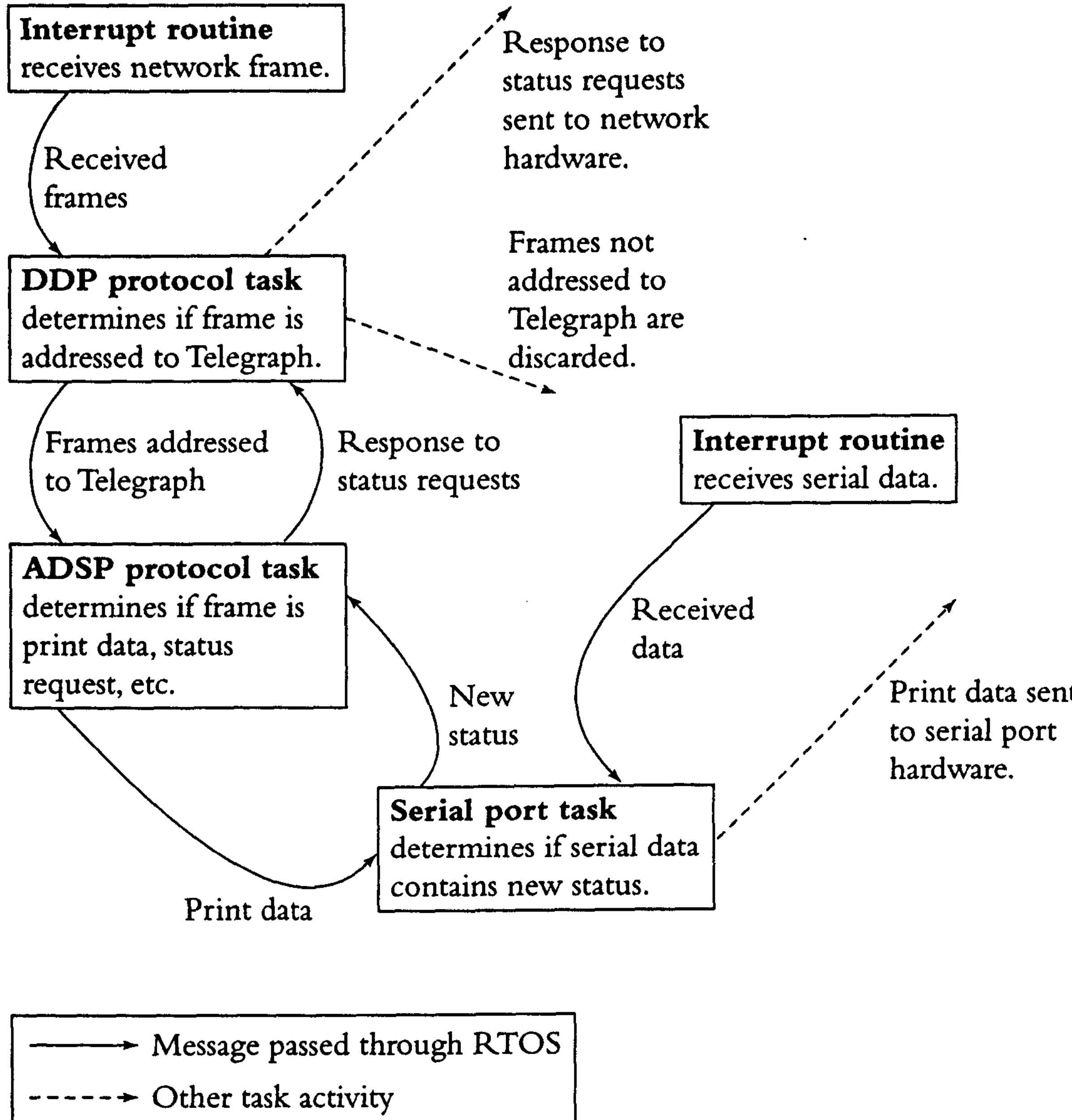
Overview

- Specification of a system is as difficult as designing it (preciseness, timing, …)
- Cordless bar-code scanner
  - respond on time 99% of the cases
  - slightly too slow 1% of the cases
  - SOFT real-time system
- Nuclear reactor system
  - absolute deadlines (100% satisfaction)
  - HARD real-time system

Principles

- Embedded systems start doing something only if:
  - time has elapsed (timer expired!), OR
  - external event arrived (interrupts)
- RTOS tasks are blocked most of the time waiting for some event
- Interrupt causes a cascade of signals and task activities (chain reaction!)
- Example: Telegraph Operation

Telegraph Operation



Write Short ISRs

- Why? Because:
  - lowest priority ISR is executed in preference to highest priority task code → slower task code response
  - ISR more bug-prone and harder to debug
- When an interrupt occurs, there may be several things to do (reset port, save data, reset controller, analyze data, formulate response, etc.)
- Distinguish between urgent & non-urgent tasks!
- Perform the urgent ones in ISR!
- Signal a task to do the non-urgent ones!

