RTOS: Tasks, Task data and Reentrant functions

Reference: Simon Chapter 6

Real-time operating systems (RTOS)

- Most real-time operating systems are different from desktop machine operating systems.
- On a desktop computer the operating system takes control of the machine as soon as it is turned on and then lets you start your applications.
- You compile and link your applications separately from the operating system.
- In an embedded system, you usually link your application and the RTOS. At boot-up time, your application usually gets control first, and it then starts the RTOS.

Embedded RTOS

- The application and the RTOS are tightly tied in an embedded system.
- Many RTOSs do not protect themselves as carefully from your application as do desktop operating systems.
  - For example:
    - Desktop computers usually check that any pointer you pass into a function is valid.
    - Embedded RTOS usually skip this for better performance.
  - Why? A bad pointer will crash the desktop application, but not the entire OS. In a embedded system, a bad pointer will crash the entire embedded system and its single application.

Customizing your RTOS

- To save memory RTOSs typically include just the services that you need for your embedded system and no more.
- Most RTOS allow you to configure them extensively so you don't link unnecessary parts to your code.
- Unless you need them, you can configure away things such as I/O drivers or even memory management.
- You don't usually write your own RTOS but buy it from a vendor such as VxWorks, VRTX, pSOS, Nucleus, C Executive, LynxOS, QNX, Multi-Task!, AMX, and dozens more.

The different RTOSs

- Unless your requirements for speed or code size or robustness are extreme, the commercial RTOSs represent a good value, in that they come already debugged and with a good collection of features and tools.

- In many ways the systems are very similar to one another: they offer most or all of the services discussed we will be discussing over the next couple of lectures.

Tasks and task states

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

Definition of a task

- **Task**: it's the basic building block of software written under an RTOS.
- Tasks are very simple: in most RTOS a task is just a sub-routine.
- At some point in your program, you call a function in the RTOS that starts a task; you tell which subroutine is the starting point for each task and some other parameters.
- There are no limit in the number of tasks in most RTOS.

Each task in an RTOS is always in one of three states

- **Running:** which means that the microprocessor is executing the instructions that make up this task.
- **Ready:** which means that some other task is in the running state but that this task has things that it could do if the microprocessor becomes available. Any number of tasks can be in this state.
- **Blocked:** which means that this task hasn't got anything to do right now, even if the microprocessor becomes available. Tasks get into this state because they are waiting for some external event. Any number of tasks can be in this state as well.

Handful of other states

- Most RTOSs may offer many other other task states. Such as **suspended**, **pended**, **waiting**, **dormant**, and **delayed**.
- We are just going to focus on the **running**, **ready** and **blocked** states.

The scheduler

What is a scheduler

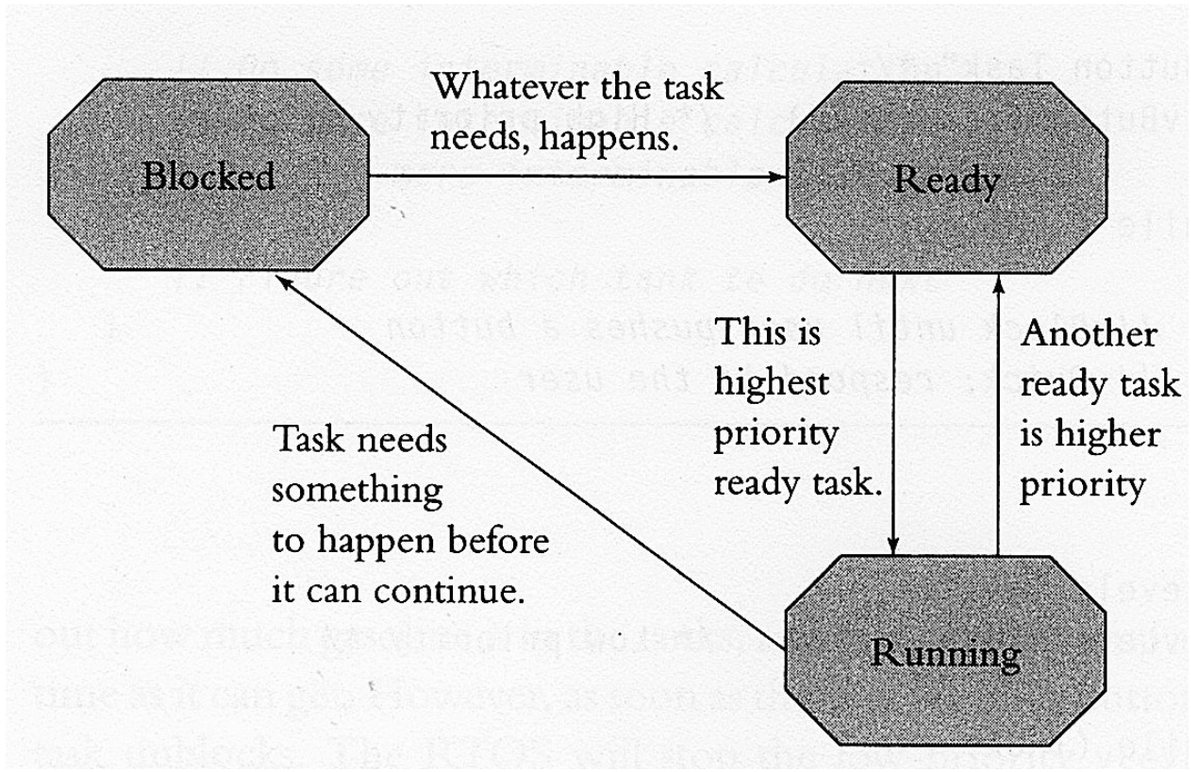
- The **scheduler**: A part of the RTOS which keeps track of the state of each task and decides which **one** task should go into the running state.

- **Very simple behavior**: schedulers look at priorities you assign to the tasks, and among the tasks that are not in the blocked state, the one with the **highest priority** runs, and the rest of them wait in the ready state.

- The **lower-priority** tasks just have to wait; the scheduler assumes that you knew what you were doing when you set the task priorities.

Task states

A piece of code (task) can be in any of the following states...

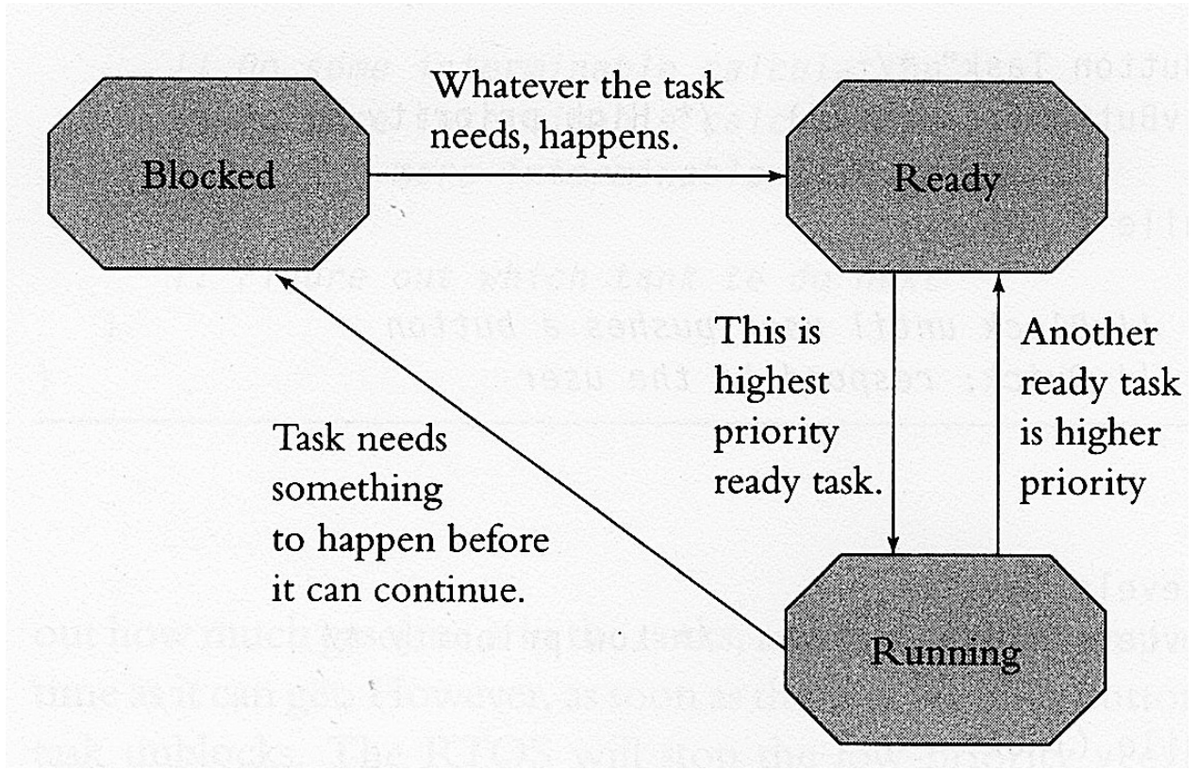


- **Block** means "move into the blocked state,"
- **Run** means "move into the running state“ or "be in the running state,"
- **Switch** means "change which task is in the running state."

