Message queues, mailboxes and pipes

Reference: Simon Chapter 7

Nuno Alves (nalves@wne.edu), College of Engineering

Other common features offered by commercial RTOS

There are other features commonly offered by commercial RTOSs:

- Intertask communication
- Timer services
- Memory management
- Events
- Interaction between interrupt routines and RTOSs.

Message queues, mailboxes, and pipes

Tasks must be able to communicate with one another to coordinate their activities or to share data.

For example the tank monitoring system.

- Tasks can use shared data and semaphores to allow task-communication.
- There are several other methods that most RTOSs offer: queues, mailboxes, and pipes.

Simplified queue example

Two high priority tasks

- Suppose we have **Task1** and **Task2**, each has a number of high-priority, urgent things to do.
- Suppose these two tasks discover error conditions that must be reported on a network, a time-consuming process.
- In order not to delay Task1 and Task2, it makes sense to have a separate task, **ErrorsTask**, that is responsible for reporting the error conditions.
- Whenever Task1 or Task2 discovers an error, it reports error to ErrorsTask and goes on about its own business.
- The error reporting process undertaken by ErrorsTask does not delay the other tasks.

Simple use of a queue

```
//RTOS queue function prototypes
void AddToQueue  (int iData);
void ReadFromQueue  (int *p,iData);

//global variables
static int cErrors;

void Task1 (void)
{
  (...)
  if  (!! problem arises)
    vLogError (ERROR_TYPE_X);
  !! Do other things
  (...)
}

void Task2 (void)
  (...)
  if  (!! problem arises)
    vLogError (ERROR_TYPE_X);
  !! Do other things
  (...)
}
```

```
void  vLogError  (int  iErrorType)
{
  AddToQueue  (iErrorType);
}

//This task is running on background
void  ErrorsTask  (void)
{
  int  iErrorType;
  while   (FOREVER)
  {
    ReadFromQueue (&iErrorType);
    ++cErrors;

    !!Send cErrors out on network
    !!Send iErrorType out on network
  }
}
```

Simple use of a queue

```
//RTOS queue function prototypes
void AddToQueue  (int iData);
void ReadFromQueue  (int *p,iData);

//global variables
static int cErrors;

void Task1 (void)
{
  (...)
  if  (!! problem arises)
    vLogError (ERROR_TYPE_X);
  !! Do other things
  (...)
}

void Task2 (void)
  (...)
  if  (!! problem arises)
    vLogError (ERROR_TYPE_X);
  !! Do other things
  (...)
}
```

Task1 and **Task2**, each has a number of high-priority, urgent things to do.

```
{
  int  iErrorType;
  while   (FOREVER)
  {
    ReadFromQueue (&iErrorType);
  }
}
```

When Task1 or Task2 needs to log errors, it calls **vLogError**.

Simple use of a queue

The **vLogError** function puts the error on a queue of errors for ErrorsTask to deal with.

AddToQueue function adds the value of the integer parameter it is passed to a queue of integer values the RTOS maintains internally.

```
void  vLogError  (int  iErrorType)
{
  AddToQueue  (iErrorType);
}

//This task is running on background
void  ErrorsTask  (void)
{
  int  iErrorType;
  while   (FOREVER)
  {
    ReadFromQueue (&iErrorType);
    ++cErrors;

    !!Send cErrors out on network
    !!Send iErrorType out on network
  }
}
```

Simple use of a queue

ReadFromQueue function reads the value at the head of the queue and returns it to the caller.

If the queue is empty, **ReadFromQueue** blocks the calling task.

The RTOS guarantees that both of these functions are reentrant.

```
void  vLogError  (int  iErrorType)
{
  AddToQueue  (iErrorType);
}

//This task is running on background
void  ErrorsTask  (void)
{
  int  iErrorType;
  while   (FOREVER)
  {
    ReadFromQueue (&iErrorType);
    ++cErrors;

    !!Send cErrors out on network
    !!Send iErrorType out on network
  }
}
```

Some ugly details

Some ugly details

- Most RTOSs require that you **initialize your queues** before you use them, by calling a function provided for this purpose.