Example: how to write ISR

System Requirements

- System must respond to commands from serial port
- Commands end with carriage return
- Commands arrive one at a time; next command will arrive only after current command is responded to
- Serial port hardware stores 1 character at a time, and characters arrive quickly
- System can respond to commands slowly

Example: how to write ISR

Possible Solution Designs

- Everything in 1 ISR ➔ long, complex, difficult to debug, slow response for all operations
- Brainless ISR (forward each character to command parsing task) ➔ short, lots of messages for transmitting, chars arrive quickly → ISR not able to keep up
- Compromise ➔ save chars in buffer, lookout for carriage return, send single msg to task

Keeping ISR short (in VRTX)

Figure 8.2 Keeping Interrupt Routines Short

```
#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 received 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.
}
```

Keeping ISR short (in VRTX)

```
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.
- Am I better off with more tasks or with fewer tasks?

How Many Tasks?

Advantages of Many Tasks

- Better control over response times
  - better response for higher-priority tasks
- More modular
  - 1 task for 1 device
- Encapsulate data more effectively
  - e.g.: network connection handled separately

How Many Tasks?

Disadvantages of Many Tasks

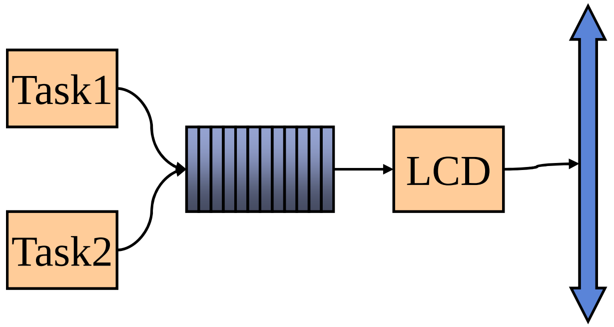
- More data sharing among tasks (more bugs, more semaphores, more time lost)
- More communication (more message queues, mailboxes, pipes, more memory, more time lost, more bugs)
- Each task requires a stack (more memory)
- More task switching (more time lost, less throughput)
- More calls to RTOS (more time lost, less throughput)

Timings of an RTOS on 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

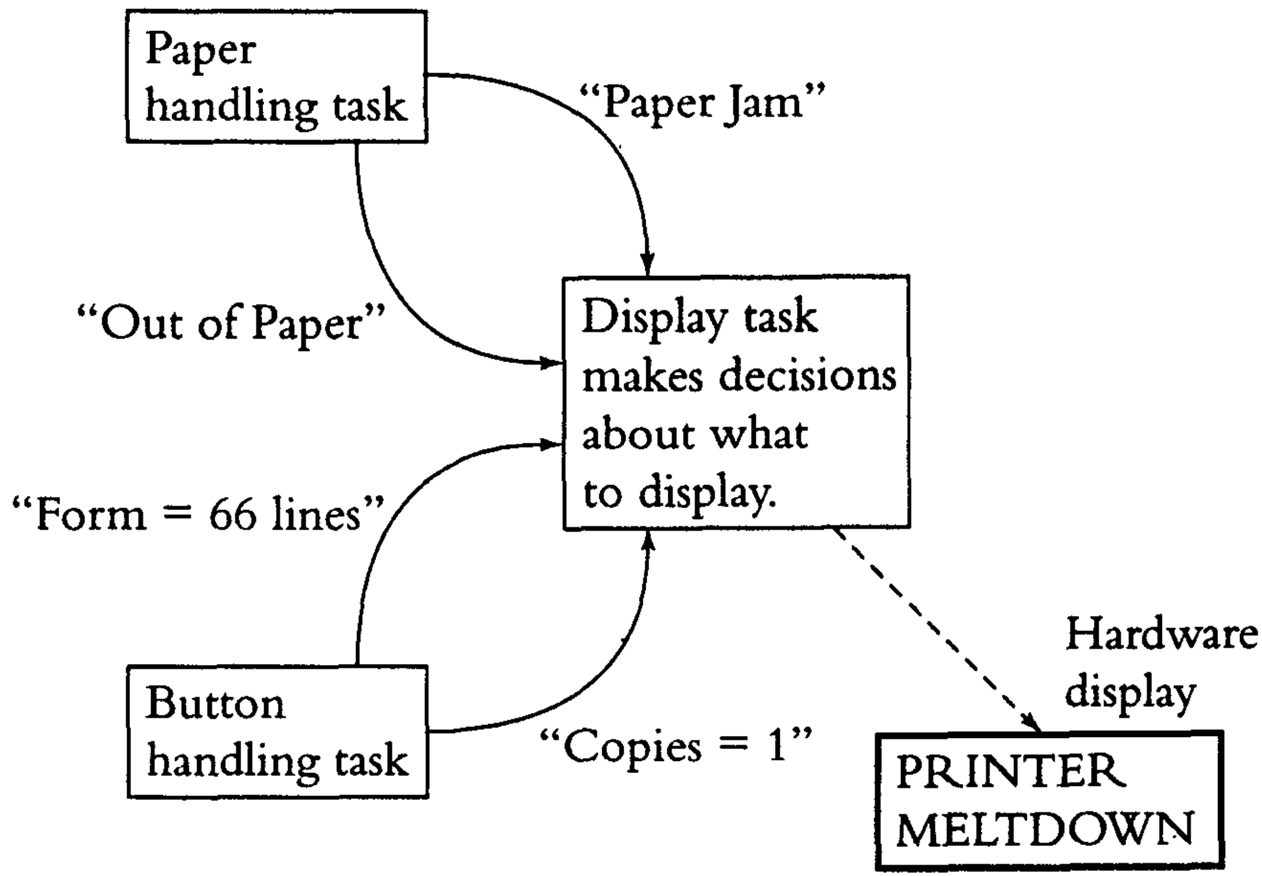
When Do You Need Tasks?

- You need tasks for **priority**
  - better control over response times
  - E.g.: response to user button presses
- You need tasks for **encapsulation**
  - to deal with shared hardware / software
  - E.g.: LCD display on printer
    - user button press
    - printing task (error reporting)
    - “TONER JAM ON LINE NOW”



Task to Control Shared Hardware

- Laser Printer



A Separate Task to Handle Shared Flash Memory Hardware