In single core systems: only one task can be running!

Consequence #1: Only a task can block itself

A piece of code (task) can be in any of the following states...

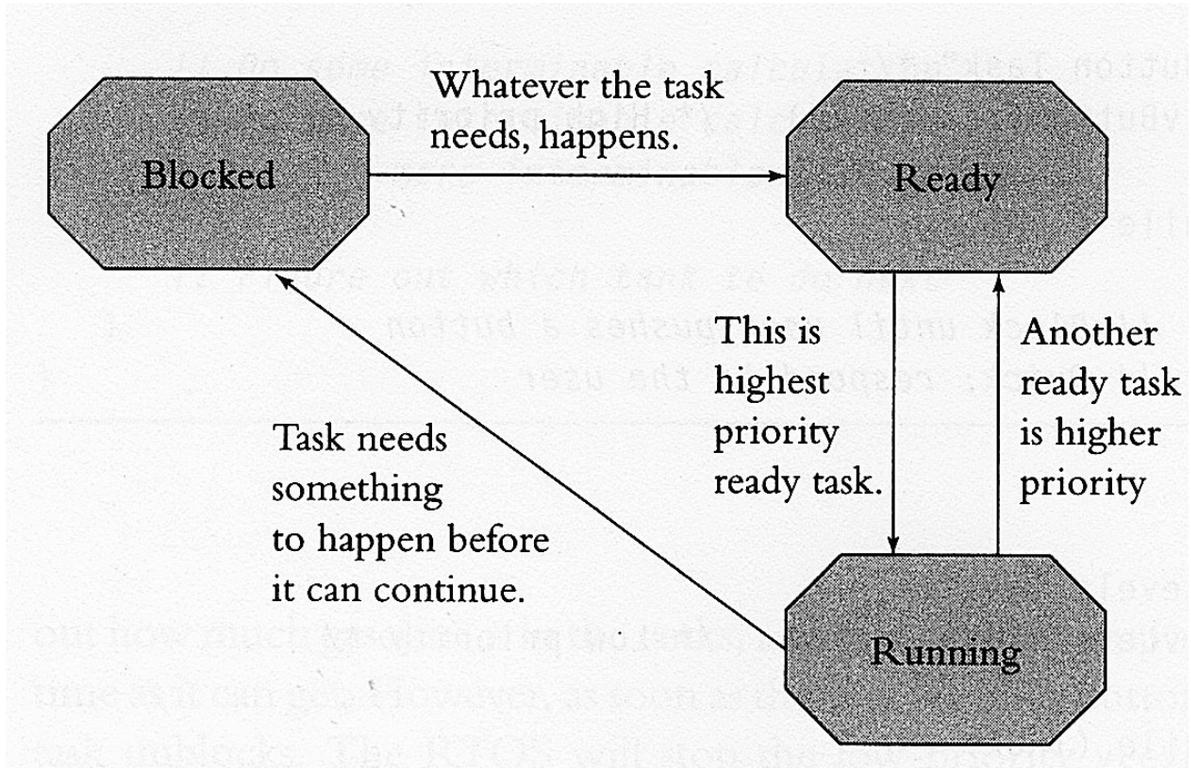


In single core systems: only one task can be running!

- A task will only switch to the block state when it decides for **itself** that it has run out of things to do (or that is waiting for something else).
- Other tasks **cannot** decide that a should go to the block state.
- A task has to be **running** before going to the block state.

Consequence #2: Only other tasks or interrupts can switch a task to ready!

A piece of code (task) can be in any of the following states...

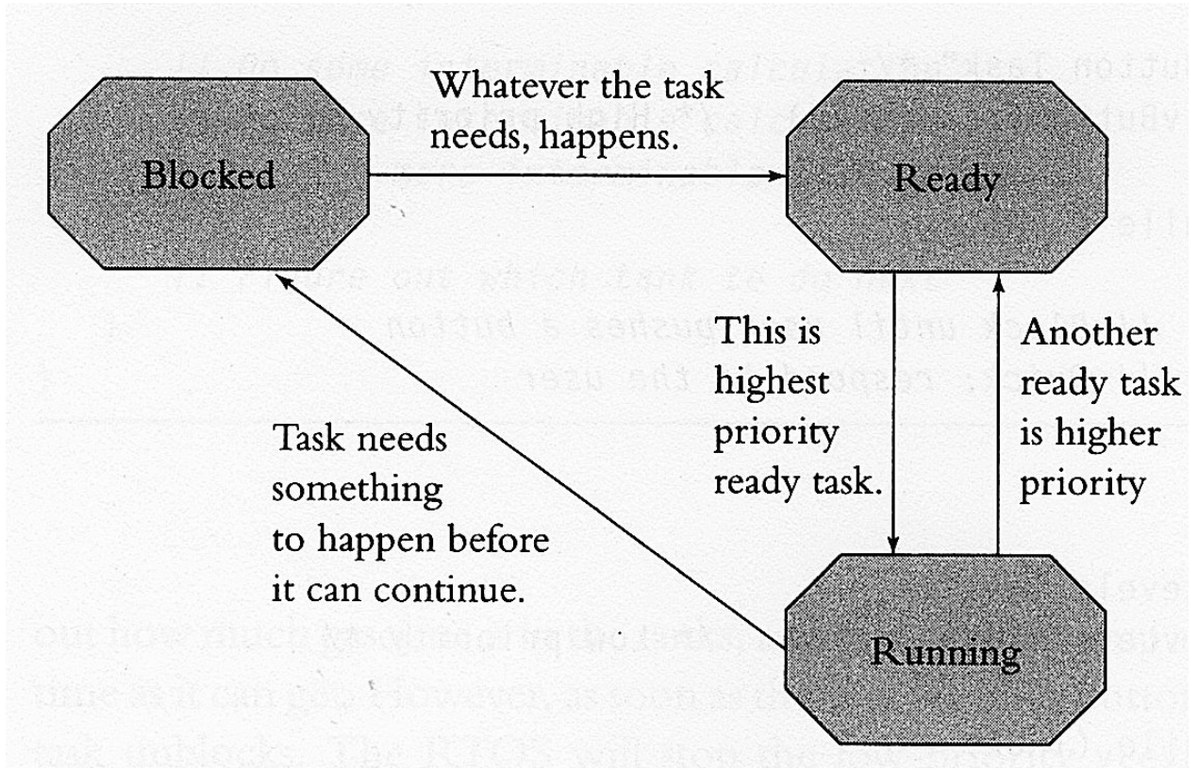


In single core systems: only one task can be running!

- While a task is in the blocked state, it **never** gets the microprocessor.
- So, an interrupt routine or some other task in the system must be able to signal that whatever the task was waiting for has happened.
- Otherwise, the task will be blocked forever.

Consequence #3: The scheduler controls which tasks to run next

A piece of code (task) can be in any of the following states...



In single core systems: only one task can be running!

- The shuffling of tasks between the ready and running states is entirely the work of the **scheduler**.
- Tasks can block themselves, and tasks and interrupt routines can move other tasks from the blocked state to the ready state, but the scheduler has control over the **running state**.

Some common questions about the scheduler and task states

How does the scheduler knows the different task states?

- Q: How does the scheduler know when a task has become blocked or unblocked?

- A: The RTOS provides a collection of functions that tasks can call to tell the scheduler what events they want to wait for and to signal that events have happened.

What happens if all the tasks are blocked?

- If all the tasks are blocked, then **the scheduler will spin in some tight loop** somewhere inside of the RTOS, waiting for something to happen.
- If nothing ever happens, then that's your fault!!!
- You must make sure that something happens sooner or later by having an interrupt routine that calls some RTOS function that unblocks a task. Otherwise, your software will not be doing very much.

What if two tasks with the same priority are ready?

- Q: What if two tasks with the same priority are ready?
- A: Depends on which RTOS you use.

1. Some RTOS make it illegal to have two tasks with the same priority.
2. Other RTOS will time-slice between two such tasks.
3. A ROTS can also run one random task until it blocks and then run the other.

Selecting which tasks to run

- Q: If one task is running and another, higher-priority task unblocks, does the task that is running get stopped and moved to the ready state right away?

- A: Depends on the RTOS. A **preemptive** RTOS will stop a lower-priority task as soon as the higher-priority task unblocks. A **nonpreemptive** RTOS will only take the microprocessor away from the lower-priority task when that task blocks.

