Real-Time Operating Systems

Fundamental Concepts and Practices
but all we have is...

CPU + memory + peripherals

Q: How can we build complex computer control systems?
A: Multiprocessing, multitasking and multithreading

- Let’s look at each in turn
Parallel Execution

- Computer program is usually executed as a single sequence of instructions because:
  - Only one CPU (uniprocessor), with
  - Sequential processor logic rather than parallel logic (regardless of von Neumann or Harvard architecture)
  - Language supports only sequential flow of instructions
A uniprocessor can effectively execute multiple programs by giving each a slice of CPU time, and *swapping* them in and out of execution according to some scheme.

Requires another program to do this swapping – a *scheduler*, part of a *real time kernel* or *real time operating system* (RTOS).

Comment: Available even for PIC – Salvo, uCOS, …
Processes

- Three programs, A, B and C sharing a single processor
- If the programs cannot directly influence each other, they are called processes
- Each will have its own PID, memory, etc.
- Can represent the processor and processes as shown
- One processor executing three processes
- Conceptually there are three `main()` functions
What is a Process?

- A process is a program that has sole control of some system resources
- Each process has its own
  - Program and data address space
  - Signals, semaphores
  - Open files, etc.
- Processes cannot therefore directly share these resources, and are so protected from each other
- Processes are usually established and maintained by an operating system (OS)
- The OS must provide facilities for inter-process communication if required (and it will be...)
Multi-threading

- In most traditional OSs, each process has an address space and a single thread of execution
- Multiple threads of execution are now common
- Advantages in having more than one thread:
  - One can run when others are waiting (e.g. for input events)
  - Gives higher throughput/performance
What is a Thread?

- A complete program that can run on a processor, including the required processor resources
- Threads are created (spawned) from within a process
- Each thread has its own
  - PC
  - Stack
  - Register set
  ⇒ the “context” of the executing program
- Swapping from executing one thread to executing another is called context-switching
- Context-switch time is a key performance indicator in a real-time operating system (RTOS)
Example

- Threads executing in a process can be represented as shown.
- Three threads in one process, executing on one processor.
- Triangles represent the PCs of each thread.
Another example

- Three threads in Process A, one thread in Process B, both executing on one processor
Processes vs Threads

- Processes are “hostile” to each other
  - different owners
- Each process has its own
  - address space
  - global variables
  - open files
  - signals and semaphores, etc.

and the OS protects them against other processes
Processes vs Threads

- Threads co-operate with each other
  - same owner
- Each thread has its own
  - PC
  - stack
  - register set
What is a Real Time Operating System?

- Define a RTOS by what it must do: manage execution of program activities ("tasks") so that
  - Multiple tasks effectively run in parallel
  - Higher priority tasks allocated a larger share of processor time
  - All tasks complete by their required deadlines

- An OS (including a RTOS) must also
  - Provide communication and synchronisation between tasks
  - Manage the use of resources (e.g. variables in memory, peripheral hardware) that are shared between tasks.

- An RTOS is general software, adapted for an application by
  - Configuring it
  - Writing code for the tasks
What Does “Real Time” Imply?

- Being able to execute tasks at intervals of real time
- Meeting completion deadlines in the real world
  - Hard real-time: failure to meet a time deadline means system failure – e.g. cardiac pacemaker
  - Soft real-time: small time overruns on deadlines are acceptable
There are several structures commonly used in embedded systems programming

- Sequential execution – the “super loop”
- Timed super loop
- Background/foreground – interrupts for prioritisation
- Schedulers of various types within a RTOS

- Multithreading is usually regarded as a subset of multitasking.
Super Loop Structure

A “super loop” structure simply loops endlessly

- Loop execution time is not constant
- Loop execution time is not deterministic, nor related to real (physical) time
- Tasks **block** each other – a task that is ready to execute has to wait for the preceding task to finish
- No task prioritisation mechanism to give higher priority tasks more time
Adding a hardware timer tick can help a little

- At least one task executes at regular intervals
- These intervals are measured in real time

But still

- Processor time is wasted when waiting
- Tasks block
- There is no prioritisation
Foreground/Background Structure

Adding interrupts allows prioritisation
- The super loop (untimed or timed) is the lower priority “background” task
- Higher priority tasks are interrupt-driven in the “foreground”
- Foreground tasks are often event-driven
A Step Towards a General OS Structure

This structure
- Requires one task counter for each task
- Task counters allow tasks to run at different rates
- Gains responsiveness and efficiency by not executing every task on every loop
- Allows tasks to be enabled / disabled
- Background tasks still block
Schedulers

- The scheduler is a central part of an RTOS
- Determines which task is allowed to run at any time
- Must account for
  - Which task(s) are ready to run
  - Which ready task has the highest priority

- The scheduler’s job: run the highest priority task that is ready to run (eligible)
Cyclic “Scheduling”

The simplest “scheduler”
- Same as a super loop
- Many disadvantages

- No scheduler activity required...
Round-Robin Scheduling

- System driven by a periodic interrupt: the “clock tick”
- Tasks run in fixed sequence, with each in turn given a fixed-length slice of processor time
- Tasks are “pre-empted” (interrupted) on the system clock tick
- Pre-emption requires significant scheduler overhead to save and restore the context of each task (“context switch”)
Task States – The Life Cycle of a Task

Once task execution is controlled by a scheduler, efficiency can be gained by recognising task states:

- **Ready (or Eligible):** Ready to run and will do so when started
- **Running:** Allocated CPU time and executing
- **Blocked/Waiting:** Unable to run because it needs some data or other resource
- **Stopped/Suspended:** Inactive because it doesn’t currently need CPU time
- **Destroyed:** Removing unneeded tasks improves efficiency. They can be recreated if needed.
Prioritised Pre-emptive Scheduling

Three prioritised tasks
- The scheduler runs the highest-priority ready task
- Higher-priority tasks complete before any time is allocated to lower priority tasks
- This is a classical RTOS
Cooperative Scheduling

- Pre-emptive scheduling has significant CPU and memory overhead in context switching
- This is particularly severe for small systems (PIC18!)