- You can have **as many queues as you want**, you pass an additional parameter to every queue function: the identity of the queue.

- If your code tries to write to a queue **when the queue is full**, the RTOS either returns an error or it blocks the task until some other task reads data from the queue and thereby creates some space

More ugly details

- Many RTOSs include a function that will read from a queue if there is any data and will return an error code if not.
- The amount of data that the RTOS lets you write to the queue in one call may not be exactly the amount that you want to write.

Complete queue example

Complete queue example

```
//RTOS queue function prototypes
OS_EVENT *OSQCreate  (void **ppStart, BYTE bySize);
unsigned char OSQPost (OS_EVENT *pOse,  void *pvMsg);
void *OSQPend(OS_EVENT *pOse, WORD wTimeout, BYTE *pByErr);

#define  WAIT__FOREVER 0
//Our message queue
static OS_EVENT *pOseQueue;
//The data space for our queue (managed by the RTOS)
#define SIZEOF_QUEUE 25
void *apvQueue[SIZEOF_QUEUE];

void main  (void)
{
  (...)
  //Queue is initialized before the tasks
  pOseQueue = OSQCreate (apvQueue, SIZEOF_QUEUE);
  (...)
  !! Start Task1 then Start Task2
  (...)
}

void Task1 (void)
{
  (...)
  if  (!! problem arises) vLogError (ERROR_TYPE_X);
  !! Do other things
  (...)
}

void Task2 (void)
  (...)
  if  (!! problem arises) vLogError (ERROR_TYPE_X);
  !! Do other things
  (...)
}
```

```
void vLogError (int iErrorType)
{
  //Return code from writing to queue
  BYTE byReturn;
  //Write to the queue.
  //Cast the error type as a void pointer
  byReturn=OSQPost(pOseQueue,(void *) iErrorType);
  if (byReturn != OS_N0_ERR)
  !! Handle the situation when queue is full
}

static int cErrors;
void ErrorsTask (void)
{
  int iErrorType;
  BYTE byErr;
  while (FOREVER)
  {
    //Cast the value received from the queue
    //back to an int. Since that there is no
    //possible error from this, so we ignore byErr.
    iErrorType=
(int)OSQPend(pOseQueue,WAIT_FOREVER,&byErr );
    ++cErrors;
    !! Send cErrors and iErrorType out on network
   }
}
```

Pointers and queues

Nuno Alves (nalves@wne.edu), College of Engineering

Pointers and queues example

```
//Queue  function  prototypes
OS_EVENT *OSQCreate(void **ppStart, BYTE bySize);
Unsigned char OSQPost(0S_EVENT *pOse,void  *pvMsg);
void *OSQPend(0S_EVENT *pOse,WORD wTimeout,BYTE *pByErr);
#define WAIT_FOREVER 0
Static 0S_EVENT *pOseQueueTemp;

void  vReadTemperaturesTask  (void)
{
  int *pTemperatures;
  while (TRUE)
  {
    !!Wait until the time to read the next temperature
    //Get a new buffer for the new set of temperatures.
    pTemperatures=(int *) malloc (2*sizeof*pTemperatures);

    pTemperatures[0] = !!read in value from hardware;
    pTemperatures[1] = !!Read in value from hardware;

    //Add a pointer to the new temperatures to the queue
    OSQPost (pOseQueueTemp, (void *) pTemperatures);
  }
}
void vMainTask (void)
{
  int *pTemperatures;
  BYTE byErr;
  while (TRUE)
  {
    pTemperatures = (int *) OSQPend (pOseQueueTemp, WAIT_FOREVER, &byErr);
    if (pTemperatures[0] != pTernperatures[1])
    !! Set  off howling alarm;
    free (pTemperatures);
  }
}
```

- A void pointer (pointer that points to a raw memory location) is written to the queue on each call
- One task can pass any amount of data to another task by putting the data into a buffer and then writing a pointer to the buffer onto the queue.