```
typedef enum
{
   FLASH_READ,
   FLASH_WRITE
} FLASH_OP;
```

```
#define SECTOR_SIZE   256

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

Shared Flash Memory Hardware

```
void vHandleFlashTask (void)
{
   mdt_q sQueueOurs;          /* Handle of our input queue */
   FLASH_MSG sFlashMsg;       /* Message telling us what to do. */
   int iMsgPriority;          /* Priority of received message */

   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)
      {
```

(continued)

Shared Flash Memory Hardware

```
            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;
         }
      }
}
```

Shared Flash Memory Hardware

```
void vTaskA (void)
{
   mdt_q sQueueFlash;       /* Handle of flash task input queue */
   FLASH_MSG sFlashMsg;     /* Message to the flash routine. */
   .
   .
   .
   /* 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 the queue and send the message with priority 5 */
   sQueueFlash = mq_open ("FLASH", O_WRONLY, 0, NULL);
   mq_send (sQueueFlash,
      (void *) &sFlashMsg, sizeof sFlashMsg, 5);
   mq_close (sQueueFlash);
   .
   .
   .
}
```

Shared Flash Memory Hardware

```
void vTaskB (void)
{
   mdt_q sQueueOurs;        /* Handle of our input queue */
   mdt_q sQueueFlash;       /* Handle of the flash input queue */
   FLASH_MSG sFlashMsg;     /* Message to the flash routine. */
   int iMsgPriority;        /* Priority of received message */
   .
   .
   .
   /* Create a queue called 'TASKB' for input to this
      task */
   sQueueOurs = mq_open ("TASKB", O_CREAT, 0, NULL);
   .
   .
   .
   /* We need to read data from the flash */
   /* Set up the data in the message structure */
   sFlashMsg.iSector = FLASH_SECTOR_FOR_TASK_B;
   sFlashMsg.eFlashOp = FLASH_READ;

   /* Open the queue and send the message with priority 5 */
   sQueueFlash = mq_open ("FLASH", O_WRONLY, 0, NULL);
   mq_send (sQueueFlash,
      (void *) &sFlashMsg, sizeof sFlashMsg, 5);
   mq_close (sQueueFlash);
   .
   .
   .
   /* Wait for the flash task's response on our queue. */
   mq_receive (sQueueOurs, (void *) &sFlashMsg,
      sizeof sFlashMsg, &iMsgPriority);

   !! Use the data in sFlashMsg.a_byData
   .
   .
   .
}
```