Example

Tank monitoring system

- Imagine I have 300 watering tanks in my greenhouse

- I have one microprocessor that constantly monitors the water level at every single tank.

- It takes a while to determine the water level for each tank.

- Whenever I press a button I want to immediately know the water level for a particular tank.

Pseudo-code for a tank monitoring system

```
/* "Button Task" */
void vButtonTask (void)   /* High priority */
{
  while (TRUE)
    {
      !! Block until  user pushes a button
      !! Quick:  respond to the user
    }
}

/* "Levels Task" */
void vLevelsTask (void)   /* Low priority */
{
while (TRUE) {
     !!  Read  levels of floats  in tank
     !!  Calculate average float  level
     !!  Do some interminable calculation
     !!  Do more  interminable calculation
     !!  Do yet more  interminable calculation
     !!  Figure out which tank to do next
  }
}
```

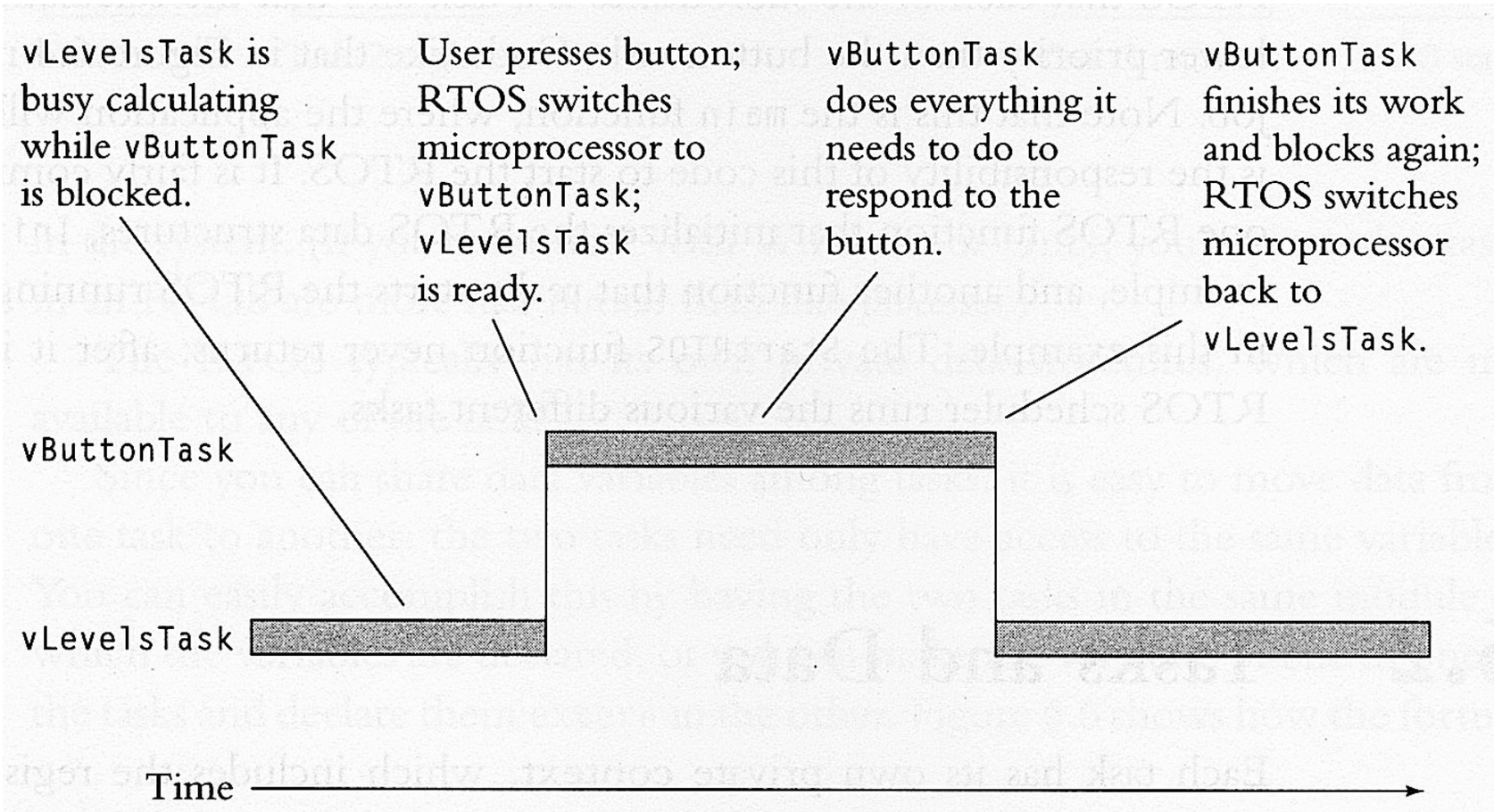
- vLevelsTask task uses up a lot of computing time figuring out how much water is in the tanks
- Low-priority: vLevelsTask task
- High-priority: vButtonTask task
- One convenient feature of the RTOS is that the two tasks can be written independently of one another, and the system will still respond well.

Microprocessor responds to a button under an RTOS

Blocked: task is waiting for other things to do.

Ready: task is ready to do useful stuff.

Running: tasks is currently executing.



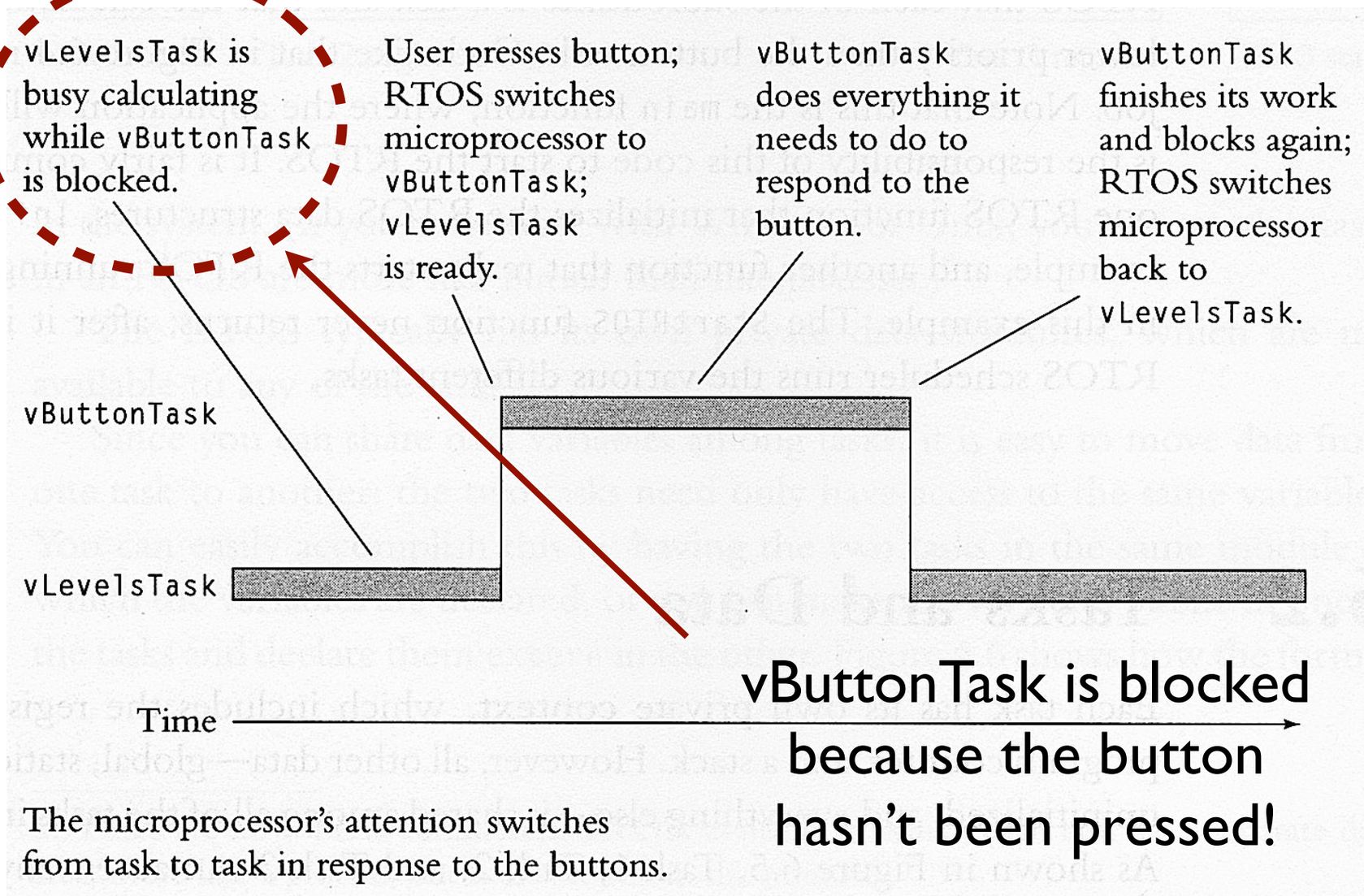
The microprocessor's attention switches from task to task in response to the buttons.

