(https://www.edx.org)

UTAustinX: UT.6.01x Embedded Systems - Shape the World

KarenWest (/dashboard) ▼

Courseware (/courses/UTAustinX/UT.6.01x/1T2014/courseware) Course Info (/courses/UTAustinX/UT.6.01x/1T2014/info)

Discussion (/courses/UTAustinX/UT.6.01x/1T2014/discussion/forum) Progress (/courses/UTAustinX/UT.6.01x/1T2014/progress)

Questions (/courses/UTAustinX/UT.6.01x/1T2014/a3da417940af4ec49a9c02b3eae3460b/)

Syllabus (/courses/UTAustinX/UT.6.01x/1T2014/a827a8b3cc204927b6efaa49580170d1/)

Embedded Systems Community (/courses/UTAustinX/UT.6.01x/1T2014/e3df91316c544d3e8e21944fde3ed46c/)

For regular function calls we use the registers and stack to pass parameters, but interrupt threads have logically separate registers and stack. More specifically, registers are automatically saved by the processor as it switches from main program (foreground thread) to interrupt service routine (background thread). Exiting an ISR will restore the registers back to their previous values. Thus, all parameter passing must occur through global memory. One cannot pass data from the main program to the interrupt service routine using registers or the stack.

In this chapter, multi-threading means one main program (foreground thread) and multiple ISRs (background threads). An operating system allows multiple foreground threads. For more information on operating systems see Embedded Systems: Real-Time Operating Systems for ARM® Cortex™-M Microcontrollers. Synchronizing threads is a critical task affecting efficiency and effectiveness of systems using interrupts. In this section, we will present in general form three constructs to synchronize threads: binary semaphore, mailbox, and FIFO queue.

Help

VIDEO 12.2. INTER-THREAD COMMUNICATION AND SYNCHRONIZATION

C12 2 Inter-thread Communication and Synchronization YouTube

11:28 / 11:28 1.0x

DR. RAMESH YERRABALLI: So let's take a look at communication.

I'm going to call those the inter-thread communication.

Remember that we talked about how we have our main program, which

is running some code repeatedly, and we have our background threads.

So background thread is some interrupt service routine that's being triggered.

That's triggered.

It performs some task, and it returns from the interrupt.

And there could be a second one and so on.

So the question is, how does information get

passed between the main thread and the background thread?

So first, we might think that well, we could use something like variable,

a local variable if you will, or a register to put things in.

And we will realize that when we do a context which all that information is

lost because we have a new context.

So what can we use to communicate between the two?

So the part that is accessible to both the main and the background

is memory, so the global memory.

So let's see how this will work.

We will look at it in three examples.

The first example we will look at is communicating

between the background and foreground using a simple flag.

A flag like this could involve the main program is waiting for some event

to occur, and the background thread will allow the main program

to know that this event has occurred by raising a flag.

The second one is a little more substantial.

That is, we will communicate back and forth

using what I call the concept of a mailbox.

A mailbox involves actual data that is being communicated and a status.

So we have data and status.

So we'll use two global variables for this.

The third one is what good real time systems

should use because they allow for good asynchronous communication.

These are buffers.

And specifically, we'll use a First in First Out

queue (FIFO), which allows us to buffer.

So the background thread, if it wants to communicate to the main thread,

adds to a buffer.

And the main program is going to consume from this buffer.

So this will be the third one we'll look at.

So let's look at the first example.

The first example involves where the main program

wants to know whether an event has occurred.

And the occurrence of this event is through an interrupt service routine.

So the occurrence of the event will be indicated by the interrupt service

routine by setting a flag-- which is a global variable-- to 1.

It sets that and it quits.

Now, the main program which was doing its work-- whatever that work was it.

This is the work of the main program, the task, the main task.

It was performing that task, but it periodically-- actually every so

often-- it'll check this flag.

And if the flag is set to 1, then it does some work.

So it'll do some work associated with that event.

And then it goes back and does its other calculations.

On the other hand, if the flag is not set,

then it'll just go back and do what it was normally meant to do,

which is repeat the core work that it was meant to design for.

And one thing we have to be sure is that when

we have in fact done the work associated to the event, we will disable the flag.

We'll set it to 0.

This way, when I come back to check the flag again,

I'm not finding the flag to be on simply because I did not turn it off.

DR. JONATHAN VALVANO: We can extend this idea of a flag

to incorporate passing of data through a mailbox.

The mailbox has two components.

It has the flag, just like the previous example,

but we're going to add a data field.

In this particular example of a mailbox, the purpose

will be to pass data from the interrupt service routine into the main program.

DR. RAMESH YERRABALLI: For example, the UART

could have been triggered by the arrival of a data packet.

And so the data that is being read from the UART

could be put into this data buffer, if you will, a mailbox.

And the flag is raised to indicate to the main program

that there is fresh data.

DR. JONATHAN VALVANO: So in the interrupt service routine,

we will read the data from the input, and we

will write it to a global variable.

And then we will set the flag to a 1.

Now, the main program, which looks a lot like this, is checking the flag.

If the flag is 0, that means there is no data.

Then it will do something else.

But if the flag is a 1, there is data.

Then, we can process or observe or look at or enjoy this data.

It's called consuming the data.

We will use it up.

Whatever we're supposed to do with that data, we will do.

And then just like we did with the flag, we will clear the flag.

And so we notice that there is a clear definition of what this mailbox is.