Task to Control Shared Software

- Example: Error log
- Log is handled by a separate task
- Centralize:
  - all writes of new errors into the log
  - flushing old data out of log when full
  - culling duplicates out of log, if necessary

Common wrong suggestions

- Have many small tasks, so that each is simple
  - share a lot of data, semaphores, inter-task communications, task switching time, etc.
- Have separate tasks for work that needs to be done in response to separate stimuli
  - use tasks for prioritization and encapsulation instead of stimuli-based

Tasks for Separate Stimuli

```
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

```
vtaska.c

!! 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;
         .
         .
         .
      }
   }
}
```

Recommended Task Structure

- Task should block in only one place
  - too many blockings will be difficult to debug
  - When another task puts a request on the task's queue, this task is not off waiting for some other event that may or may not happen in a timely fashion.
- Nothing to do ➔ input queue empty ➔ task will block ➔ use no CPU time
- No public data to share
  - other tasks must make requests to read/write private data
  - no need of semaphores

Avoid Creating & Destroying Tasks

- Create all tasks at system start
- Avoid creating & destroying tasks dynamically
  - time-consuming functions
  - creating a task = reliable operation, but destroying a task = leaves little pieces lying around to cause bugs!
    - E.g.: Semaphore-owning task destroyed, other tasks need semaphore blocked forever
    - Some RTOS takes care automatically, but:
    - message in task's queue? destroy queue, delete message? what if it has a pointer to memory to be freed later? ➔ memory leak!

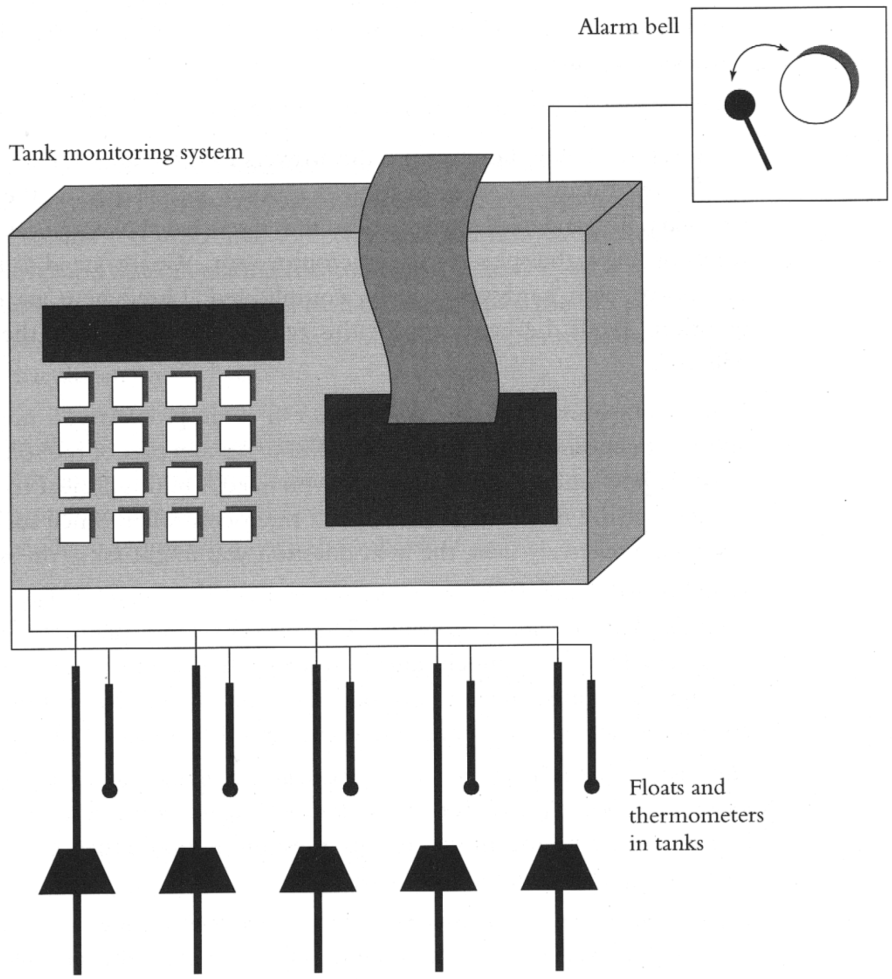
Turn off Time-Slicing

- Same priority → may use time slicing
- Time-slicing → fair use of resources → good for interactive users → not for embedded systems!
- Cuts throughput (more task switches)
- Example: 6 tanks, 6 computations of gasoline amounts (each 5 seconds)
  - one after another → better throughput
  - all 6 results after 30 seconds → bad throughput

Restrict Use of RTOS

- Configure RTOS
  - remove from RTOS whatever is not necessary for your applications
  - save memory space and time
  - Example: 7 pipes, 1 queue ➔
    - replace queue with an 8th pipe, no need of queue code in RTOS, OR
    - replace all 7 pipes with 7 queues, if queues are smaller in size than pipes!
  - Use fixed size messages in a pipe (opcode, error code, pointer) ➔ less bugs, predictable time
  - Use a shell, all code access RTOS through shell ➔ RTOS-independent, more portable!!!