Pointers and queues example

```
//Queue  function  prototypes
OS_EVENT *OSQCreate(void **ppStart, BYTE bySize);
Unsigned char OSQPost(0S_EVENT *pOse,void  *pvMsg);
void *OSQPend(0S_EVENT *pOse,WORD wTimeout,BYTE *pByErr);
#define WAIT_FOREVER 0
Static 0S_EVENT *pOseQueueTemp;

void  vReadTemperaturesTask  (void)
{
  int *pTemperatures;
  while (TRUE)
  {
    !!Wait until the time to read the next temperature
    //Get a new buffer for the new set of temperatures.
    pTemperatures=(int *) malloc (2*sizeof*pTemperatures);

    pTemperatures[0] = !!read in value from hardware;
    pTemperatures[1] = !!Read in value from hardware;

    //Add a pointer to the new temperatures to the queue
    OSQPost (pOseQueueTemp, (void *) pTemperatures);
  }
}
void vMainTask (void)
{
  int *pTemperatures;
  BYTE byErr;
  while (TRUE)
  {
    pTemperatures = (int *) OSQPend (pOseQueueTemp, WAIT_FOREVER, &byErr);
    if (pTemperatures[0] != pTernperatures[1])
    !! Set  off howling alarm;
    free (pTemperatures);
  }
}
```

- This task calls the C library malloc function to allocate a new data buffer for each pair of temperatures and writes a pointer to that buffer into the queue.

Pointers and queues example

```
//Queue  function  prototypes
OS_EVENT *OSQCreate(void **ppStart, BYTE bySize);
Unsigned char OSQPost(0S_EVENT *pOse,void  *pvMsg);
void *OSQPend(0S_EVENT *pOse,WORD wTimeout,BYTE *pByErr);
#define WAIT_FOREVER 0
Static 0S_EVENT *pOseQueueTemp;

void  vReadTemperaturesTask  (void)
{
  int *pTemperatures;
  while (TRUE)
  {
    !!Wait until the time to read the next temperature
    //Get a new buffer for the new set of temperatures.
    pTemperatures=(int *) malloc (2*sizeof*pTemperatures);

    pTemperatures[0] = !!read in value from hardware;
    pTemperatures[1] = !!Read in value from hardware;

    //Add a pointer to the new temperatures to the queue
    OSQPost (pOseQueueTemp, (void *) pTemperatures);
  }
}
void vMainTask (void)
{
  int *pTemperatures;
  BYTE byErr;
  while (TRUE)
  {
    pTemperatures = (int *) OSQPend (pOseQueueTemp, WAIT_FOREVER, &byErr);
    if (pTemperatures[0] != pTernperatures[1])
    !! Set  off howling alarm;
    free (pTemperatures);
  }
}
```

- vMainTask subsequently reads the pointer to the buffer from the queue, compares the temperatures, and frees the buffer.

Mailboxes

Mailboxes introduction

- Mailboxes are like queues.
- The typical RTOS has functions to **create**, to **write** to, and to **read** from mailboxes, and functions to check whether the mailbox **contains any messages** and to **destroy** the mailbox if it is no longer needed.
- Some RTOSs allow a certain number of messages in each mailbox (chosen during compile time), others allow only one message in a mailbox at a time. Once one message is written to a mailbox under this last systems, the mailbox is full; no other message can be written to the mailbox until the first one is read.

Prioritize mailbox messages

- In certain RTOS, you can prioritize mailbox messages.
- For example, in the MultiTask! RTOS each message is a void pointer. You must create all of the mailboxes you need when you configure the system, after which you can use these three functions:

```
int sndmsg  (unsigned int uMbId, void *p_vMsg, unsigned int uPriority);

void *rcvmsg  (unsigned  int  uMbld,   unsigned  int uTimeout);

void *chkmsg  (unsigned  int  uMbld);
```

sndmsg function

```
int sndmsg  (unsigned int uMbId, void *p_vMsg, unsigned int uPriority);
void *rcvmsg  (unsigned  int  uMbld,   unsigned  int uTimeout);
void *chkmsg  (unsigned  int  uMbld);
```