Cooperative scheduling:
- Each task relinquishes CPU access at a time of its choosing
- It can therefore control its own context switching, minimising scheduler overhead
- Not as responsive to tight deadlines as a pre-emptive scheduler
- Needs less memory and can task-switch quickly
Interrupts and Scheduling

- First use of interrupts is to provide the system clock tick, usually by a timer overflow interrupt.
- Note that the ISRs are not tasks in themselves (unlike in Foreground/Background).
- Interrupts can be used to supply event information to the tasks or scheduler: e.g. releasing a blocked task by signalling from within an ISR.
Defining Tasks

An early step in defining a software architecture using a RTOS is to define the tasks

- Not too many: many tasks leads to complex software and time/memory overhead in context switch
- Define the deadlines, then try one task per deadline
- A task is in some ways like a module: activities that are closely related and operate on the same data usually belong in the same task
Structuring Task Code

- Completely different from foreground/background!
- Each task should be written
  - To run continuously – it will be stopped/started by the RTOS
  - To be as autonomous as possible, to minimise need for communication with other tasks
- Tasks
  - May not directly call code from other tasks
  - May depend on services (data etc.) provided by other tasks
  - May therefore need to be synchronised with other tasks
  - These services must be provided by the RTOS
Setting Task Priorities

- Priorities can be
  - Static – fixed priorities are often quite adequate
  - Dynamic – change during run time

- Allocate priorities
  - Highest priority: tasks essential for system survival
  - Middle priority: tasks essential for correct system operation
  - Lowest priority: tasks that can be skipped or delayed

- Tasks with the most critical/tightest deadlines must have the highest priority
- Not all tasks can have high priority!
Events and Intertask Communication

- For a task to react to an event, the event must trigger some communication with the task.
- All methods of intertask communications involve:
  - **Signalling**: sending a communication via the OS
  - **Waiting**: receiving a communication via the OS
- Common methods include:
  - Semaphores (binary and counting)
  - Messages
  - Message queues
Semaphores

- A semaphore is a variable maintained by the OS that can take on a range of values.
- A **binary semaphore** (sometimes ‘binsem”) can only take the values 0 or 1.
- A **counting semaphore** (usually just “a semaphore”) can take values from 0 to \( n \).
A binary semaphore can be used to indicate that an event has occurred.

Create a binary semaphore and initialise it to 0 to indicate that the event has not yet occurred.

When the event occurs, the semaphore is set to 1 by *signalling the semaphore* from a task, ISR or background code.

Meanwhile another task *waits the semaphore*. This task is released when the semaphore is received.

---

// Pseudocode
// Signal event from ISR

// In Task 1
create binsem bufferFull;
initialise bufferFull to 0;
...

// In Task 2
for (;;) {
    ...
    wait binsem bufferFull;
    // Event has now occurred
    ...
}

// In ISR (for example)
...
if (buffer is full) {
    signal binsem bufferFull;
}
...
BinSem for Task Synchronisation

- Since tasks can wait on semaphores, a binary semaphore can be used to indicate that a task has finished.
- This can be used to synchronise or otherwise coordinate two or more tasks.
- Useful when (for example) two tasks must both complete for some set of information to be available.
- In example, TaskUpper() and TaskLower() alternately run to completion.

```c
// Pseudocode
// Synchronise two tasks
create binsem semLower, semUpper;
initialise semLower to 0;
initialise semUpper to 1;

TaskUpper ()
{
   for (;;)
   {
      // Wait for LowerTask()
      wait binsem semLower;
      do stuff;
      signal binsem semUpper;
   }
}

TaskLower ()
{
   for (;;)
   {
      do stuff;
      signal binsem semLower;
      // Wait for UpperTask()
      wait binsem semUpper;
   }
}
```
Binary Semaphore for Mutual Exclusion

- A binary semaphore can be used to protect a resource that is shared between tasks
  - Resource = a variable or peripheral
  - Shared between = accessed by
- Task can only access the resource by *acquiring* it by successfully waiting the semaphore

```
// Pseudocode
// Two tasks accessing a display

create binsem semDisplay;
initialise semDisplay to 1;

TaskShowTime()
{
    for (;;)
    {
        prepare time string;
        wait binsem semDisplay;
        write time string to display;
        signal binsem semDisplay;
    }
}

TaskShowAlert()
{
    for (;;)
    {
        wait binsem semDisplay;
        write alert string to display;
        signal binsem semDisplay;
    }
}
Counting Semaphore

- A counting semaphore is associated with (protects) a set of \( n \) identical resources that are shared between tasks.
- The semaphore is initialised to \( n \).
- The semaphore is decremented when a task successfully waits the semaphore and acquires the use of one of the resources.
- The semaphore is incremented when a task signals the semaphore and releases the resource.
- A counting semaphore therefore holds the number of the resource units that are available for use.
Counting Semaphore (ctd)

- If a `wait()` request would cause the semaphore to go negative the requesting task is blocked on the semaphore and added to the semaphore’s wait queue.
- If a `signal()` would cause the semaphore to increase from, one task is transferred from the semaphore’s wait queue to its ready queue.

Historical note: `wait()` is \( P() \), from “probeer te verlagen”: literally “try to reduce” and `signal()` is \( V() \) for “verhogen”: “increase”; Edsger Dijkstra (1930–2002) was Dutch.
Semaphore for Circular Buffer

- Implementation of circular buffer of length LEN
- Task writes to tail pointer
- ISR reads from head pointer

