Embedded System-Operating System

- Use of an OS or Monitor can Aid in Implementation
  - Increased Cost and Licensing
  - Increases Memory Footprint
  - Allows for Easier Extensions/Modifications to ES Software
- If OS not Used:
  - Controlling ES Program Must be Loaded through an Event such as Assertion of RESET
  - eg. RESET Asserted, Reset Interrupt Vector Points to Control Program Entry Point which is “INIT” State of SM
  - Control Program SM has no “Halting State”

1

When to use an RTOS

- Typically used When ES has Several Concurrent Tasks
- Splitting up ES Software into Independent Parts can Simplify System Complexity
- Concurrency, Timing, and Synchronization can be Challenging (but doable)
- You might want to use an RTOS if:
  - ES Software more Natural as Implemented in Set of Tasks or Concurrent Activities
  - Need Different Activities to Occur at Different Times, and they Initiate Based on “Events” (not static sched.)
  - Need to Prioritize Tasks
  - Anticipate Adding New Tasks to ES in Future
  - Lots of Timing (RT) Involved

2

Keil RL-RTX

- Need to Include `rtl.h` Header File in C Program
- Provides Access to RTX Functions
- Can Create RT ES Without RTOS, but RTOS Provides Access to
  - I/O Allocation
  - Scheduling
  - Maintenance
  - Timing
- RTX Enables Flexible Scheduling of Resources Such as CPU and Memory
- Provides Methods to Communicate Between Tasks

3

RTX Interprocess Communication

- Event Flags
  - Primary Instrument for Task Communication
  - Each Task has 16 Flags Assigned to it
  - Task "Waits" for Flag Events to Execute
    - All Selected Flags (AND-connection)
    - Any One of Selected Flags (OR-connection)
- Event Flags Set by Other Tasks or by an ARM Interrupt
- Synchronize to External Event by Making an ARM Interrupt Set a Flag

4

The Dining Philosophers

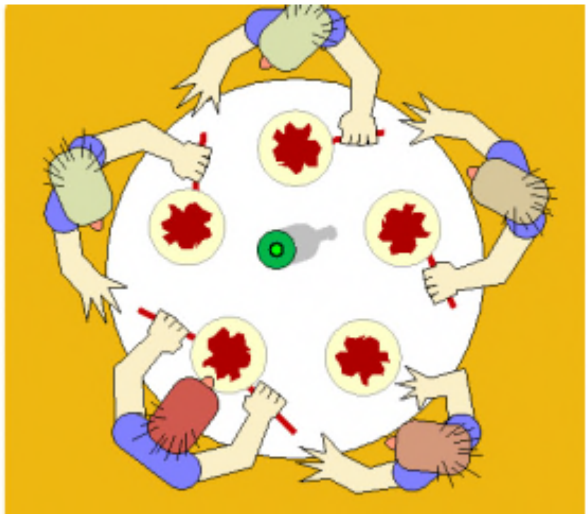
- Classic Problem in Task Synchronization
- Each Philosopher must Alternately Dine and Think (Task Processes data and Access I/O Device)
- Each Fork can Only be Held by One Philosopher and they Need Two of them to Eat
- The Philosopher can Grab a Fork if it is not Being Held by Another
- There is an Infinite Supply of Spaghetti
- The Problem is how to let all Philosophers think and eat Fairly-One Solution is to use Semaphores

5

Dining Philosophers

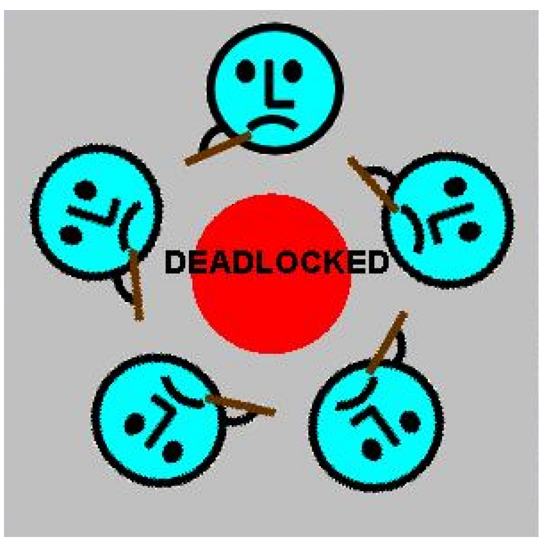
6

Dining Philosophers - Allocated



7

Dining Philosophers - Deadlocked



8

Semaphores

- Used When More than One Task Needs Access to a Single Common Resource
- eg, if 2 tasks assigned to process 2 different sensors and each task must output to common device, need a means to prevent both tasks from attempting to output to common device at same time
- Can Cause Unexpected Behavior or DEADLOCK
  - Dining Philosopher's Problem
- Binary Semaphores are Data Objects Containing a Virtual Token
- Details on Semaphores in OS Class (CSE 5343)