- the **uMbId** parameter identifies the mailbox on which to operate.
- The **sndmsg** function adds **p_vMsg** into the queue of messages held by the **uMbId** mailbox with the priority indicated by **uPriority**
- It returns an error if **uMbId** is invalid or if too many messages are already pending in mailboxes.

rcvmsg and chkmsg functions

```
int sndmsg  (unsigned int uMbId, void *p_vMsg, unsigned int uPriority);
void *rcvmsg  (unsigned  int  uMbId,   unsigned  int uTimeout);
void *chkmsg  (unsigned  int  uMbId);
```

- The **rcvmsg** function returns the highest-priority message from the specified mailbox; it blocks the task that called it if the mailbox is empty.
- The task can use the **uTimeout** parameter to limit how long it will wait if there are no messages
- The **chkmsg** function returns the first message in the mailbox; it returns a NULL immediately if the mailbox is empty.

Pipes

Introduction to pipes

- Pipes are also like queues.
- The RTOS can **create**, **write** , **read**, and so on.
- Pipes in some RTOSs are entirely byte-oriented. For example:
  - **Task A** writes **11 bytes** to the pipe and then **Task B** writes **19 bytes** to the pipe
  - If **Task C** reads **14 bytes** from the pipe, it will get the 11 bytes that Task A wrote plus the first 3 bytes that Task B wrote.
  - The **other 16 bytes** that task B wrote remain in the pipe for whatever task reads from it next.

How are pipes implemented?

- Some RTOSs use the standard C library functions **fread** and **fwrite** to read from and write to pipes.

Which one to use?

So... which one to use?

- Since queues, mailboxes, and pipes vary so much from one RTOS to another, it is hard to give much universal guidance about which to use in any given situation.

- When RTOS vendors design these features, they must make the usual programming trade-offs among flexibility, speed, memory space, the length of time that interrupts must be disabled within the RTOS functions, and so on.

- Most RTOS vendors describe these characteristics in their documentation; read it to determine which of the communications mechanisms best meets your requirements.

Pitfalls

- Pitfall #1: Queues, mailboxes, and pipes make it easy to share data among tasks, but they also make it easy to insert bugs.
- Pitfall #2: Most RTOSs do not restrict which tasks can read from or write to any given queue, mailbox, or pipe. Therefore, you must ensure that tasks use the correct one each time.
- **Pitfall #3: RTOS cannot ensure that data written onto a queue, mailbox, or pipe will be properly interpreted by the task that reads it.**
  - For example, if one task writes an integer onto the queue and another task reads it and then treats it as a pointer, your product will not work.

RTOS cannot ensure that data written will be properly interpreted...

Code #1

```
//Compiler will find the bug

//This function takes a pointer parameter
void vFunc (char *p_ch);
void main  (void)
{
  int i ;
  (...)
  //Call it with an int..
  //and get a compiler error
  vFunc (i);
  (...)
}
```

Code #2

```
//compiler will NOT find the bug

static OS_EVENT *pOseQueue;

void TaskA (void)
{
  int i;
  (...)
  //Put an integer on the queue.
  OSQPost (pOseQueue,(void *) i);
  (...)
}

void TaskB (void)
{
  char *p_ch;
  BYTE byErr;
  (...)
  //Reads from the queue and
  //expects a character pointer.
  p_ch=(char *) OSQPend (pOseQueue,
FOREVER, byErr);
  (...)
}
```

More pitfalls

- Pitfall #4: Running out of space in queues, mailboxes, or pipes is usually a disaster for embedded software.
  - When one task needs to pass data to another, it is usually **not optional**. Often, the only solution is to make your queues, mailboxes, and pipes large enough in the first place.
- Pitfall #5: Passing pointers from one task to another through a queue, mailbox, or pipe is one of several ways to create shared data inadvertently.