Underground Tank Monitoring System



Tank Monitoring System Requirements

- 8 underground tanks
- Read thermometers and floats in each tank
- Calculate #gallons of gasoline in a tank using temperature and float readings
- Monitor tank level periodically,
  - indicate leak
  - warn possible overflow
- 16-button keypad, 20-char LCD, thermal printer

Initial Questions

- How often to read floats?
  - Several times per second
- How quickly to respond to user button push?
  - In no more than 0.1 second
- How fast does printer print?
  - 2 or 3 lines per second
- What microprocessor to use?
  - 8-bit microcontroller (no profit from this system)

Initial Questions

- How long is the gasoline amount calculation?
  - 4 or 5 seconds (not definite, depends on CPU)
- How long to recognize leak/overflow?
  - some hundredths of a sec (need experiments)
- Read level from more than 1 tank at once?
  - No, one after another
- How difficult to turn alarm bell on & off?
  - Simple! Just writing 1 or 0 to a bit

Resolving a Timing Problem

- Check each tank several times per second
- 4 or 5 seconds to calculate gas quantity!
- Impossible to build!
  - Use 20 times faster CPU? **No!**
  - Detect overflow from raw float levels? **Yes!**
  - Use RTOS? **Yes!**
- Calculation: processor hog
- Button response: need interrupts

Dividing the Work into Tasks

- A level calculation task
- An overflow detection task
- A float hardware task
- A button handling task
- A display task
- An alarm bell task
- A print formatting task

Dividing Work into Tasks

- Level Calculation Task
  - takes as input float levels, temperatures
  - calculates gasoline in tank
  - detects leaks by looking at previous readings
  - 4 or 5 seconds ➔ processor hog
  - separate, low-priority task in RTOS
  - How many tasks?
    - one task per tank? (one float level reading at once! ➔ communication between tasks!)
    - only one task? Yes!

Dividing Work into Tasks

- Overflow Detection Task
  - read float levels
  - fast, high-priority
  - separate task
- Floats are read by both level calculation task and overflow detection task
  - separate float hardware task? OR
  - use a semaphore?
    - waiting at most 1 or 2 ms on semaphore!
    - yes, all the tasks can wait that long! No problem!

Dividing Work into Tasks

- Button Handling Task
  - need state machine to track buttons pressed by user
  - need interrupt routine
- Shared LCD ➔ Display Task?
  - use semaphore? OR
    - Suppose “Leak!!” message on LCD, user presses button before he can read the leak message, message is gone!
  - separate display task? Yes!

Semaphore Cannot Protect LCD

