Synchronization | 11.1 I/O Synchronization | ... (https://www.edx.org) https://courses.edx.org/courses/UTAustinX/UT...

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/)

Periodic Polling uses a clock interrupt to periodically check the I/O status. At the time of the interrupt the software will check the I/O status, performing actions as needed. With an input device, a ready flag is set when the input device has new data. At the next periodic interrupt after an input flag is set, the software will read the data and save them in global RAM. With an output device, a ready flag is set when the output device is idle. At the next periodic interrupt after an output flag is set, the software will get data from a global structure, and write it. Periodic polling will be used in situations that require interrupts, but the I/O device does not support interrupt requests directly.

DMA, or direct memory access, is an interfacing approach that transfers data directly to/from memory. With an input device, the hardware will request a DMA transfer when the input device has new data. Without the software's knowledge or permission the DMA controller will read data from the input device and save it in memory. With an output device, the hardware will request a DMA transfer when the output device is idle. The DMA controller will get data from memory, and then write it to the device. Sometimes we configure the hardware timer to request DMA transfers on a periodic basis. DMA can be used to implement a high-speed data acquisition system. DMA synchronization will be used in situations where high bandwidth and low latency are important. DMA will not be covered in this introductory class. For details on how to implement DMA on the LM4F120/TM4C123, see Embedded Systems: Real-Time Operating Systems for ARM Cortex-M Microcontrollers, ISBN: 978-1466468863 (http://users.ece.utexas.edu/%7Evalvano/arm/outline3.htm).

Help

One can think of the hardware being in one of three states. The **idle** state is when the device is disabled or inactive. No I/O occurs in the idle state. When active (not idle) the hardware toggles between the **busy** and **ready** states. The interface includes a **flag** specifying either busy (0) or ready (1) status. Hardware-software synchronization revolves around this flag:

- The hardware will set the flag when the hardware component is complete.
- The software can read the flag to determine if the device is busy or ready.
- The software can clear the flag, signifying the software component is complete.
- This flag serves as the hardware triggering event for an interrupt.

For an input device, a status flag is set when new input data is available. The "busy to ready" state transition will cause a busy-wait loop to complete, see middle of Figure 11.1. Once the software recognizes the input device has new data, it will read the data and ask the input device to create more data. It is the **busy to ready** state transition that signals to the software that the hardware task is complete, and now software service is required. When the hardware is in the ready state the I/O transaction is complete. Often the simple process of reading the data will clear the flag and request another input.

The problem with I/O devices is that they are usually much slower than software execution. Therefore, we need synchronization, which is the process of the hardware and software waiting for each other in a manner such that data is properly transmitted. A way to visualize this synchronization is to draw a state versus time plot of the activities of the hardware and software. For an input device, the software begins by waiting for new input. When the input device is busy it is in the process of creating new input. When the input device is ready, new data is available. When the input device makes the transition from busy to ready, it releases the software to go forward. In a similar way, when the software accepts the input, it can release the input device hardware. The arrows in Figure 11.3 represent the synchronizing events. In this example, the time for the software to read and process the data is less than the time for the input device to create new input. This situation is called **I/O bound**, meaning the bandwidth is limited by the speed of the I/O hardware.