MUTEX Blocks

- Concept of "Mutual Exclusion" can be Used for Process Synchronization
- Keil RTX Provides MUTEX Block Services
- MUTEX is Software Object used by a Task to "Lock" a Common Resource
- OS Kernel Blocks all Tasks for using a Common Resource until Original Locking Task Releases it
- When Task Needs Resource, it Attempts to Acquire it and if Available it "Locks" Resource using a MUTEX
- Task Must Wait Until Resource is Available "Unlocked" to Acquire Control
  - can be tricky when there are Real-time Deadlines
  - uses concept of "time out" and task priorities

10

The "Talking Stick"

- aka "Speaker's Staff" an Instrument of Aboriginal Democracy
- Talking Stick Passed Around a Group as Symbol of Authority and Right to Speak
- Enables Everyone the Right to "Speak"
- Stick is Passed Around Group (Scheduling)
- Order of Passing it Around Indicates Priority
- Person Holding Stick May Choose to Give it to Someone Temporarily and They must Give it Back after they have Spoken
  - One Task Signals Another

11

Mailboxes

- Each Task can have a Mailbox to Receive Messages from other Tasks
- Message is Typically a Pointer to a Block of Memory containing a data frame
  - system designer has responsibility to allocate/deallocate the memory when task processes message (not RTX)
- RTX Kernel puts Waiting Task to Sleep if there is no Message
- RTX Kernel "wakes up" Task whenever it Receives a Mailbox Message from another Task

12

RL-ARM Technical Data

Description	ARM7™/ARM9™	Cortex™-M
Defined Tasks	Unlimited	Unlimited
Active Tasks	250 max	250 max
Mailboxes	Unlimited	Unlimited
Semaphores	Unlimited	Unlimited
Mutexes	Unlimited	Unlimited
Signals / Events	16 per task	16 per task
User Timers	Unlimited	Unlimited
Code Space	<4.2 Kbytes	<4.0 Kbytes
RAM Space for Kernel	300 bytes + 80 bytes User Stack	300 bytes + 128 bytes Main Stack
RAM Space for a Task	TaskStackSize + 52 bytes	TaskStackSize + 52 bytes
RAM Space for a Mailbox	MaxMessages * 4 + 16 bytes	MaxMessages * 4 + 16 bytes
RAM Space for a Semaphore	8 bytes	8 bytes
RAM Space for a Mutex	12 bytes	12 bytes
RAM Space for a User Timer	8 bytes	8 bytes
Hardware Requirements	One on-chip timer	SysTick timer
User task priorities	1 - 254	1 - 254
Task switch time	<5.3 µsec @ 60 MHz	<2.6 µsec @ 72 MHz
Interrupt lockout time	<2.7 µsec @ 60 MHz	Not disabled by RTX

13

RL-ARM Timing Data

Function	ARM7™/ARM9™ (cycles)	Cortex™-M (cycles)
Initialize system (os_sys_init), start task	1721	1147
Create task (no task switch)	679	403
Create task (switch task)	787	461
Delete task (os_tsk_delete)	402	218
Task switch (by os_tsk_delete_self)	458	230
Task switch (by os_tsk_pass)	321	192
Set event (no task switch)	128	89
Set event (switch task)	363	215
Send semaphore (no task switch)	106	72
Send semaphore (switch task)	364	217
Send message (no task switch)	218	117
Send message (switch task)	404	241
Get own task identifier (os_tsk_self)	23	65
Interrupt lockout	<160	0

14

Example RTX Application

- Taken from Folder:
  `\Keil\ARM\RL\RTX\Examples\RTX_ex1`
- ES Application Divided into Two Activities
  - Activity 1: Continuously Repeats every 50ms
  - Activity 2: Repeats 20ms after Activity 1 completes
- Each Activity Task Processing is in Separate C Function uses **`__task`** Defined in **`RTL.H`**

```
__task void task1 (void) {
  // .... place code of task 1 here ....
}

__task void task2 (void) {
  // .... place code of task 2 here ....
}
```