Pitfall #5 example: Shared data problem from passing pointers between tasks

```
//Queue function  prototypes
OS_EVENT *OSQCreate (void **ppStart, BYTE bySize);
unsigned char OSQPost (0S_EVENT *pOse, void *pvMsg);
void *0SQPend (0S_EVENT *pOse, WORD wTimeout,BYTE *pByErr);
#define WAIT_FOREVER 0
static OS_EVENT *pOseQueueTemp;

void vReadTemperaturesTask (void)
{
  int  iTemperatures[2] ;
  while   (TRUE)
  {
    !!Wait until it's time to read the next temperature
    iTemperatures[0] = !!read in  value  from hardware;
    iTemperatures[1] = !!read in value from hardware;
    //Add to the queue a pointer to the temperatures
    //we just read
    OSQPost (pOseQueueTemp, (void *) iTemperatures);
  }
}
void vMainTask  (void)
{
 int *pTemperatures;
 BYTE byErr;
 while   (TRUE)
 {
   pTemperatures = (int *)OSQPend (pOseQueueTemp,
WAIT_FOREVER, &byErr);
   if  (pTemperatures[0] ! = pTemperatures[1])
   !!  Set off howling alarm;
  }
}
```

When the main task gets a value for pTemperatures from the queue, pTemperatures will point to the iTemperatures array in vReadTemperaturesTask.

```
//Queue function  prototypes
OS_EVENT *OSQCreate (void **ppStart, BYTE bySize);
unsigned char OSQPost (0S_EVENT *pOse, void *pvMsg);
void *0SQPend (0S_EVENT *pOse, WORD wTimeout,BYTE *pByErr);
#define WAIT_FOREVER 0
static OS_EVENT *pOseQueueTemp;

void vReadTemperaturesTask (void)
{
  int  iTemperatures[2] ;
  while   (TRUE)
  {
    !!Wait until it's time to read the next temperature
    iTemperatures[0] = !!read in  value  from hardware;
    iTemperatures[1] = !!read in value from hardware;
    //Add to the queue a pointer to the temperatures
    //we just read
    OSQPost (pOseQueueTemp, (void *) iTemperatures);
  }
}
void vMainTask  (void)
{
 int *pTemperatures;
 BYTE byErr;
 while   (TRUE)
 {
   pTemperatures = (int *)OSQPend (pOseQueueTemp,
WAIT_FOREVER, &byErr);
   if  (pTemperatures[0] ! = pTemperatures[1])
   !!  Set off howling alarm;
  }
}
```

If the RTOS switches from vMainTask to vReadTemperaturesTask while vMainTask was comparing iTemperatures[0] to iTemperatures[1], and if vReadTemperaturesTask then changes the values in iTemperatures, you will have the shared-data bugs.

```
//Queue function  prototypes
OS_EVENT *OSQCreate (void **ppStart, BYTE bySize);
unsigned char OSQPost (0S_EVENT *pOse, void *pvMsg);
void *0SQPend (0S_EVENT *pOse, WORD wTimeout,BYTE *pByErr);
#define WAIT_FOREVER 0
static OS_EVENT *pOseQueueTemp;

void vReadTemperaturesTask (void)
{
  int  iTemperatures[2] ;
  while   (TRUE)
  {
    !!Wait until it's time to read the next temperature
    iTemperatures[0] = !!read in  value  from hardware;
    iTemperatures[1] = !!read in value from hardware;
    //Add to the queue a pointer to the temperatures
    //we just read
    OSQPost (pOseQueueTemp, (void *) iTemperatures);
  }
}
void vMainTask  (void)
{
 int *pTemperatures;
 BYTE byErr;
 while   (TRUE)
 {
   pTemperatures = (int *)OSQPend (pOseQueueTemp,
WAIT_FOREVER, &byErr);
   if  (pTemperatures[0] ! = pTemperatures[1])
   !!  Set off howling alarm;
  }
}
```

The similar code on slide 15 did not have this problem, because vMainTask and vReadTemperaturesTask never use the same memory location at the same time.