If the flag is equal to 0, that means there is no data and the main program

is waiting.

DR. RAMESH YERRABALLI: Doing other things.

DR. JONATHAN VALVANO: But if the flag is 1, that means there is data,

and the main program, if it notices it, can take the data.

DR. RAMESH YERRABALLI: So this is a classic paradigm

called producer consumer paradigm.

The background thread is going to produce data.

And the foreground thread, which is our main thread, consumes data.

So Jon, can we also have the other way around?

DR. JONATHAN VALVANO: Yes.

The main program could be the producer.

And what it would do is it would write into data and it would set the flag.

And the interrupt service routine could be the consumer

and it would check the flag and then use the data.

So data could flow in the other direction in a mailbox.

DR. RAMESH YERRABALLI: So this allows for communication

between any threads in any direction.

DR. JONATHAN VALVANO: Absolutely.

Let's look at the third way to communicate,

and that's called a FIFO, or a First In, First Out buffer.

If we're passing data from the interrupt service routine,

this means this is probably an input device.

If we're passing data, we are going to put that data into a FIFO.

This is a buffer which will store this data.

When we put, we're going to put it into this FIFO.

It's a buffer that contains data.

The main program, when it wants data, it will call Get.

So there are two functions with a FIFO.

And then it will consume that data or use it up.

And so the data flows from the producer, which is, in this case, the interrupt,

through the FIFO into the main program, which is the consumer.

There are two functions, Put, which stores data in, and Get,

which stores it out.

So the FIFO has a number of properties, one of which

is it is order preserving, which means the order in which I

put it will match the order in which I get it.

So data is streamed from one thread to another.

DR. RAMESH YERRABALLI: So the FIFO is extending the idea of a mailbox.

A mailbox is just one data item.

But now we've expanded it so that we can have n data items, many data items.

Help

A **binary semaphore** is simply a shared flag, as described in Figure 12.1. There are two operations one can perform on a semaphore. **Signal** is the action that sets the flag. **Wait** is the action that checks the flag, and if the flag is set, the flag is cleared and important stuff is performed. This flag must exist as a private global variable with restricted access to only these two code pieces. In C, we add the qualifier `static` to a global variable to restrict access to software within the same file. In order to reduce complexity of the system, it will be important to limit the access to this flag to as few modules as possible.

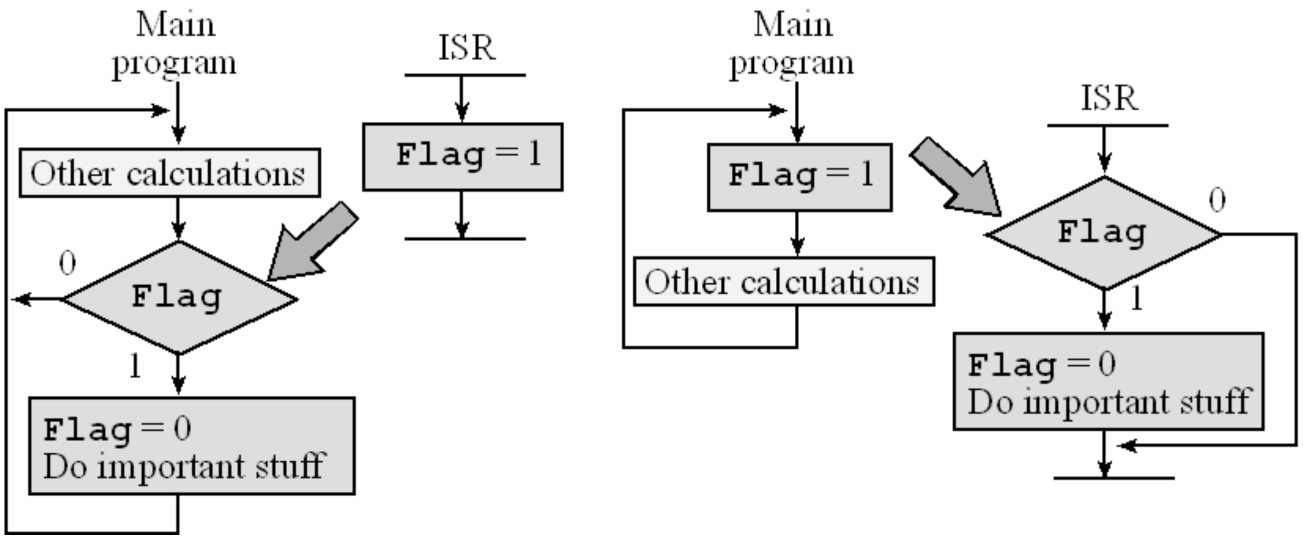

Figure 12.1. A semaphore can be used to synchronize threads.

A flag of course has two states: 0 and 1. However, it is good design to assign a meaning to this flag. For example, 0 might mean the switch has not been pressed, and 1 might mean the switch has been pressed. Figure 12.1 shows two examples of the binary semaphore. The big arrows in this figure signify synchronization links between the threads. In the example on the left, the ISR signals the semaphore and the main program waits on the semaphore. Notice the "important stuff" is run in the foreground once per execution of the ISR. In the example on the right, the main program signals the semaphore and the ISR waits. It is good design to have NO backwards jumps in an ISR. In this particular application, if the ISR is running and the semaphore is 0, the action is just skipped and the computer returns from the interrupt.