15

Example RTX Application (cont)

- Main Function Must Invoke the RTX Kernel Initially
  `os_sys_init`
- Need to Pass Task Function Name to Kernel as Argument of **`os_sys_init`**
  - This Starts the Execution of the Task
- In Example, Initialize **`task1`** and then **`task1`** Initializes **`task2`** using
  `os_task_create`

```
void main (void) {
  os_sys_init (task1);
}
__task void task1 (void) {
  os_tsk_create (task2, 0);
 //  .... place code of task 1 here ....
}
```

16

Implement Timing

- Code for Each Task is in Form of Infinite Loop
- When **task1** Finishes, it Sends a Signal to **task2** and Waits (**os_dly_wait**) for it to Complete
- RTX Kernel uses on-chip HW Timer and Programs it Directly based on **os_dly_wait** Arguments
  - Default is Timer 0 with Each Time Interval=10ms
  - Can Configure to use Different Timers and Intervals
- Can use **os_evt_wait_or** to Make **task1** Wait for **task2** to Complete
- Can use **os_evt_set** to Send Signal (Event) to **task2**
  - example uses bit 2 (position 3) of Event Flags

17

Example Code

```
/* Include type and function declarations for RTX. */
#include <rtl.h>

/* id1, id2 will contain task identifications at run-time. */
OS_TID id1, id2;

/* Forward declaration of tasks. */
__task void task1 (void);
__task void task2 (void);

void main (void) {
  /* Start the RTX kernel, and then create and execute task1. */
  os_sys_init(task1);
}
```

18

Example Code

```
__task void task1 (void){
  /* Obtain own system task identification number. */
  id1 = os_tsk_self();

  /* Create task2 and obtain its task identification number. */
  id2 = os_tsk_create (task2, 0);

  for (;;) {    //infinite loop
    /* ... place code for task1 activity here ... */

    /* Signal to task2 that task1 has completed. */
    os_evt_set(0x0004, id2);

    /* Wait for completion of task2 activity. */
    /*  0xFFFF makes it wait without timeout. */
    /*  0x0004 represents bit 2. */
    os_evt_wait_or(0x0004, 0xFFFF);

    /* Wait for 50 ms before restarting task1 activity. */
    os_dly_wait(5);
  }
}
```

19

Example Code

```
__task void task2 (void) {
  for (;;) {    //infinite loop
    /* Wait for completion of task1 activity. */
    /*  0xFFFF makes it wait without timeout. */
    /*  0x0004 represents bit 2. */
    os_evt_wait_or(0x0004, 0xFFFF);

    /* Wait for 20 ms before starting task2 activity. */
    os_dly_wait(2);

    /* ... place code for task2 activity here ... */

    /* Signal to task1 that task2 has completed. */
    os_evt_set(0x0004, id1);
  }
}
```

20

Using Keil MDK

- To Compile and Link with RTX
  - select RTX operating system for the Project Project →Options for Target
  - Select Target tab
  - Select RTX Kernel for Operating System
  - Build Project to Generate absolute File
- Can Run Project (object file output)
  - on the Target (the ARM board)
  - on the µVision Simulator

21

RTX Functions (9 Classes)

- Event Flag Management
- Mailbox Management
- Memory Allocation Functions
- Mutex Management
- Semaphore Management
- System Functions
- Task Management
- Time Management
- User Timer Management

22

RTX Functions (9 Classes)

- Event Flag Management
- Mailbox Management
- Memory Allocation Functions
- Mutex Management
- Semaphore Management
- System Functions
- Task Management
- Time Management
- User Timer Management

23

RTX Functions (9 Classes)

- Event Flag Management
- Mailbox Management
- Memory Allocation Functions
- Mutex Management
- Semaphore Management
- System Functions
- Task Management
- Time Management
- User Timer Management

24

Lab 6 RTX Functions

- **`os_tsk_create`** creates/starts new task
- **`os_dly_wait`** pauses calling task
- **`os_evt_set`** sets an event flag
- **`os_evt_wait_and`** waits for event flags to be set
- **`os_mut_init`** initializes a MUTEX object
- **`os_mut_release`** releases a MUTEX object
- **`os_mut_wait`** waits for MUTEX object to become available