Microprocessor responds to a button under an RTOS

Blocked: task is waiting for other things to do.

Ready: task is ready to do useful stuff.

Running: tasks is currently executing.



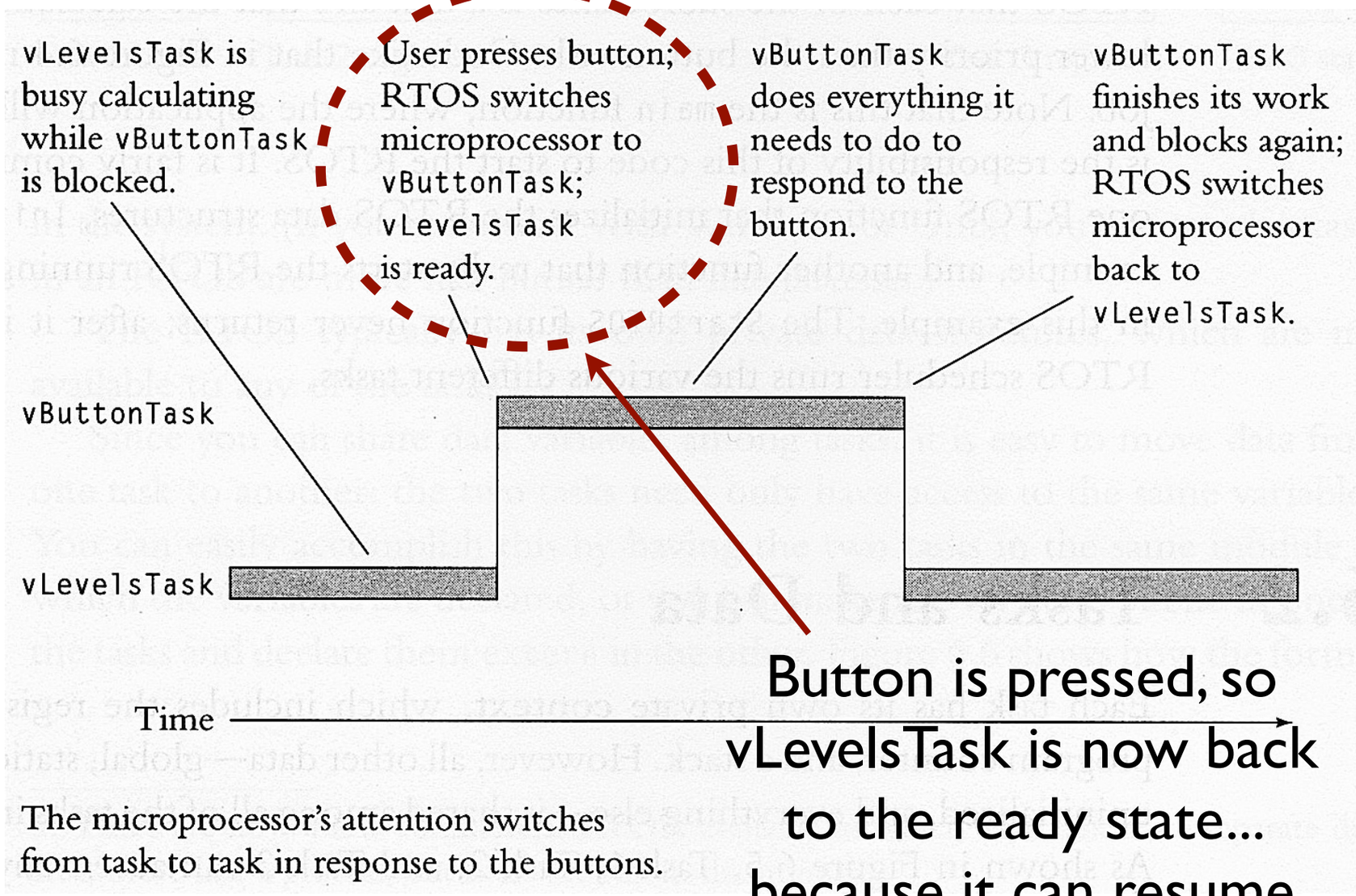
The microprocessor's attention switches from task to task in response to the buttons.

Microprocessor responds to a button under an RTOS

Blocked: task is waiting for other things to do.

Ready: task is ready to do useful stuff.

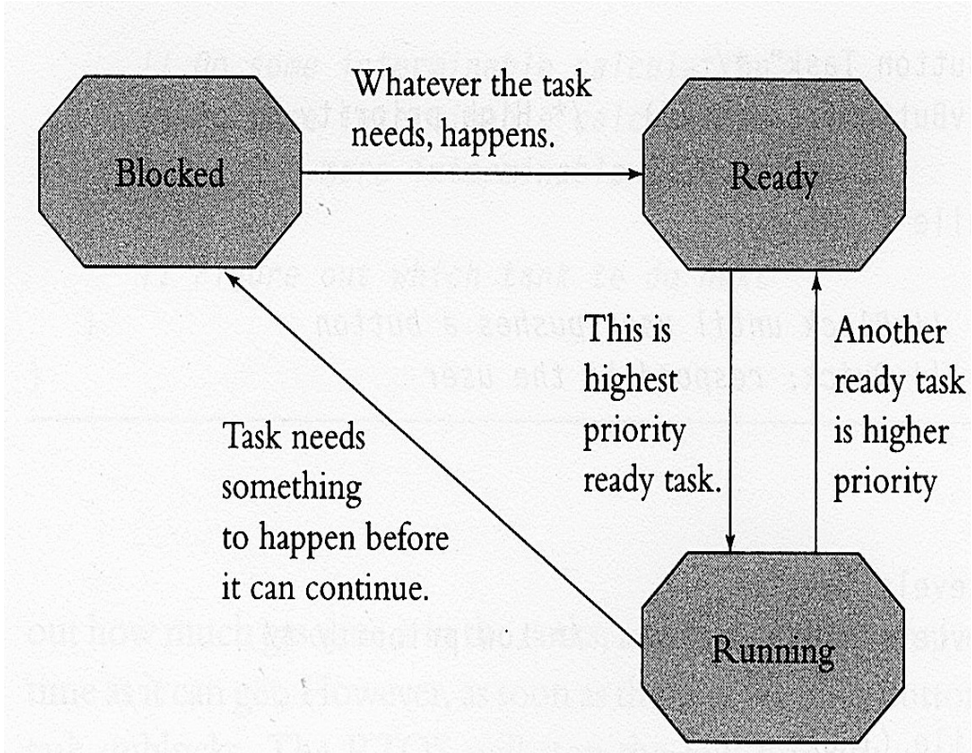
Running: tasks is currently executing.