Timer functions

Most embedded systems must keep track of the passage of time

For example:

- To extend its battery life, a cordless bar-code scanner must turn itself off after a certain number of seconds.
- Systems with network connections must wait for acknowledgements to data that they have sent and retransmit the data if an acknowledgement doesn't show up on time.
- Manufacturing systems must wait for robot arms to move or for motors to come up to speed.

Delay is a task blocking operation

- Most RTOSs offer a function that delays a task for a period of time
- That is... blocks the task until the period of time expires.

Telephone call example

vMakePhoneCallTask

```
//Message queue for phone numbers to dial.
extern MSG_Q_ID queuePhoneCall;
void vMakePhoneCallTask (void)
{
  #define MAX_PHONE_NUMBER    11
  char a_chPhoneNumber[MAX_PHONE_NUMBER];
  //Buffer for null-terminated ASCII number
  char *p_chPhoneNumber;
  //Pointer into a_chPhoneNumber
  while  (TRUE){
  msgQreceive (queuePhoneCall, a_chPhoneNumber,
MAX_PHONE_NUMBER, WAIT_FOREVER);
  //Dial  each of the digits
  p_chPhoneNumber = a_chPhoneNumber;
  while  (*p_chPhoneNumber)
  {
    taskDelay(100); //1/10th of a second silence
    vDialingToneOn(*p_chPhoneNumber  -'0');
    taskDelay(100); //1/10th of a second with tone
    vDialingToneOff ();
    //Go to the next digit in the phone number
    ++p_chPhoneNumber;
    }
   (...)
    }
}
```

In the US each of the tones that represents a digit must sound for one-tenth of a second, and there must be one-tenth-second silences between the tones.

vMakePhoneCallTask receives a phone number from an RTOS message queue

taskDelay

```
//Message queue for phone numbers to dial.
extern MSG_Q_ID queuePhoneCall;
void vMakePhoneCallTask (void)
{
  #define MAX_PHONE_NUMBER    11
  char a_chPhoneNumber[MAX_PHONE_NUMBER];
  //Buffer for null-terminated ASCII number
  char *p_chPhoneNumber;
  //Pointer into a_chPhoneNumber
  while  (TRUE){
  msgQreceive (queuePhoneCall, a_chPhoneNumber,
MAX_PHONE_NUMBER, WAIT_FOREVER);
  //Dial  each of the digits
  p_chPhoneNumber = a_chPhoneNumber;
  while  (*p_chPhoneNumber)
  {
    taskDelay(100); //1/10th of a second silence
    vDialingToneOn(*p_chPhoneNumber  -'0');
    taskDelay(100); //1/10th of a second with tone
    vDialingToneOff ();
    //Go to the next digit in the phone number
    ++p_chPhoneNumber;
    }
   (...)
   }
}
```

The function **msgQreceive** copies the phone number from the queue into **a_chPhoneNumber**.

While-loop calls **taskDelay** first to create a silence and then to create a tone of appropriate length for each digit in the phone number.

msgQreceive and taskDelay are VxWorks functions

```
//Message queue for phone numbers to dial.
extern MSG_Q_ID queuePhoneCall;
void vMakePhoneCallTask (void)
{
  #define MAX_PHONE_NUMBER    11
  char a_chPhoneNumber[MAX_PHONE_NUMBER];
  //Buffer for null-terminated ASCII number
  char *p_chPhoneNumber;
  //Pointer into a_chPhoneNumber
  while  (TRUE){
  msgQreceive (queuePhoneCall, a_chPhoneNumber,
MAX_PHONE_NUMBER, WAIT_FOREVER);
  //Dial  each of the digits
  p_chPhoneNumber = a_chPhoneNumber;
  while  (*p_chPhoneNumber)
  {
    taskDelay(100); //1/10th of a second silence
    vDialingToneOn(*p_chPhoneNumber  -'0');
    taskDelay(100); //1/10th of a second with tone
    vDialingToneOff ();
    //Go to the next digit in the phone number
    ++p_chPhoneNumber;
    }
   (...)
    }
}
```

The functions vDialingToneOn and vDialingToneOff turn the tone generator on and off.