```
void vLevelCalculationTask (void)
{
   .
   .
   .
   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);
   }
   .
   .
   .
}
```

Dividing Work into Tasks

- Alarm bell
  - separate task? No!
  - direct control by other tasks? Yes!
    - bell is never “in the middle of something”!
    - user turns bell off? intentionally!
- Print Formatting Task
  - report formatting and printing is slower than button responses required (0.1 sec)
  - several reports in queue to format

Dealing with the Shared Data

- The level data is shared by several tasks
  - The level calculation task
  - The display task
  - The print formatting task
- A semaphore or another task?
  - What is the longest that any one task will hold on to the semaphore?
  - Can every other task wait that long?

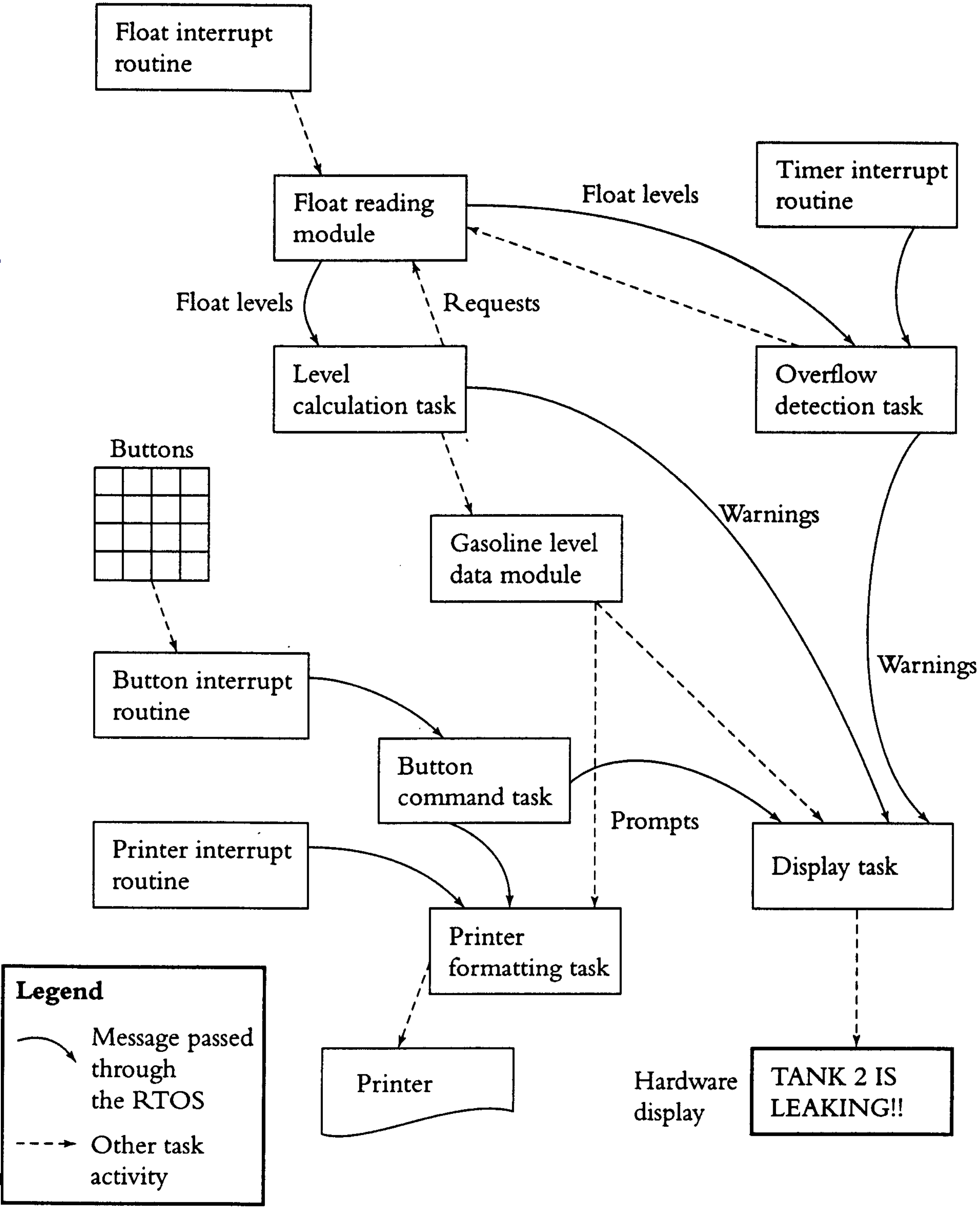
Moving System Forward

- Button Press → button hardware interrupts CPU → button ISR sends message to button handling task → interpret commands → forward messages to display task, printer task
- Timer to read floats at a specific rate, check for possible overflow
- Float reading set up by tasks, floats read → hardware interrupts
- Print formatting task sends 1st line to printer, then printer interrupts after finishing each line

Tasks in 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	Medium	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.

Tank Monitoring Design



Encapsulating Semaphores & Queues

- Encapsulate semaphores and queues into functions
  - no direct access of semaphores or queues
  - fewer bugs
  - more modular

Encapsulating 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);
   .
   .
   .
}
```