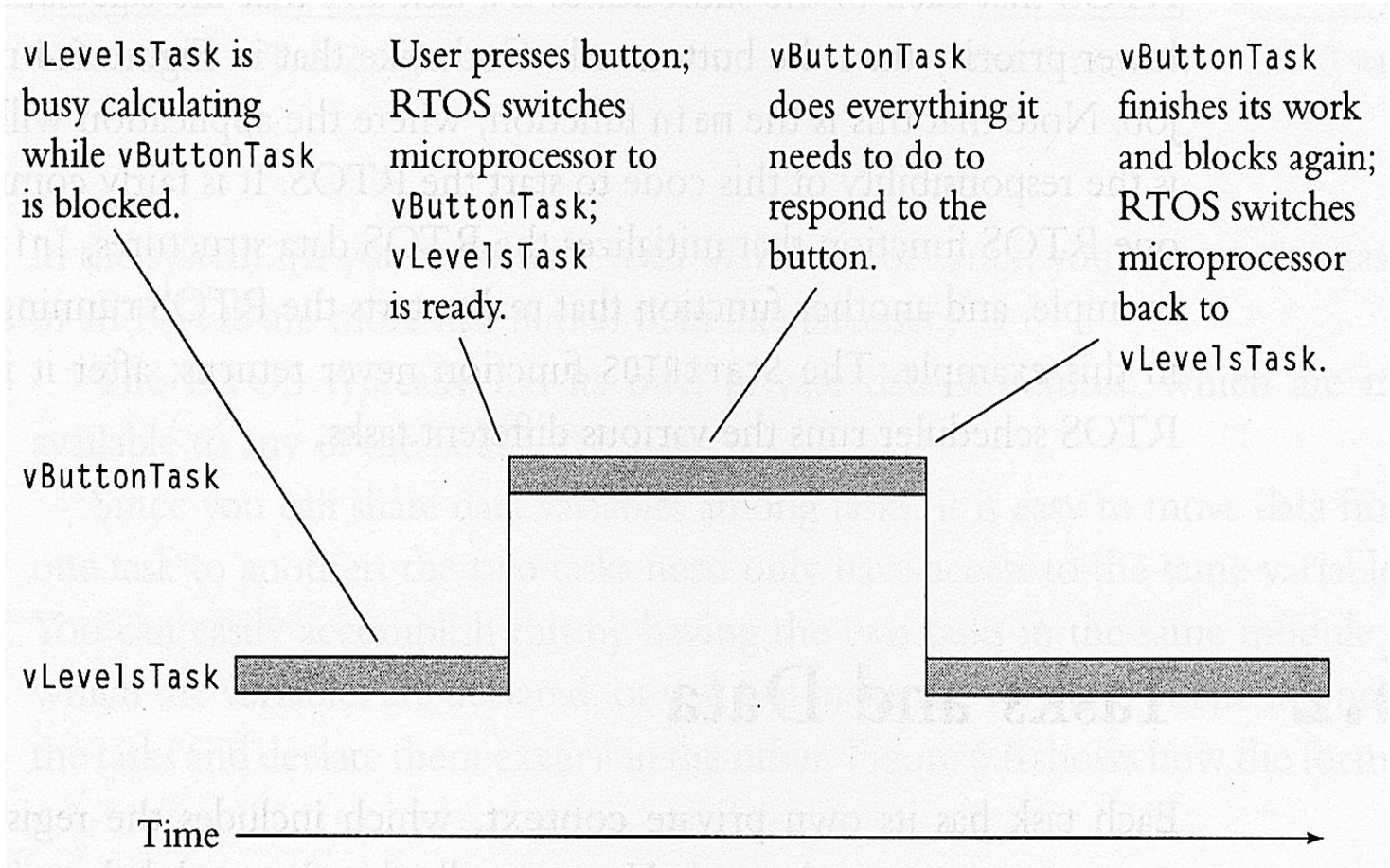


The microprocessor's attention switches from task to task in response to the buttons.

Button is pressed, so vLevelsTask is now back to the ready state... because it can resume operation anytime.

The three different states





The microprocessor's attention switches from task to task in response to the buttons.

RTOS initialization code

```
/* "Button Task" */
void vButtonTask (void)  /* High priority */
{
  while (TRUE)
    {
      !! Block until  user pushes a button
      !! Quick:  respond to the user
    }
}

/* "Levels Task" */
void vLevelsTask (void)  /* Low priority */
{
while (TRUE) {
     !!Read  levels of floats  in tank
     !!Calculate average float  level
     !!Do some interminable calculation
     !!Do more  interminable calculation
     !!Do yet more  interminable calculation
     !!Figure out which tank to do next
  }
}
```

```
void main  (void)
{
 //Initialize (but do not start) the RTOS
 InitRTOS ();

 // Tell  the RTOS about our tasks
 StartTask  (vRespondToButton,  HIGH_ PRIOTY);
 StartTask  (vCalculateTankLevels, L0W_PRIOTY);

 //Start the RTOS.
 //(This function never returns.)
 StartRTOS  ();
}
```

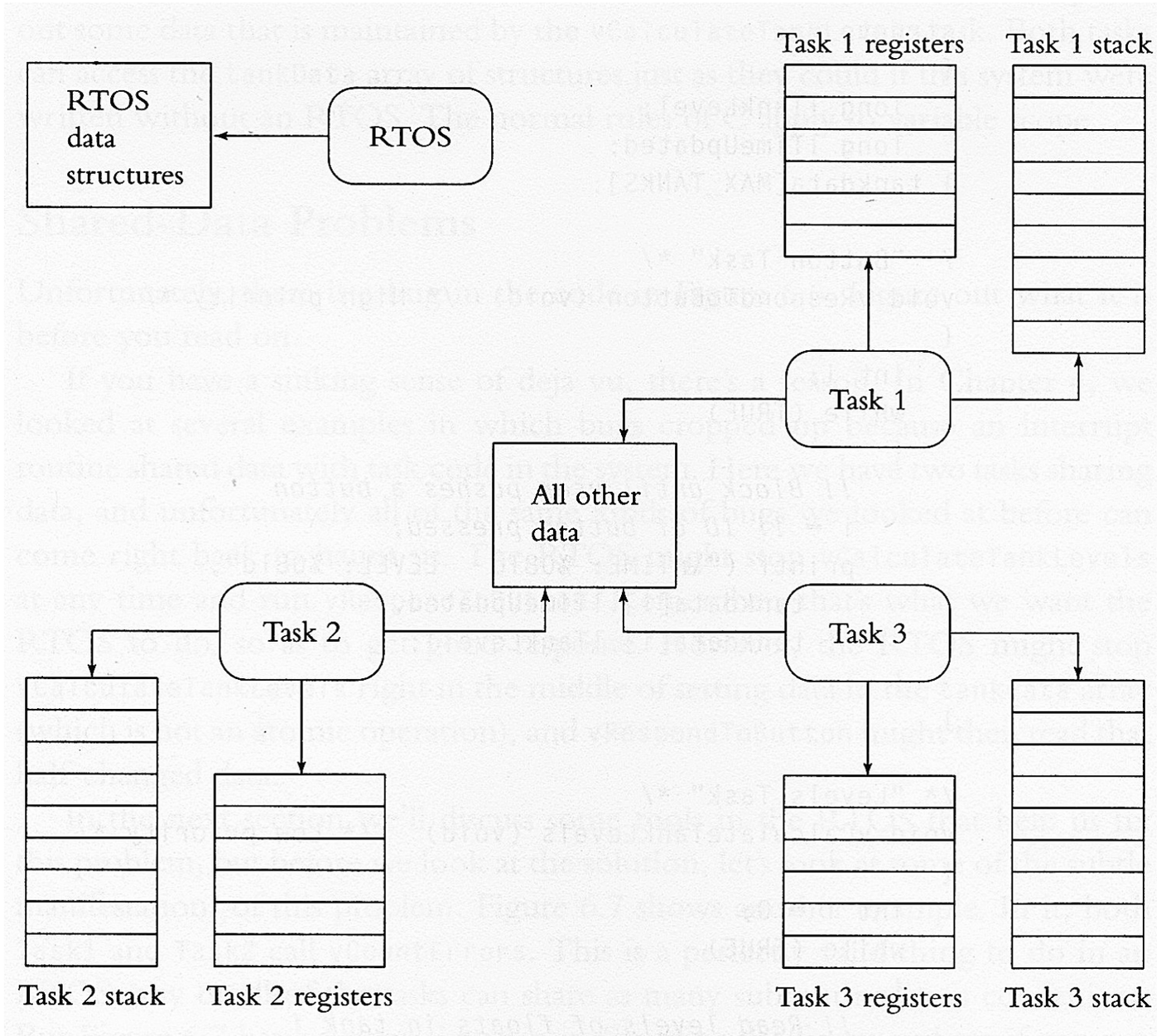
Tasks and data

Tasks and data

- **Each task has its own private context**, which includes the register values, a program counter, and a stack.
- All other data is shared among all of the tasks in the system (e.g global, static, initialized, uninitialized, and other data).
- The RTOS typically has its own private data structures, which are not available to any of the tasks.

Example

- Task 1, Task 2, and Task 3 can access any of the data in the system.
- Each task has its own private context, which includes the register values, a program counter, and a stack.
- The RTOS has some private data structures, which are not available to any of the tasks.



Sharing data variables between tasks

- Since you can share data variables among tasks, it is easy to move data from one task to another: the two tasks need only have access to the same variables.

- You can easily accomplish this by having the two tasks in the same module in which the variables are declared, or you can make the variables public in one of the tasks and declare them extern in the other.

- vRespondToButton task prints out some data that is maintained by the vCalculateTankLevels task.
- Both tasks can access the tankData array of structures just as they could if this system were written without an RTOS.

```
struct {
long ITankLevel;
long lTimellpdated;
}  tankdata[MAX_TANKS];

//"Button Task"
//High priority
void vRespondToButton (void)
{
  int i;
  while (TRUE)
  {
    !!Block until  user pushes a button
    i = !!ID of button pressed;
    printf ("\nTIME: %081d  LEVEL: %081d",
    tankdata[i].ITimeUpdated,
    tankdata[i].ITankLevel);
  }
}
```

```
//"Levels Task"
//Low priority
void vCalculateTankLevels (void)
{
  int i =0;
  while (TRUE)
  {
    !!Read  levels of floats in tank i
    !!Do more  interminable calculation
    !!Do yet more  interminable calculation

    //Store the result
    tankdata[i].ITimeUpdated = !!Current time

    //Between these two instructions is a
    //bad place for a task switch
    tankdata[i].ITankLevel = !!Calc. Result

    !!Figure out which tank to do next
    i = !!something new
  }
}
```

Shared-data problems

- Bugs may show up because an interrupt routine shares data with task code in the system.
- Two tasks are sharing data, and unfortunately all of the same kinds of bugs we previously discussed will show up again.
- The RTOS might stop vCalculateTankLevels at any time and run vRespondToButton.
- However, the RTOS might stop vCalculateTankLevels right in the middle of setting data in the tankdata array (which is not an atomic operation), and vRespondToButton might then read that half-changed data.