The **msgQreceive** and **taskDelay** functions in this code are from VxWorks.

taskDelay function takes a number of milliseconds as its parameter?

```
//Message queue for phone numbers to dial.
extern MSG_Q_ID queuePhoneCall;
void vMakePhoneCallTask (void)
{
  #define MAX_PHONE_NUMBER    11
  char a_chPhoneNumber[MAX_PHONE_NUMBER];
  //Buffer for null-terminated ASCII number
  char *p_chPhoneNumber;
  //Pointer into a_chPhoneNumber
  while  (TRUE){
  msgQreceive (queuePhoneCall, a_chPhoneNumber,
MAX_PHONE_NUMBER, WAIT_FOREVER);
  //Dial  each of the digits
  p_chPhoneNumber = a_chPhoneNumber;
  while  (*p_chPhoneNumber)
  {
    taskDelay(100); //1/10th of a second silence
    vDialingToneOn(*p_chPhoneNumber  -'0');
    taskDelay(100); //1/10th of a second with tone
    vDialingToneOff ();
    //Go to the next digit in the phone number
    ++p_chPhoneNumber;
    }
   (...)
   }
}
```

No. The taskDelay function in VxWorks (like most RTOSs), takes the number of system ticks as its parameter.

The length of time represented by each system tick is something you can usually control when you set up the system.

How accurate are the delays produced by the taskDelay function?

```
//Message queue for phone numbers to dial.
extern MSG_Q_ID queuePhoneCall;
void vMakePhoneCallTask (void)
{
  #define MAX_PHONE_NUMBER    11
  char a_chPhoneNumber[MAX_PHONE_NUMBER];
  //Buffer for null-terminated ASCII number
  char *p_chPhoneNumber;
  //Pointer into a_chPhoneNumber
  while  (TRUE){
  msgQreceive (queuePhoneCall, a_chPhoneNumber,
MAX_PHONE_NUMBER, WAIT_FOREVER);
  //Dial  each of the digits
  p_chPhoneNumber = a_chPhoneNumber;
  while  (*p_chPhoneNumber)
  {
    taskDelay(100); //1/10th of a second silence
    vDialingToneOn(*p_chPhoneNumber  -'0');
    taskDelay(100); //1/10th of a second with tone
    vDialingToneOff ();
    //Go to the next digit in the phone number
    ++p_chPhoneNumber;
    }
   (...)
   }
}
```

They are accurate to the nearest, system tick.

The RTOS works by setting up a single hardware timer to interrupt periodically, say, every millisecond, and bases all timings on that interrupt.