```
// Pseudocode
// Circular buffer via semaphore

create semaphore semTxBuff;
initialise semTxBuff to LEN;

TaskFillTxBuffer()
{
    for (;;)
    {
        wait semaphore semTxBuff;
        write char to TxBuff[tail];
        increment tail pointer;
    }
}

ISR TxChar()
{
    send char at TxBuff[head] out RS-232
    increment head pointer
    signal semTxBuff
}
Messages

- Messages provide a way of sending arbitrary information to a task
- A message typically contains a pointer to the information rather than the information itself (since a message sends information of arbitrary type)
- The pointer must therefore correctly be de-referenced
- A task sends a message by signaling it
- A message is received by a task by waiting it

```
// Pseudocode
// Send information via message - in this example a text string

// In Task 1
create message msgString;
string[] = “a message”;
msgString ptr = address( string );
signal message msgString

// In Task 2
for (;;)
{
    // Wait message containing ptr
    wait msgString;

    // Do something with *ptr
    // Note the explicit cast
    // in dereferencing the pointer.
    character = *(char *) ptr;
}
```
Message Queues

- Message queues are an extension of messages
- A message queue can contain multiple messages up to a predetermined number
- Sending (signaling) messages to the queue can continue until the message mailbox is full
- A task that waits the message queue will receive (in FIFO order) messages until the message queue is empty
Task / Event Rules Summary

1. An event must be initialised. It is initialised with no waiting tasks.
2. A task cannot wait an event until the event has been initialised.
3. Only a task can wait an event.
4. A task can only wait one event at a time.
5. A semaphore’s value can range from 0 to a maximum value that depends on its size (8-bit, 16-bit etc).
6. A message contains a *pointer* to some information.
7. Message queues can hold multiple messages at once.
8. An event can be signaled from *anywhere*: ISR, task or background.
Task / Event Rules Summary

9. A task will be blocked (change to waiting state) if the event it waits is not available

10. Multiple tasks can wait a single event

11. Which waiting task becomes eligible when an event is signaled depends on how the OS implements event services (FIFO or highest priority)

12. If an event has been signaled, no task is waiting it, and it is signaled again then either (a) an error has occurred or (b) the signaling task is blocked. This depends on how the OS implements event services.
Deadlock

- Deadlock occurs with two or more tasks when each task is waiting for a resource that is controlled (has been acquired) by another task.
- No task can acquire the resource that it needs so each task will wait forever.
Deadlock

- To avoid deadlock, each task must
  - Acquire the resources in a predetermined order; and
  - Acquire \textit{all} required resources before continuing; and
  - Release the resources in the reverse order

- \textbf{Waiting with a \textit{timeout} can break a deadlock}
  - If timeout occurs, the \texttt{WaitWithTimeout()} returns an error code and execution continues
  - The error code can be used to decide what to do next
Priority Inversions

- Priority inversions occur when a high-priority task is waiting on a resource that is controlled by a lower-priority task.
- The priority of the higher-priority task is effectively reduced to that of the lower-priority task.
Mutex

- A mutex is like a binary semaphore but has an *owner* – the task that locked the mutex
- Only the task that locked (acquired) the mutex can unlock it
- Mutexes are not subject to priority inversion since the priority of the owner can be increased if a higher-priority task starts waiting the mutex

- Salvo does not support mutexes (or spinlocks)
  - See http://www.barrgroup.com/Embedded-Systems/How-To/RTOS-Mutex-Semaphore
On Mutual Exclusion and Inter-Task Communication and Synchronisation

See the following interesting thought experiments...

- The *Cigarette Smokers Problem*
- The *Dining Philosophers Problem*
- The *Sleeping Barber Problem*
The Salvo RTOS
Salvo

- Salvo is a commercial RTOS supplied by Pumpkin Inc
- Specifically designed for small embedded systems
- Supports Microchip C18 compiler and compilers from eight other manufacturers
- Supports PIC18 and 11 other target uC families
- Free version “Salvo Lite’ – limited to three tasks and five events
- Documentation is very good – use it!
  - Salvo User Manual v 3.2.3 (compiler and target agnostic)
  - Salvo Compiler Reference Manual for Microchip MPLAB-C18
Basic Features of Salvo

- Salvo is a cooperative multitasking RTOS with full support for event and timer services
- Multiple prioritised tasks (limited by available memory)
- Up to 16 priority levels (0 is highest, 15 lowest)
- Events for inter-task communications and resource management:
  - Binary semaphores
  - Counting semaphores
  - Event flags
  - Messages
  - Message queues
Salvo Builds

There are two ways to build a Salvo project

- “Library build” – build user code, link to a Salvo library
- “Source-code build” – requires Salvo Pro to build source code

Files that must be in every build (source or library)

- `mem.c` – file that defines global objects that Salvo uses for tasks, semaphores, etc. Do not edit!
- `salvo.h` – the main header. Include in any source file that calls Salvo services. Do not edit! `salvo.h` includes `salvocfg.h`
- `salvocfg.h` – user-defined header file that configures Salvo for the user’s particular application. Defines number of tasks and events, any libraries, etc.
Which Salvo Library?