Another example

Tasks can share code

```
static  int cErrors;

void Task1   (void)
{
   (...)
   vCountErrors(9);
   (...)
}

void Task2   (void)
{
  (...)
  vCountErrors(11);
  (...)
}

void vCountErrors   (int cNewErrors)
{
   cErrors += cNewErrors;
}
```

- In this example, both Task1 and Task2 call function vCountErrors.
- This is a perfectly valid thing to do in an RTOS
- Any or all of the tasks can share as many functions as it is convenient.

Potential bug on this example

```
static  int cErrors;

void Task1   (void)
{
   (...)
   vCountErrors(9);
   (...)
}

void Task2   (void)
{
  (...)
  vCountErrors(11);
  (...)
}

void vCountErrors   (int cNewErrors)
{
   cErrors += cNewErrors;
}
```

- Task1 and Task2 call vCountErrors, and since vCountErrors uses the variable cErrors, the variable cErrors is now shared by the two tasks.
- If Task1 calls vCountErrors, and if the RTOS then stops Task1 and runs Task2, which then calls vCountErrors, the variable cErrors may get corrupted in just the same way as it would if Task2 were an interrupt routine that had interrupted Task1.

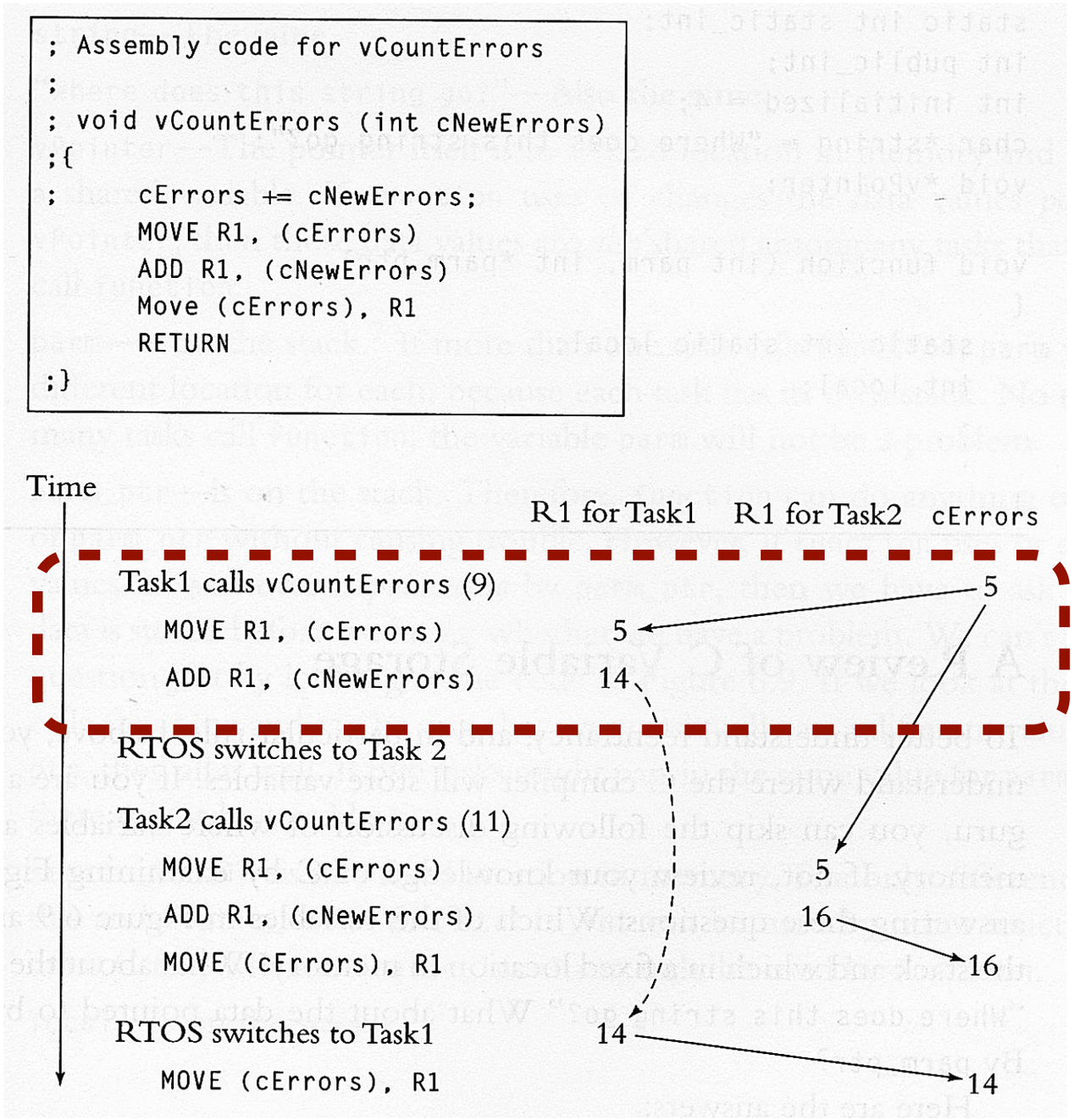
How can cErrors become corrupted? (Part 1)

```
; Assembly code for vCountErrors
;
; void vCountErrors (int cNewErrors)
;{
;       cErrors += cNewErrors;
        MOVE R1, (cErrors)
        ADD R1, (cNewErrors)
        Move (cErrors), R1
        RETURN
;}
```

- cErrors = cErrors + cNewErrors;
- Move cErrors into register R1
- Add cNewErrors to register R1 and store the result in R1
- Move the contents of R1 into the memory location where cErrors is.

How can cErrors become corrupted? (Part 2)

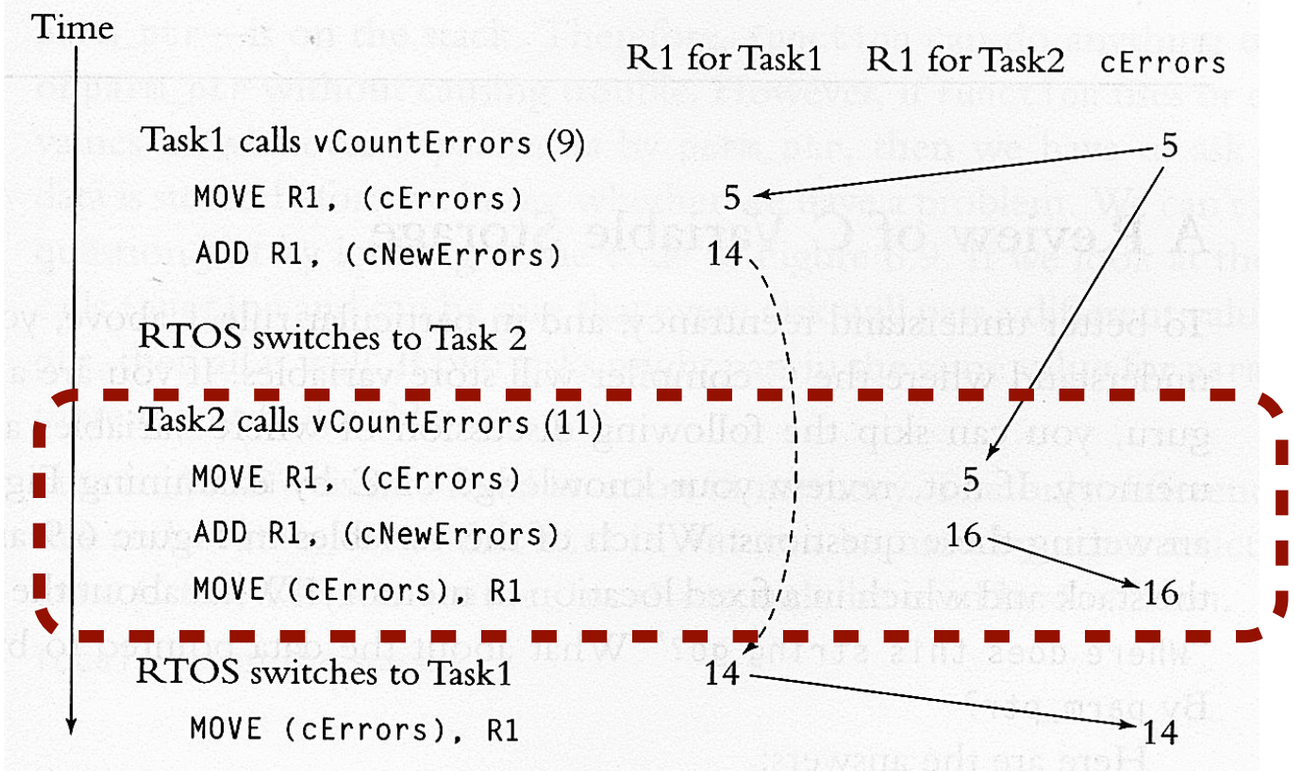
```
; Assembly code for vCountErrors
;
; void vCountErrors (int cNewErrors)
;{
;       cErrors += cNewErrors;
        MOVE R1, (cErrors)
        ADD R1, (cNewErrors)
        Move (cErrors), R1
        RETURN
;}
```



- Suppose that the value 5 is initially stored in cErrors.
- Suppose that Task1 calls vCountErrors(9), and suppose that vCountErrors does the MOVE and ADD instructions, leaving the result in register R1(14).