(continued)

Encapsulating Semaphore

Figure 8.10 *(continued)*

```
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)
   .
   .
   .
}
```

Encapsulating Semaphore

```
/* File: junior.c */

long lSecondsSinceMidnight (void);

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

Wretched Alternative

Figure 8.11 The Wretched Alternative

```
/* File: tmrtask.c */

/* global */ 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: hacker.c */

extern long int lSecondsToday;

void vHackerTask (void)
{
   .
   .
   .
   /* (Hope he remembers to use the semaphore) */
   lDeadline = lSecondsToday + 1800L;
   .
   .
   .
```


(continued)

Wretched Alternative

Figure 8.11 *(continued)*

```
    /* (Here, too) */
    if (lSecondsToday > 3600 * 12)
    .
    .
    .
}

-----------------------------

/* File: junior.c */

extern long int lSecondsToday;

void vJuniorProgrammerTask (void)
{
    .
    .
    .
    /* (Hope junior remembers to use the semaphore here, too) */
    for (l = lSecondsToday; l < lSecondsToday + 10; ++l)
    .
    .
    .
}
```

Encapsulating Semaphore (float)

Figure 8.12 Another Semaphore Encapsulation Example

```
/* floats.c */

typedef void (*V_FLOAT_CALLBACK) (int iFloatLevel);

static V_FLOAT_CALLBACK vFloatCallback = NULL;

SEMAPHORE SEM_FLOAT;

void interrupt vFloatISR (void)
{
   int iFloatLevel;

   V_FLOAT_CALLBACK vFloatCallbackLocal;

   iFloatLevel = !! Read the value of the float;

   vFloatCallbackLocal = vFloatCallback;
   vFloatCallback = NULL;
   ReleaseSemaphore (SEM_FLOAT);

   vFloatCallbackLocal (iFloatLevel);
}

void vReadFloats (int iTankNumber, V_FLOAT_CALLBACK vCb)
{
   TakeSemaphore (SEM_FLOAT);

   /* Set up the callback function */
   vFloatCallback = vCb;

   !! Set up the hardware to read from iTankNumber
}
```

Encapsulating Message Queue

Figure 8.13 Encapsulating a Message Queue

```
/* File: flash.h */

#define SECTOR_SIZE   256
typedef void (*V_RD_CALLBACK) (BYTE *p_byData);
void vWriteFlash (int iSector, BYTE *p_byData);
void vReadFlash (int iSector, V_RD_CALLBACK vRdCb);

-----------------------------

/* File: flash.c */

typedef enum
{
   FLASH_READ,
   FLASH_WRITE
} FLASH_OP;
```

(continued)

Encapsulating Message Queue

Figure 8.13 *(continued)*

```
typedef struct
{
   FLASH_OP eFlashOp;          /* FLASH_READ or FLASH_WRITE */
   V_RD_CALLBACK *vRdCb;       /* Function to callback on read. */
   int iSector;                /* Sector of data */
   BYTE a_byData[SECTOR_SIZE];
                               /* Data in sector */
} FLASH_MSG;

#include "flash.h"

static mdt_q sQueueFlash;     /* Handle of our input queue */

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 */
   sQueueFlash = mq_open ("FLASH", O_CREAT, 0, NULL);
}
```

Encapsulating Message Queue

```
void vWriteFlash (int iSector, BYTE *p_byData)
{
   FLASH_MSG sFlashMsg;

   sFlashMsg.eFlashOp = FLASH_WRITE;
   sFlashMsg.vRdCb = NULL;
   sFlashMsg.iSector = iSector;
   memcpy (sFlashMsg.a_byData, p_byData, SECTOR_SIZE);
   mq_send (sQueueFlash,
      (void *) &sFlashMsg, sizeof sFlashMsg, 5);
}

void vReadFlash (int iSector, V_RD_CALLBACK *vRdCb)
{
   FLASH_MSG sFlashMsg;

   sFlashMsg.eFlashOp = FLASH_READ;
   sFlashMsg.vRdCb = vRdCb;
   sFlashMsg.iSector = iSector;
   mq_send (sQueueFlash,
      (void *) &sFlashMsg, sizeof sFlashMsg, 6);
}
```

Encapsulating Message Queue