- A Salvo library build must link with a Salvo library
- Which library configuration should you choose?

<table>
<thead>
<tr>
<th>Library Configuration</th>
<th>m</th>
<th>d</th>
<th>e</th>
<th>a</th>
<th>t</th>
</tr>
</thead>
<tbody>
<tr>
<td>Multitasking</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>Delays</td>
<td></td>
<td>✓</td>
<td></td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>Events</td>
<td></td>
<td></td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>Idling</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>Priorities</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>Timeouts</td>
<td></td>
<td>✓</td>
<td></td>
<td></td>
<td>✓</td>
</tr>
</tbody>
</table>

Link with the smallest library that contains what you need
- ‘m’ is basic multitasking
- ‘d’ has delays...
- ‘e’ has events
- ‘a’ has all, except timeouts
- ‘t’ really does have everything
Salvo Library Naming Convention

- **Salvo**
- **Library Type**
  - l = standard
  - f = freeware
- **MPLAB C18**
- **Memory Model**
  - l = large (2MB address space)
  - s = small (64K address space)

---

**Library Configuration**
- m = multitasking only
- d = multitasking with delays
- e = multitasking with events
- a = multitasking with delays and events
- t = multitasking with delays and events; tasks can wait with timeouts

**Memory type for global Salvo objects**
- f = far (16-bit; banked RAM)
- n = near (8-bit; access RAM)
Salvo Configuration

- Salvo is configured for a particular application by editing the user file `salvocfg.h`.
- Salvo is *highly* configurable (particularly Salvo Pro) and can therefore be confusing.
- Work from simple examples and extend them.
Salvo Configuration

//salvocfg.h for Tute 4 example
 /*******************************************************************************/

#if defined (MAKE_WITH_FREE_LIB)

/* Salvo Lite build */
#define OSUSE_LIBRARY TRUE
#define OSLIBRARY_TYPE OSF /* free library */
#define OSLIBRARY_GLOBALS OSF /* far objects */
#define OSLIBRARY_CONFIG OSE /* config 'e' */
#define OSEVENTS 1
#define OSEVENT_FLAGS 0
#define OSMESSAGE_QUEUES 0
#define OSTASKS 2
#endif

#elif defined (MAKE_WITH_STD_LIB)

/* Salvo LE & Pro library build - link with slc18sfe.lib */
#define OSUSE_LIBRARY TRUE
#define OSLIBRARY_TYPE OSL /* std library */
#define OSLIBRARY_GLOBALS OSF /* far objects */
#define OSLIBRARY_CONFIG OSE /* config 'e' */
#define OSEVENTS 1
#define OSEVENT_FLAGS 0
#define OSMESSAGE_QUEUES 0
#define OSTASKS 2
#endif

David Rye :: MTRX 3700
Real-Time Operating Systems :: Slide 56 of 91
#ifdef (MAKE_WITH_SOURCE)

/* Salvo Pro source-code build */
#define OSENABLE_BINARY_SEMAPHORES TRUE
#define OSEVENTS 1
#define OSTASKS 2

#else

#error No MAKE_WITH_??? symbol defined. See salvocfg.h.
#endif

#if !defined(SYSE)

#error SYSE must be defined for this MPLAB-C18 tutorial project ...
#endif
Salvo Cautions

- Salvo defines many data types – **be careful to use them strictly.**
  - Examples: cast literals to the correct (Salvo) type.
    *(OSTypeMsgP) &CODE_B*

- Salvo defines many constants (macros) – use them
  - Example: `if (error == OSERR_EVENT_FULL)`

- All Salvo functions with names containing `OS_` cause a context switch
  - Example: `OS_Yeild()`, `OS_WaitBinSem()`
Salvo Rule #1

- Rule #1: Every task needs a context switch

```c
void ForlornTask( void )
{
    myFunction();
} // ERROR - execution “drops through” as consequence of scheduler behaviour...

void StuckTask( void )
{
    for (;;)
    {
        myFunction();
        // ERROR - execution stuck in loop since it doesn’t yield to scheduler...
    }
}
```
Salvo Rule #2

- Rule #2: Context switches may only occur in tasks, not in other functions, background code or ISRs

```c
void Task37( void )
{
    while (1)
    {
        myFunction();
    }
}

/* Just an ordinary function */
void myFunction( void )
{
    DoStuff();
    DoThings();

    OS_Yield( ); /* ERROR - Salvo cannot yield to the scheduler from outside a task */
}
```
Salvo Rule #3

- Rule #3: Persistent local variables must be declared as `static`

- Why? Salvo’s context switcher performs a minimal context save that does not include local variables
- `static` variables are allocated fixed locations in RAM and are therefore preserved when the task is not running
Examples: A Salvo Application