How can cErrors become corrupted? (Part 3)

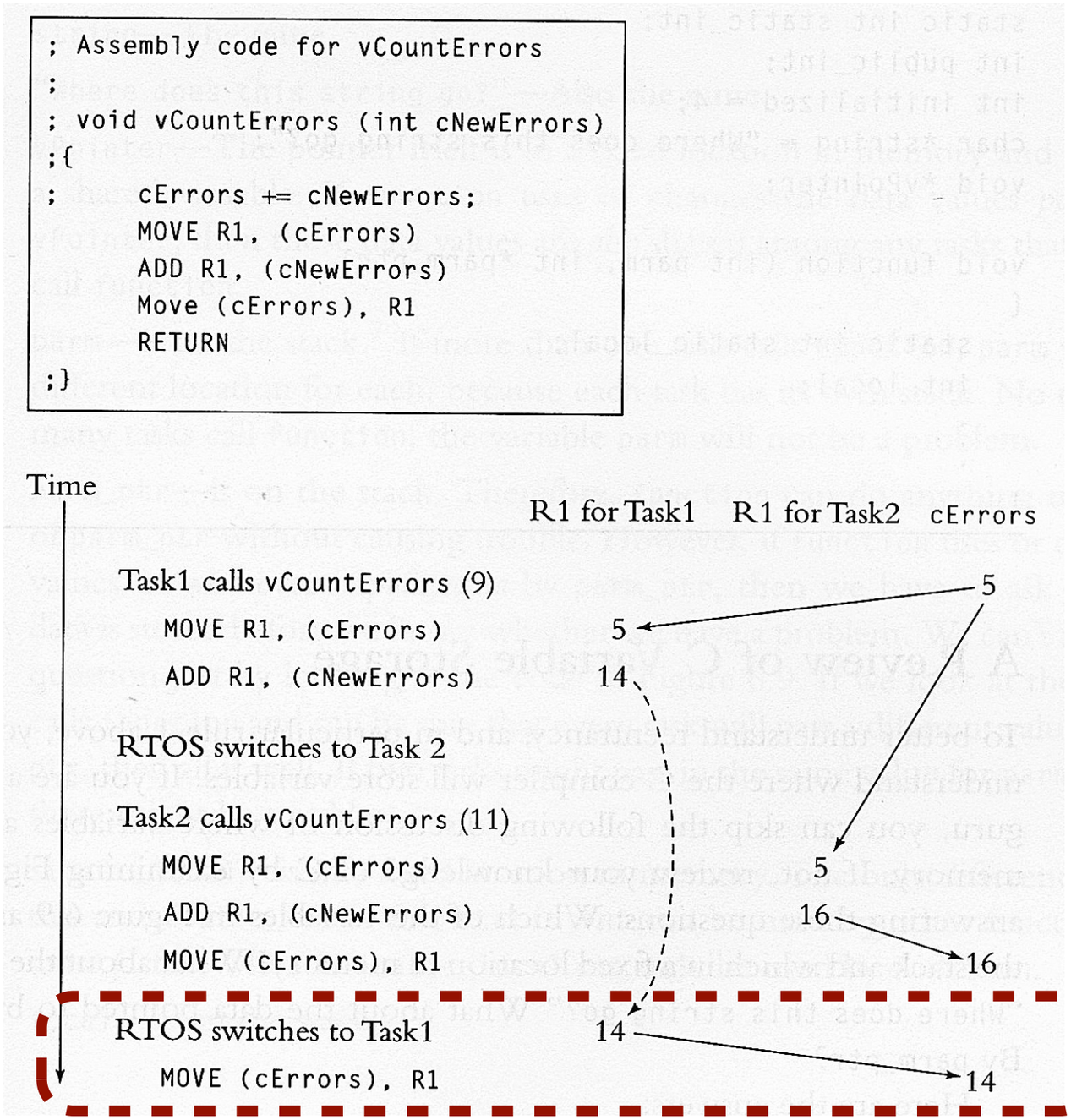
```
; Assembly code for vCountErrors
;
; void vCountErrors (int cNewErrors)
;{
;       cErrors += cNewErrors;
        MOVE R1, (cErrors)
        ADD R1, (cNewErrors)
        Move (cErrors), R1
        RETURN
;}
```



- **Remember:** each task has their own stack which have unique register values.
- Suppose now that the RTOS stops Task1 and runs Task2 and that Task2 calls vCountErrors(11).
- The code in vCountErrors fetches the old value of cErrors(5), adds 11 to it, and stores the result(16).

How can cErrors become corrupted? (Part 4)

```
; Assembly code for vCountErrors
;
; void vCountErrors (int cNewErrors)
;{
;       cErrors += cNewErrors;
        MOVE R1, (cErrors)
        ADD R1, (cNewErrors)
        Move (cErrors), R1
        RETURN
;}
```



- Eventually, the RTOS switches back to Task1, which then executes the next instruction in vCountErrors, saving whatever is in register R1(14) to cErrors and overwriting the value written by Task2(16).
- Instead of cErrors ending up as 25 (the original 5, plus 11 plus 9), it ends up as 14, which is incorrect!

Automatic, static and volatile, variables in C

Example of an automatic variable

```
#include <stdio.h>

void hello_function(int b)
{
 int variable_z = 0;

 //same as variable_z = variable_z + b
 variable_z += b;

 //outputs the value of variable_z
 printf("%d\n", variable_z);
}

int main(int argc, char * const argv[]) {
        hello_function(10); // prints 10
        hello_function(12); // prints 12
        return 0;
}
```

- The **variable_z** is initialized inside the function hello_function.
- Its scope is only within the hello_function.
- The contents of the **variable_z** will be erased after we leave the hello_function.
- **Variable_z** is located in the stack.

Static variables

- A static variable is a variable that has been allocated statically.
- Its lifetime extends across the entire run of the program.
- This is in contrast to the more ephemeral automatic variables (local variables), whose storage is allocated and deallocated on the call stack; and in contrast to objects whose storage is dynamically allocated.
- Static variables are stored in a fixed memory location (**NOT on the stack**).

Example of a static variable

```
#include <stdio.h>

void func()
{
 //x is initialized only once across three
 //calls of func()
 static int x = 0;

 //outputs the value of x
 printf("%d\n", x);
 x = x + 1;
}

int main(int argc, char * const argv[]) {
        func(); // prints 0
        func(); // prints 1
        func(); // prints 2
        return 0;
}
```

- Even though its inside a function, the variable x is only initialized once.
- The contents of the variable will be incremented every time **func()** is entered

Static is a storage class

- In C, static is a reserved word controlling both **lifetime** (as discussed on previous slides) and **linkage** (visibility).
- To be precise, static is a **storage class** (not to be confused with classes in object-oriented programming), as are extern, auto and register (which are also reserved words).
- Every variable and function has one of these storage classes; if a declaration does not specify the storage class, a context-dependent default is used.

Storage class	Lifetime	Linkage
`extern`	static	external (whole program)
`static`	static	internal (translation unit only)
`auto`, `register`	function call	(none)

Effects of a static storage class in C

- Static **global** variables: variables declared as static at the top level of a source file (outside any function definitions) are only visible throughout that source-code file ("file scope", also known as "internal linkage").

- Static **local** variables: variables declared as static inside a function are statically allocated while having the same scope as automatic local variables. Hence whatever values the function puts into its static local variables during one call will still be present when the function is called again.