```
void vHandleFlashTask (void)
{
   FLASH_MSG sFlashMsg;    /* Message telling us what to do. */
   int iMsgPriority;       /* Priority of received message */

   while (TRUE)
   {
      /* Get the next request. */
      mq_receive (sQueueFlash, (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 to the task that
               sent the message to us. */
            sFlashMsg.vRdCb (sFlashMsg.a_byData);
            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;
      }
   }
}
```


(continued)

Encapsulating Message Queue

```
/* File: taska.c */

#include "flash.h"

void vTaskA (void)
{
   BYTE a_byData[SECTOR_SIZE];   /* Place for flash data */
   :
   :
   /* We need to write data to the flash */

   vWriteFlash (FLASH_SECTOR_FOR_TASK_A, a_byData);
   :
   :
}

-----------------------------

/* File: taskb.c */

#include "flash.h"

void vTaskBFlashReadCallback (BYTE *p_byData)
{
   !! Copy the data into local variables.
   !! Signal vTaskB that the data is ready.
}

void vTaskB (void)
{
   :
   :
   /* We need to read data from the flash */
   vReadFlash (FLASH_SECTOR_FOR_TASK_B, vTaskBFlashReadCallback);
   :
   :
}
```

Saving Memory Space

- Memory space is very limited in embedded systems (not like desktop systems with GBs!)
- Program size must fit in ROM
- Data size must fit in RAM
- They are not interchangeable!
- Squeezing data into efficient structures → savings in data size, BUT
- More code needed to read those data → extra space needed in program size!!!

Saving Memory Space

- How to determine stack space size?
- Analysis:
  - Each function call, parameter, local variable
  - Deepest combination of function nesting, parameters, and local variables
  - Worst-case nesting of interrupt routines
  - RTOS itself (in manual)
- Experiment:
  - run and measure (not necessarily worst-case!)

Saving Memory Space

- Don't use 2 functions to do same thing
  - 28 memcpy, 1 memmove ➔
    - change memmove to memcpy OR
    - change all memcpy to memmove
- Check development tools which might drag unnecessary codes into your application
  - drags memmove, memset, memcmp, strcpy, strncpy along with memcpy
- Configure RTOS to include only what you need
- Check assembly-language listings created by compiler
  - different ways of doing same thing in C give different amount of assembly code

Saving Memory Space

```
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 (i = 0; i < 3; ++i)
   a_sMyData[i].iMember = 5 * i;
```

Saving Memory Space

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

Saving Memory Space

Use static variables instead of variables on stack (parameters, local variables)

```
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);
}
```

```
void vFixStructureLarge (struct sMyStruct *p_sMyData)
{
    int i, j, k;

    !! Do all sorts of work in structure pointed to by
    !! p_sMyData, using i, j, and k as scratch variables.
}
```

Saving Memory Space

- On an 8-bit processor, use char instead of int

```
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;
```

Saving Memory Space

- If all else fails, write in assembly language!

Saving Power

- Microprocessor has one or more power-saving modes (sleep, low-power, standby)
- Software can put microprocessor into one of those modes
  - by a special instruction, or
  - by writing into a control register

Common Power-Saving Mode (1)

- stop executing instructions, stop peripherals, stop clock circuit ➔ saves lot of power
- requires restarting software because microprocessor is reset
- software must figure out if it just started or is restarting
  - by writing value 0x9283ab3c into location 0x0100
  - check location on starting
- Static RAM uses little power ➔ no need of stopping

Common Power-Saving Mode (2)

- stop executing instructions, peripherals continue to operate ➔ saves less power
- no special hardware required
- no need of restarting software
- DMA continues to send data to UART
- Timers continue to run, interrupt microprocessor, etc.

Common Power-Saving Mode (3)

- Turn off entire system → power consumption = 0
- User turns it back on when needed
- Example
  - Cordless bar-code scanner
  - User pulls trigger to initiate scan
  - Trigger-pull turns entire system back on
  - Software needs to turn system off
  - Software needs to save to EEROM

What to turn off?

- Parts that have lots of signals that change frequently from high to low and back use the most power
- Turn those parts off!
- Lookup data sheets to find such parts and if it is worthwhile to turn them off