- The following sequence of examples gradually develops a Salvo application
- These examples run on the PICDEM board and use the Port B LEDs to show task activity
- See Chapter 4 – Tutorial in the Salvo User Manual
Example: Tute 1, main.c

/***************************************************************************/
Initialising Salvo and running the scheduler. No tasks, no user code and no observable output.
/***************************************************************************/

#include "main.h"
#include "ConfigRegs.h" /* fails if included after salvo.h */
#include <salvo.h>

void main( void )
{
    Init();

    OSInit(); /* initialise Salvo */

    for (;;)
    {
        OSSched(); /* run the scheduler */
    }
}
Example: Tute 2, main.c

```c
#include "main.h"
#include "ConfigRegs.h"    /* fails if included after salvo.h */
#include <salvo.h>

/* Define labels for the context switch */
_OSLabel( TaskA1 )
_OSLabel( TaskB1 )

void main( void ) {
    Init();
    OSInit();               /* initialise Salvo */

    /* Create and start two tasks */
    OSCreateTask( TaskA, OSTCBP(1), 10 ); /* everything about a task is in a TCB */
    OSCreateTask( TaskB, OSTCBP(2), 10 );

    for (;;) {
        OSSched();          /* run the scheduler */
    }

}   // main()
```
Example: Tute 2, main.c

```c
void TaskA( void )
{
    for (;;)
    {
        OS_Yield( TaskA1 ); /* return control to scheduler */
    }
}

void TaskB( void )
{
    for (;;)
    {
        OS_Yield( TaskB1 ); /* return control to scheduler */
    }
}
```
Example: Tute 3, main.c

 disponíveis ao longo do tempo (i.e. código de usuário). Um tarefa conta e a outra exibe o contagem. Saída observável no LED Port B. Tarefas correm o mais rápido possível conforme o processador permite.

```c
#include "main.h"
#include "ConfigRegs.h" /* fails if included after salvo.h */
#include <salvo.h>

/* Define labels for the context switch */
_OSLabel( TaskCount1 )
_OSLabel( TaskShow1 )

unsigned int counter; /* global shared between two tasks */

void main( void ) {
    Init();
    OSInit(); /* initialise Salvo */

    /* Create and start two tasks */
    OSCreateTask( TaskCount, OSTCBP(1), 10 );
    OSCreateTask( TaskShow, OSTCBP(2), 10 );

    counter = 0;

    for (;;) {
        OSSched(); /* run the scheduler */
    }
} // main()
```
Example: Tute 3, main.c

```c
void TaskCount( void )
{
    for (;;)
    {
        counter++;

        OS_Yield( TaskCount1 ); /* return control to scheduler */
    }
}

void TaskShow( void )
{
    InitPORT();

    for (;;)
    {
        /* Note that PORT is a char and changes by 1 every 512 calls of TaskCount() */
        PORT = (PORT & ~0xFE) | ((counter >> 8) & 0xFE);

        OS_Yield( TaskShow1 ); /* return control to scheduler */
    }
}
```
Example: Tute 4, main.c

```c
#include "main.h"
#include "ConfigRegs.h"     /* fails if included after salvo.h */
#include <salvo.h>

/* Defines to improve code readability                          */
#define TASK_COUNT_P            OSTCBP(1)   /* task #1          */
#define TASK_SHOW_P             OSTCBP(2)   /* task #2          */
#define PRIO_COUNT              10          /* task 1 priority  */
#define PRIO_SHOW               10          /* task 2 priority  */
#define BINSEM_UPDATE_PORT_P    OSECBP(1)   /* binSem #1        */

(OSLabel(TaskCount1)
(OSLabel(TaskShow1)
(unsigned int counter;     /* global shared between two tasks */

/* Define labels for the context switch */
(OSLabel( TaskCount1 )
(OSLabel( TaskShow1 )
(OSLabel( TaskBlink1 )

odelist

Inter-task communications: events. The display task waits on an event in the count task. A binary semaphore is used for inter-task communication of the event. Observable output on the Port B LEDs. Tasks run as fast as processor allows, but notably faster than Tute 3 because the waiting task consumes no CPU cycles.

************************************************************************/

Real-Time Operating Systems :: Slide 68 of 91
Example: Tute 4, main.c

```c
void main( void )
{
    Init();
    OSInit();               /* initialise Salvo */

    /* Create and start two tasks */
    OSCreateTask( TaskCount, TASK_COUNT_P, PRIO_COUNT );
    OSCreateTask( TaskShow, TASK_SHOW_P, PRIO_SHOW );

    /* Create a binary semaphore to communicate between tasks */
    OSCreateBinSem( BINSEM_UPDATE_PORT_P, (OStypeBinSem) 0 );

    counter = 0;

    for (;;)
    {
        OSSched();          /* run the scheduler */
    }
}
```
Example: Tute 4, main.c

```c
void TaskCount( void )
{
    for (;;)
    {
        counter++;

        /* Signal if the upper seven bits of counter change */
        if ( !(counter & 0x01FF) )
        {
            OSSignalBinSem( BINSEM_UPDATE_PORT_P );
        }

        OS_Yield( TaskCount1 ); /* return control to scheduler */
    }
}

void TaskShow( void )
{
    InitPORT();

    for (;;)
    {
        /* Wait (forever - no timeout) on BINSEM_UPDATE_PORT_P */
        OS_WaitBinSem( BINSEM_UPDATE_PORT_P, OSNO_TIMEOUT, TaskShow1 );

        PORT = (PORT & ~0xFE) | ((counter >> 8) & 0xFE);
    }
}
```
Delaying a task: real-time operation. Real-time execution is enabled by generating a system tick from a hardware counter (Timer1 – see isr.c). A third task (TaskBlink) runs after a delay of 50 system ticks, therefore at a frequency that can be set accurately in real-time.