Storage class	Lifetime	Linkage
`extern`	static	external (whole program)
`static`	static	internal (translation unit only)
`auto`, `register`	function call	(none)

Volatile variables

- A variable or object declared with the volatile keyword usually has special properties related to optimization and/or threading.
- Generally speaking, the volatile keyword is intended to prevent the compiler from applying any optimizations on the code that assume values of variables cannot change "on their own."
- Operations on volatile variables are **NOT** atomic.
- Volatile variables are intended to allow access to memory mapped devices.

Example of memory-mapped I/O

```
static int foo;

void bar(void) {
    foo = 0;

    while (foo != 255)
        ;
}
```

In this example, the code sets the value stored in foo to 0. It then starts to poll that value repeatedly until it changes to 255.

```
void bar_optimized(void) {
    foo = 0;

    while (true)
        ;
}
```

An optimizing compiler will notice that no other code can change the value stored in foo, and will assume that it will remain equal to 0 at all times.

The compiler will then create an infinite loop.

Static vs volatile

```
static int foo;

void bar(void) {
    foo = 0;

    while (foo != 255)
        ;
}
```

- However, foo might represent a location that can be changed by other events such as interrupts.
- This code would never detect a change in the variable foo through interrupts.

```
static volatile int foo;

void bar (void) {
    foo = 0;

    while (foo != 255)
        ;
}
```

- The compiler assumes that the current program is the only part of the system that could change the value (which is by far the most common situation).
- We solve this with the volatile keyword.

Compiler optimizations and volatile variables

```
static int foo;

void bar (void) {
    foo = 0;

    while (*(volatile int *)&foo != 255)
        ;
}
```

- Unfortunately when you include volatile variables, the compile will not optimize the code.
- It is usually overkill to mark the variable volatile as that disables the compiler from optimizing any accesses of that variable instead of the ones that could be problematic.
- It is usually a better idea to cast to volatile where it is needed.

Reentrancy

Reentrant functions

```
static  int cErrors;

void Task1   (void)
{
   (...)
   vCountErrors(9);
   (...)
}

void Task2   (void)
{
  (...)
  vCountErrors(11);
  (...)
}

void vCountErrors   (int cNewErrors)
{
   cErrors += cNewErrors;
}
```

- Reentrant functions are functions that can be called by more than one task and that will always work correctly, even if the RTOS switches from one task to another in the middle of executing the function.
- The function vCountErrors, described on the left, does not qualify.
- You apply three rules to decide if a function is reentrant.

Reentrancy rules

1. A reentrant function may not use variables in a non-atomic way (unless they are stored on the stack of the task that called the function or are otherwise the private variables of that task).

2. A reentrant function may not call any other functions that are not themselves reentrant.

3. A reentrant function may not use the hardware in a non-atomic way.

Reentrant (from wikipedia)

- A computer program or routine is described as reentrant if it can be safely executed concurrently; that is, the routine can be re-entered while it is already running. To be reentrant, a function must:
  - Hold no static (global) non-constant data.
  - Must not return the address to static (global) non-constant data.
  - Must work only on the data provided to it by the caller.
  - Must not rely on locks to singleton resources.
  - Must not call non-reentrant functions.

C-variable storage

```
static int static_int;

int public_int;
int initialized = 4;

char *string = "Where is this string?";
void *vPointer;

void f1 (int parm, int *parm_ptr)
{
   static int static_local;
   int local;
   (...)
 }
```

- To better understand reentrancy, we must understand where the C compiler will store variables.
  - Which of the variables are stored on the stack and which in a fixed location in memory?

C-variable storage

```
static int static_int;

int public_int;
int initialized = 4;

char *string = "Where is this string?";
void *vPointer;

void f1 (int parm, int *parm_ptr)
{
   static int static_local;
   int local;
   (...)
 }
```

- **static_int**: is in a fixed location in memory and is therefore shared by any task that happens to call function.
- **public_int**: Ditto. The only difference between static_int and public_int is that functions in other C files can access public_int, but they cannot access static_int.

C-variable storage

```
static int static_int;

int public_int;
int initialized = 4;

char *string = "Where is this string?";
void *vPointer;

void f1 (int parm, int *parm_ptr)
{
   static int static_local;
   int local;
   (...)
 }
```

- **initialized**: is in a fixed location in memory and is therefore shared by any task that happens to call function. The initial value makes no difference to where the variable is stored.

- ***string**: The same.

C-variable storage

```
static int static_int;

int public_int;
int initialized = 4;

char *string = "Where is this string?";
void *vPointer;

void f1 (int parm, int *parm_ptr)
{
   static int static_local;
   int local;
   (...)
 }
```

- **vPointer**: The pointer itself is in a fixed location in memory and is therefore a shared variable.

- If function f1 uses or changes the data values pointed to by vPointer, then those data values are also shared among any tasks that happen to call function.

C-variable storage

```
static int static_int;

int public_int;
int initialized = 4;

char *string = "Where is this string?";
void *vPointer;

void f1 (int parm, int *parm_ptr)
{
   static int static_local;
   int local;
   (...)
 }
```

- **parm**: is on the stack. If more than one task calls function, parm will be in a different location for each, because each task has its own stack.

- **parm_ptr**: is also on the stack. Therefore, f1 can do anything to the value of parm_ptr without causing trouble. However, if function uses or changes the values of whatever is pointed to by parm_ptr, then we have to ask where that data is stored before we know whether we have a problem.

What is actually a static variable?

```
static int static_int;

int public_int;
int initialized = 4;

char *string = "Where is this string?";
void *vPointer;

void f1 (int parm, int *parm_ptr)
{
   static int static_local;
   int local;
   (...)
 }
```

- A **static variable** is a variable that has been allocated statically (in a well defined memory location).
- Its lifetime extends across the entire run of the program.
- This is in contrast to the more ephemeral automatic variables (local variables), whose storage is allocated and deallocated on the call stack; and in contrast to objects whose storage is dynamically allocated.

C-variable storage

```
static int static_int;

int public_int;
int initialized = 4;

char *string = "Where is this string?";
void *vPointer;

void f1 (int parm, int *parm_ptr)
{
   static int static_local;
   int local;
   (...)
 }
```

- **static_local**: is in a fixed location in memory.
- The only difference between static_local and static_int is that static_int can be used by other functions in the same C file, whereas static_local can only be used by the function f1.
- **local**: is on the stack

This function is not reentrant: Problem #1

```
// Someone else sets this
BOOL fError;

void display  (int j)
{
  if  (!fError)
  {
    printf  ("\nValue:   %d",   j);
    j  = 0;
    fError = TRUE;
  }

  else
  {
    printf("Could not display value");
    fError =  FALSE;
  }
}
```

- The variable **fError** is in a fixed location in memory and is therefore shared by any task that calls display.

- The use of **fError** is not atomic, because the RTOS might switch tasks between the time that it is tested and the time that it is set. This function therefore violates rule 1.

- Note that the variable j is no problem; it's on the stack.

This function is not reentrant: Problem #2

```
// Someone else sets this
BOOL fError;

void display  (int j)
{
  if  (!fError)
  {
    printf  ("\nValue:   %d",   j);
    j  = 0;
    fError = TRUE;
  }

  else
  {
    printf("Could not display value");
    fError =  FALSE;
  }
}
```

- The second problem is that this function may violate rule 2 as well.
- For this function to be reentrant, printf must also be reentrant. Is printf reentrant?

Gray areas of reentrancy

Reentrancy rule #1: A reentrant function may not use variables in a non-atomic way (unless they are stored on the stack of the task that called the function or are otherwise the private variables of that task).

```
static  int cErrors;

void vCountErrors  (void) {++cErrors; }
```

- There are some gray areas between reentrant and non-reentrant functions.
- The code here shows a very simple function in the gray area.
- This function obviously modifies a non-stack variable, but rule 1 says that a reentrant function may not use non-stack variables in a non-atomic way. The question is: is incrementing cErrors atomic?

cErrors is NOT atomic

```
static  int cErrors;

void vCountErrors  (void) {++cErrors; }
```

- If you're using an 8051, an 8-bit micro-controller, then ++cErrors is likely to compile into assembly code something like this:

```
    MOV     DPTR,#cErrors+01H
    MOVXA,@DPTR
    INC  A
    MOVX©DPTR,A
    JNZ  noCarry
    MOV  DPTR,# cErrors
    MOVX    A,@DPTR
    MOVX©DPTR,A
noCarry:
    RET
```

- Which isn't anywhere close to atomic, since it takes nine instructions to do the real work, and an interrupt might occur anywhere among them.

cErrors is atomic

```
static  int cErrors;

void vCountErrors  (void) {++cErrors; }
```

- if you're using an Intel 80x86, you might get:

```
INC  (cErrors)
RET
```

- Which is atomic!
- If you really need the performance of the one-instruction function and you're using an 80x86 and you put in lots of comments, perhaps you can get away with writing vCountErrors this way.
- However, there's no way to know that it will work with the next version of the compiler or with some other microprocessor to which you later have to port it.