25

os_mut_init

- Initializes a MUTEX Object Specified by Funciton Argument
- MUTEX Object is of Type **`OS_MUT`**

```
#include <rtl.h>
void os_mut_init (
    OS_ID mutex);      /* The MUTEX to initialize  */
```

- Type **`OS_ID`** Identifies an Object (defined in **`rtl.h`**)

```
typedef void *OS_ID; // System calls returning an
                     // object identification
```

- Example:

```
#include <rtl.h>
void os_mut_init  (
     OS_ID mutex );   /* The mutex to initialize */
```

26

`os_mut_init` Example

- Example Code for Initializing a MUTEX Block

```
#include <rtl.h>

OS_MUT mutex1;

__task void task1 (void) {
    ..
  os_mut_init (&mutex1);
    ..
}
```


`os_mut_release`

- This Function Decrements Internal MUTEX Counter Specified by Function Argument
- When Internal Counter Value Reaches Value of Zero, MUTEX is Free to be Acquired by Another Task
- MUTEX Object "knows" Which Task has it Currently Locked
- Owning Task can Acquire/Lock MUTEX as Needed through Call to **`os_mut_wait`**
- If Task that Owns MUTEX Tries to Acquire it Again, the Internal Counter is Incremented

`os_mut_release` (cont)

- Task that Owns MUTEX must Release it Same Number of Times that it was Acquired
  - in order to decrement internal count to zero
- Interacts with Task Priority if Priority Inheritance Feature is Used
- Function Returns a Value (One of):

OS_R_OK	MUTEX Successfully Released
OS_R_NOK	Error Occurred Because MUTEX Value is Already Zero or Calling Task is not Current MUTEX Owner

`os_mut_release` Example

```
#include <rtl.h>

OS_MUT mutex1;
void f1 (void) {
  os_mut_wait (&mutex1, 0xffff);
   ..
  /* Critical region 1 */
   ..
  /* f2() will not block the task. */
  f2 ();
  os_mut_release (&mutex1);
}

void f2 (void) {
  os_mut_wait (&mutex1, 0xffff);
   ..
  /* Critical region 2 */
   ..
  os_mut_release (&mutex1);
}
```

os_mut_release Example (cont)

```
__task void task1 (void) {
    ..
  os_mut_init (&mutex1);
  f1 ();
    ..
}

__task void task2 (void) {
    ..
  f2 ();
    ..
}
```

31

os_mut_wait

- This Function Attempts to Acquire MUTEX Specified by Function Argument
- If MUTEX not Locked, Calling Task Acquires and Locks Mutex
- If MUTEX Locked, RTX Kernel puts Calling Task to Sleep Until
  - MUTEX Becomes Unlocked OR
  - A **`timeout`** Value is Exceeded
- Function Temporarily Raises Priority of Task Owning MUTEX if Lower than Priority of Calling Task
  - This is priority inheritance

32

os_mut_wait timeout Values

- **timeout** Argument has a Value [**0x0**, **0xffff**]
  - **0x0** Value Allows Calling Task to Acquire MUTEX Even if Higher Priority Task in the Ready List
  - **0xffff** Indicates timeout Value is Infinite (dangerous)
  - **0x1** through **0xfffe** Assign a Finite Value to timeout which causes task to Release MUTEX upon Expiration
- **timeout** Measured in Units of System Intervals
  - default value is 10ms

33

os_mut_wait (cont)

- Function Returns a Value (One of):

OS_R_MUT	MUTEX Successfully Acquired & Locked
OS_R_TMO	timeout has Expired
OS_R_OK	MUTEX was Available and Function Returned to Calling Task Immediately

34

`os_mut_wait` Example

```
#include <rtl.h>

OS_MUT mutex1;

void f1 (void) {
  os_mut_wait (&mutex1, 0xffff);
   ..
  /* Critical region 1 */
   ..
  /* f2() will not block the task. */
  f2 ();
  os_mut_release (&mutex1);
}
```

35

`os_mut_wait` Example

```
void f2 (void) {
  os_mut_wait (&mutex1, 0xffff);
   ..
  /* Critical region 2 */
   ..
  os_mut_release (&mutex1);
}

__task void task1 (void) {
   ..
  os_mut_init (&mutex1);
  f1 ();
   ..
}

__task void task2 (void) {
   ..
  f2 ();
   ..
}
```

36