```c
#include "main.h"
#include "ConfigRegs.h"     /* fails if included after salvo.h */
#include <salvo.h>

/* Defines to improve code readability                          */
#define TASK_COUNT_P            OSTCBP(1)   /* task #1          */
#define TASK_SHOW_P             OSTCBP(2)   /* task #2          */
#define TASK_BLINK_P            OSTCBP(3)   /* task #3          */
#define PRIO_COUNT              10          /* task 1 priority  */
#define PRIO_SHOW               10          /* task 2 priority  */
#define PRIO_BLINK               2          /* task 3 priority  */
#define BINSEM_UPDATE_PORT_P    OSECBP(1)   /* binSem #1        */

unsigned int counter;        /* global shared between two tasks */

/* Define labels for the context switch */
_OSLabel( TaskCount1 )
_OSLabel( TaskShow1 )
_OSLabel( TaskBlink1 )
```
Example: Tute 5, main.c

```c
void main( void )
{
    Init();
    OSInit(); /* initialise Salvo */

    /* Create and start three tasks */
    OSCreateTask( TaskCount, TASK_COUNT_P, PRIO_COUNT );
    OSCreateTask( TaskShow, TASK_SHOW_P, PRIO_SHOW );
    OSCreateTask( TaskBlink, TASK_BLINK_P, PRIO_BLINK );

    /* Create a binary semaphore */
    OSCreateBinSem( BINSEM_UPDATE_PORT_P, (OStypeBinSem) 0 );

    counter = 0;

    OSEi(); /* enable interrupt response */

    for (;;)
    {
        OSSched(); /* run the scheduler */
    }
}
```
void TaskCount( void )
{
    for (;;)
    {
        counter++;

        /* Signal if the upper seven bits of counter change */
        if ( !(counter & 0x01FF) )
        {
            OSSignalBinSem( BINSEM_UPDATE_PORT_P );
        }

        OS_Yield( TaskCount1 ); /* return control to scheduler */
    }
}

void TaskShow( void )
{
    for (;;)
    {
        /* Wait (forever - no timeout) on BINSEM_UPDATE_PORT_P */
        OS_WaitBinSem( BINSEM_UPDATE_PORT_P, OSNO_TIMEOUT, TaskShow1 );

        PORT = (PORT & ~0xFE) | ((counter >> 8) & 0xFE);
    }
}
Example: Tute 5, main.c

```c
void TaskBlink( void )
{
    InitPORT();       /* initialised here because TaskBlink() will run first */
    for (;;) {
        PORT ^= 0x01;
        /* Return control to scheduler and wait 50 system ticks before this task becomes eligible again. */
        OS_Delay( 50, TaskBlink1 );
    }
}
```
Example: Tute 5, isr.c

#include " isr.h"
#include <salvo.h>

#pragma interrupt
void ISRHigh( void )
{
    if (INTCONbits.TMR0IE && INTCONbits.TMR0IF)
    {
        INTCONbits.TMR0IF = 0;
        TMR0 -= TMR0_RELOAD;

        OSTimer(); /* this call defines one system tick */
    }
}

#pragma code IntVectorHigh = 0x08
void IntVectorHigh( void )
{
    _asm goto ISRHigh _endasm
}
Example: Tute 6, main.c

```c
#include "main.h"
#include "ConfigRegs.h"    /* fails if included after salvo.h */
#include <salvo.h>

#define TASK_COUNT_P        OSTCBP(1)   /* task #1   */
#define TASK_SHOW_P         OSTCBP(2)   /* "" #2   */
#define TASK_BLINK_P        OSTCBP(3)   /* "" #3   */
#define PRIO_COUNT           12          /* task priorities */
#define PRIO_SHOW            10          /* "" */
#define PRIO_BLINK           2           /* "" */
#define MSG_UPDATE_PORT_P    OSECPBP(1)  /* semaphore #1 */

unsigned int counter;

char CODE_B = 'B';
char CODE_C = 'C';

_OSLabel( TaskCount1 )
_OSLabel( TaskShow1 )
_OSLabel( TaskBlink1 )
_OSLabel( TaskBlink2 )
```

Delaying a task: real-time operation. Real-time execution is enabled by generating a system tick from a hardware counter (Timer1 - see isr.c). A third task (TaskBlink) runs after a delay of 50 system ticks, therefore at a frequency that can be set accurately in real-time. **/
```c
void main( void )
{
    Init();
    OSInit();           /* initialise Salvo */

    /* Create and start three tasks */
    OSCreateTask( TaskCount, TASK_COUNT_P, PRIO_COUNT );
    OSCreateTask( TaskShow, TASK_SHOW_P, PRIO_SHOW );
    OSCreateTask( TaskBlink, TASK_BLINK_P, PRIO_BLINK );

    OSCreateMsg( MSG_UPDATE_PORT_P, (OStypeMsgP) 0 );

    OSEi();          /* enable interrupt response */
    for (;;)
    {
        OSSched();       /* run the scheduler */
    }
}
```
Example: Tute 6, main.c