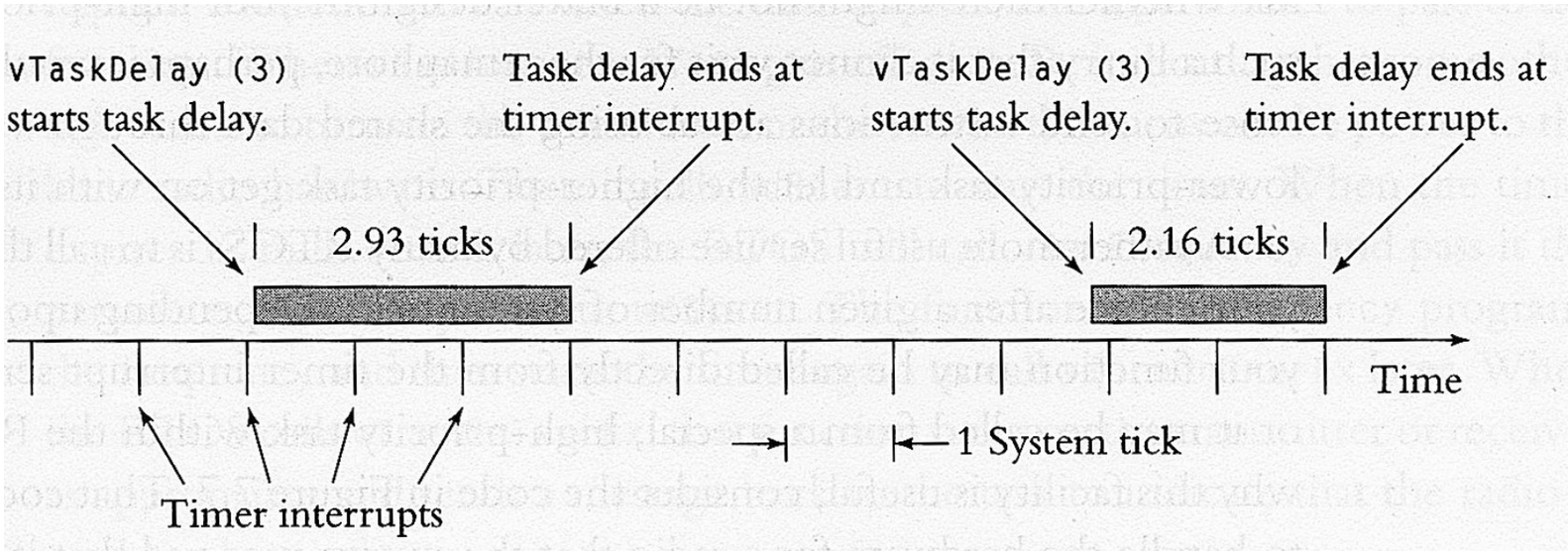
Heartbeat timer

```
//Message queue for phone numbers to dial.
extern MSG_Q_ID queuePhoneCall;
void vMakePhoneCallTask (void)
{
  #define MAX_PHONE_NUMBER    11
  char a_chPhoneNumber[MAX_PHONE_NUMBER];
  //Buffer for null-terminated ASCII number
  char *p_chPhoneNumber;
  //Pointer into a_chPhoneNumber
  while  (TRUE){
  msgQreceive (queuePhoneCall, a_chPhoneNumber,
MAX_PHONE_NUMBER, WAIT_FOREVER);
  //Dial  each of the digits
  p_chPhoneNumber = a_chPhoneNumber;
  while  (*p_chPhoneNumber)
  {
    taskDelay(100); //1/10th of a second silence
    vDialingToneOn(*p_chPhoneNumber  -'0');
    taskDelay(100); //1/10th of a second with tone
    vDialingToneOff ();
    //Go to the next digit in the phone number
    ++p_chPhoneNumber;
    }
   (...)
    }
}
```

This timer is often called the heartbeat timer.

For example, if one of your tasks passes 3 to taskDelay, that task will block until the heartbeat timer interrupts three times.

Timer function accuracy



- The first timer interrupt may come almost immediately after the call to taskDelay or it may come after just under one tick time or after any amount of time between those two extremes.

- The task will therefore be blocked for a period of time that is between just a tiny more than two system ticks and just a tiny less than three system ticks.

Setting up timer hardware

- **Question**: How does the RTOS know how to set up the timer hardware on my particular hardware?
- It is common for microprocessors to have timers.
- If you are using nonstandard timer hardware, then you may have to write your own timer setup software and timer interrupt routine.
- The RTOS will have an entry point for your interrupt routine to call every time the timer expires.
- Many RTOS vendors provide board support packages , which contain driver software for common hardware components.

What is a "normal" length for the system tick?

- There really isn't one.
- The **advantage** of a short system tick is that you get accurate timings.
- The **disadvantage** is that the microprocessor must execute the timer interrupt routine frequently.
- Since the hardware timer that controls the system tick usually runs all the time, whether or not any task has requested timing services, a short system tick can decrease system throughput quite considerably by increasing the amount of microprocessor time spent in the timer interrupt routine.

What if my system needs extremely accurate timing?

- One choice is to make the system tick short enough that RTOS timings fit your definition of "extremely accurate."
- Other choice is to use a separate hardware timer for those timings that must be extremely accurate.
- It is not uncommon to design an embedded system that uses dedicated timers for a few accurate timings and uses the RTOS functions for other timings that need not be so accurate.
- The advantage of the RTOS timing functions is that one hardware timer times many number of operations simultaneously.