```c
void TaskCount( void )
{
    counter = 0;
    for (;;) {
        counter++;
        /* Signal if the upper seven bits of counter change */
        if ( !(counter & 0x01FF) ) {
            OSSignalMsg( MSG_UPDATE_PORT_P, (OStypeMsgP) &CODE_C );
        }
        OS_Yield( TaskCount1 );
    }
}
```
Example: Tute 6, main.c

```c
void TaskShow( void )
{
    OStypeMsgP msgP;      /* note use of Salvo defined type */
    InitPORT();

    for (;;)
    {
        /* Wait on message MSG_UPDATE_PORT_P */
        OS_WaitMsg( MSG_UPDATE_PORT_P, &msgP, OSNO_TIMEOUT, TaskShow1 );

        /* Correctly extract the char pointed to by msgP and test it */
        if ( *(char *) msgP == CODE_C )
        {
            PORT = (PORT & ~0xFE) | ((counter >> 8) & 0xFE);
        }
        else
        {
            PORT ^= 0x01;
        }
    }
}
```
Example: Tute 6, main.c

```c
void TaskBlink( void )
{
    OStypeErr err;

    for (;;)
    {
        OS_Delay( 50, TaskBlink1 );

        err = OSSignalMsg( MSG_UPDATE_PORT_P, (OStypeMsgP) &CODE_B );

        if ( err == OSERR_EVENT_FULL )
        {
            OS_SetPrio( PRIO_SHOW + 1, TaskBlink2 ); /* decrease priority and context switch */
            OSSignalMsg( MSG_UPDATE_PORT_P, (OStypeMsgP) &CODE_B );
            OSSetPrio( PRIO_BLINK ); /* initial priority, no context switch */
        }
    }
}
```
Salvo Services

- Salvo functions are grouped here by functionality
- See Salvo User Manual ch. 7 for full details
Salvo System Services

- Salvo services for initialisation and timing

<table>
<thead>
<tr>
<th>Service</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>OSInit()</td>
<td>Prepare for Multitasking</td>
</tr>
<tr>
<td>OSSched()</td>
<td>Run the Highest-Priority Eligible Task</td>
</tr>
<tr>
<td>OS_Yield()</td>
<td>Context-switch (Unconditional)</td>
</tr>
<tr>
<td>OSGetTicks()</td>
<td>Return the current value of the System Timer</td>
</tr>
<tr>
<td>OSSetTicks()</td>
<td>Set the value of the System Timer</td>
</tr>
<tr>
<td>OSTimer()</td>
<td>Run the System Timer. Usually called from an ISR on timer overflow every 5ms to 100ms. Required if delays, elapsed time and/or timeout services are used.</td>
</tr>
</tbody>
</table>
### Task Management Services

- **Salvo services for task management**
  ("explicit" task functions)

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>OSCreateTask()</td>
<td>Create and Start a Task. Creating assigns a Task Control Block (TCB) to the task.</td>
</tr>
<tr>
<td>OSStartTask()</td>
<td>Make a Stopped Task Eligible To Run</td>
</tr>
<tr>
<td>OSSetPrioTask()</td>
<td>Change a Specified Task's Priority. Can’t change the priority of a task that is waiting an event.</td>
</tr>
<tr>
<td>OSStopTask()</td>
<td>Stop a Task. Can’t stop a task that is waiting an event.</td>
</tr>
<tr>
<td>OSDestroyTask()</td>
<td>Destroy a Task. Destroying releases the TCB for reuse. Can’t destroy a task that is waiting an event.</td>
</tr>
<tr>
<td>OSGetPrioTask()</td>
<td>Return the Specified Task's Priority</td>
</tr>
<tr>
<td>OSGetStateTask()</td>
<td>Return the Specified Task's State</td>
</tr>
</tbody>
</table>
**Task Management Services (ctd)**

- Services for managing the currently executing task ("implicit" task functions)

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>OSGetPrio()</td>
<td>Return the Current Task's Priority</td>
</tr>
<tr>
<td>OSSetPrio()</td>
<td>Change the Current Task's Priority. Takes effect after the next context switch.</td>
</tr>
<tr>
<td>OS_SetPrio()</td>
<td>Change the Current Task's Priority and Context-switch</td>
</tr>
<tr>
<td>OSGetTS()</td>
<td>Return the Current Task's Timestamp</td>
</tr>
<tr>
<td>OSSetTS()</td>
<td>Set the Current Task's Timestamp</td>
</tr>
<tr>
<td>OSSyncTS()</td>
<td>Synchronize the Current Task's Timestamp</td>
</tr>
<tr>
<td>OS_Delay()</td>
<td>Delay the Current Task and Context-switch</td>
</tr>
<tr>
<td>OS_DelayTS()</td>
<td>Delay the Current Task Relative to its Timestamp and Context-switch</td>
</tr>
<tr>
<td>OS_Replace()</td>
<td>Replace the Current Task with the one specified and Context-switch</td>
</tr>
<tr>
<td>OS_Stop()</td>
<td>Stop the Current Task and Context-switch</td>
</tr>
<tr>
<td>OS_Destroy()</td>
<td>Destroy the Current Task and Context-switch</td>
</tr>
</tbody>
</table>
Binary Semaphores

- Salvo services for managing binary semaphores

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>OSCreateBinSem()</td>
<td>Create a Binary Semaphore. Creating assigns an event control block (ECB) to the Binary Semaphore.</td>
</tr>
<tr>
<td>OSSignalBinSem()</td>
<td>Signal a Binary Semaphore</td>
</tr>
<tr>
<td>OSTryBinSem()</td>
<td>Obtain a Binary Semaphore if Available. Like OS_WaitBinSem(), but does not context switch if the binary semaphore is not available.</td>
</tr>
<tr>
<td>OSReadBinSem()</td>
<td>Read the value of a Binary Semaphore. There is no effect on the specified binary semaphore.</td>
</tr>
<tr>
<td>OS_WaitBinSem()</td>
<td>Context-switch and Wait the Current Task on a Binary Semaphore. A timeout can be specified.</td>
</tr>
</tbody>
</table>
(Counting) Semaphores

- Salvo services for managing counting semaphores

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>OSCreateSem()</td>
<td>Create a Semaphore. Creating assigns an event control block (ECB) to the Semaphore.</td>
</tr>
<tr>
<td>OSSignalSem()</td>
<td>Signal a Semaphore</td>
</tr>
<tr>
<td>OSTrySem()</td>
<td>Obtain a Semaphore if Available. Like OS_WaitSem(), but does not context switch if the semaphore is not available</td>
</tr>
<tr>
<td>OSReadSem()</td>
<td>Read the value of a Semaphore. There is no effect on the specified semaphore.</td>
</tr>
<tr>
<td>OS_WaitSem()</td>
<td>Context-switch and Wait the Current Task on a Semaphore. A timeout can be specified.</td>
</tr>
</tbody>
</table>
## Event Flag Services

- **Salvo services for managing event flags**

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>OSCreateEFlag()</code></td>
<td>Create an Event Flag. Creating assigns an event control block (ECB) to the Event Flag.</td>
</tr>
<tr>
<td><code>OSSetEFlag()</code></td>
<td>Set Event Flag Bit(s)</td>
</tr>
<tr>
<td><code>OSClrEFlag()</code></td>
<td>Clear Event Flag Bit(s). Use after successfully waiting an event flag.</td>
</tr>
<tr>
<td><code>OSReadEFlag()</code></td>
<td>Read the value of an Event Flag. There is no effect on the Event Flag.</td>
</tr>
<tr>
<td><code>OS_WaitEFlag()</code></td>
<td>Context-switch and Wait the Current Task on an Event Flag. A timeout can be specified.</td>
</tr>
</tbody>
</table>
Message Services

- Salvo services for managing messages

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>OSCreateMsg()</td>
<td>Create a Message. Creating assigns an event control block (ECB) to the Message.</td>
</tr>
<tr>
<td>OSSignalMsg()</td>
<td>Send a Message</td>
</tr>
<tr>
<td>OSTryMsg()</td>
<td>Obtain a Message if Available. Like OS_WaitMsg(), but does not context switch if the Message is not available.</td>
</tr>
<tr>
<td>OSReadMsg()</td>
<td>Read a Message's Message Pointer. No effect on the Message.</td>
</tr>
<tr>
<td>OS_WaitMsg()</td>
<td>Context-switch and Wait the Current Task on a Message. A timeout can be specified.</td>
</tr>
</tbody>
</table>
# Message Queue Services

- Salvo services for managing message queues

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>OSCreateMsgQ()</td>
<td>Create a Message Queue. Creating assigns an event control block (ECB) to the Message Queue.</td>
</tr>
<tr>
<td>OSMsgQCount()</td>
<td>Return Number of Messages in Message Queue</td>
</tr>
<tr>
<td>OSMsgQEmpty()</td>
<td>Check for Available Space in Message Queue</td>
</tr>
<tr>
<td>OSSignalMsgQ()</td>
<td>Send a Message via a Message Queue</td>
</tr>
<tr>
<td>OSTryMsgQ()</td>
<td>Obtain a Message from a Message Queue if Available. Like OS_WaitMsgQ(), but does not context switch if the Message is not available.</td>
</tr>
<tr>
<td>OSReadMsgQ()</td>
<td>Read a Message Queue's Message Pointer. No effect on the Message Queue.</td>
</tr>
<tr>
<td>OS_WaitMsgQ()</td>
<td>Context-switch and Wait the Current Task on a Message Queue. A timeout can be specified.</td>
</tr>
</tbody>
</table>
Cyclic Timer Services

- **Salvo services for managing cyclic timers**

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>OSCreateCycTmr()</td>
<td>Create a Cyclic Timer. Creating assigns an task control block (TCB) to the Cyclic Timer.</td>
</tr>
<tr>
<td>OSStartCycTmr()</td>
<td>Start a Cyclic Timer</td>
</tr>
<tr>
<td>OSStopCycTmr()</td>
<td>Stop a Cyclic Timer</td>
</tr>
<tr>
<td>OSResetCycTmr()</td>
<td>Reset a Cyclic Timer</td>
</tr>
<tr>
<td>OSSetCycTmrPeriod()</td>
<td>Set a Cyclic Timer's Period</td>
</tr>
<tr>
<td>OSDestroyCycTmr()</td>
<td>Destroy a Cyclic Timer. Releases the TCB for reuse.</td>
</tr>
</tbody>
</table>

- **Cyclic Timers are used to create recurring timers**
- **They are ordinary functions, *not* tasks.**
References