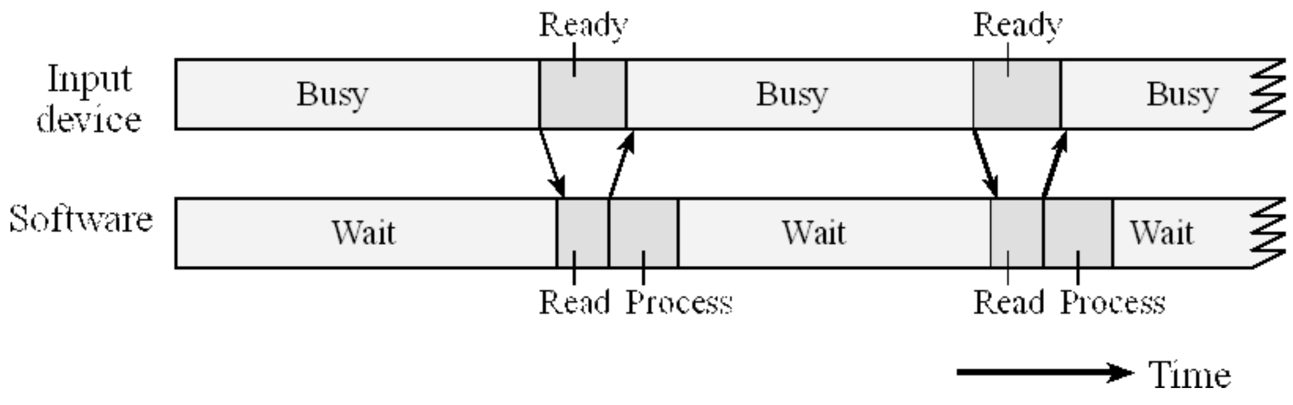

Figure 11.3. The software must wait for the input device to be ready (I/O bound input interface).

If the input device were faster than the software, then the software waiting time would be zero. This situation is called **CPU bound** (meaning the bandwidth is limited by the speed of the executing software). In real systems the bandwidth depends on both the hardware and the software. Another characteristic of real systems is the data can vary over time, like car traffic arriving and leaving a road intersection. In other words, the same I/O channel can sometimes be I/O bound, but at other times the channel could be CPU bound.

We can store or buffer data in a **first in first out** (FIFO) queue, see Figure 11.4, while passing the data from one module to another. These modules may be input devices, output devices or software. Because the buffer separates the generation of data from the consumption of data, it is very efficient, and hence it is prevalent in I/O communication. In particular, it can handle situations where there is an increase or decrease in the rates at which data is produced or consumed. Other names for this important interfacing mechanism include **bounded buffer**, **producer-consumer**, and **buffered I/O**. Data are entered into the FIFO as they arrive; we call **Put** to store data in the FIFO. Data are removed as they leave; we call **Get** to remove data from the FIFO. The FIFO maintains the order of the data, as it passes through the buffer. We can think of a FIFO like a line at the post office. There is space in the lobby for a finite number of people to wait. As customers enter the post office they get in line at the end (put onto FIFO). As the postal worker services the customers, people at the front leave the line (get from the FIFO). It is bad situation (a serious error) if the waiting room becomes full and there is no room for people to wait (full FIFO). However, if there are no customers waiting (empty FIFO) the postal worker sits idle. An empty FIFO may be inefficient, but it is not considered an error.

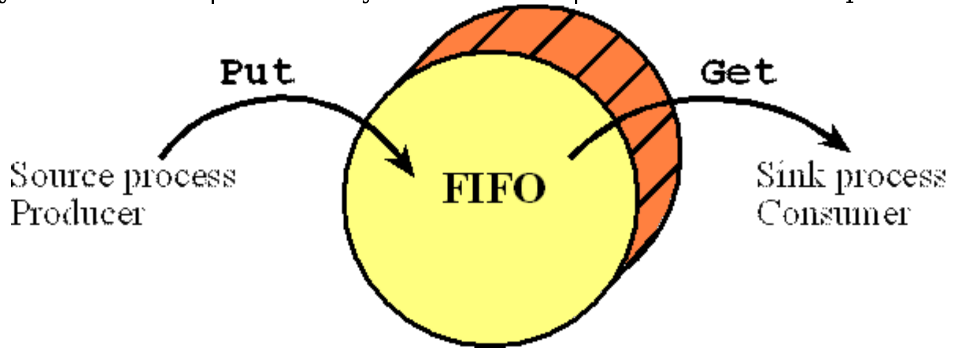

Figure 11.4. A FIFO queue can be used to pass data from a producer to a consumer. At any given time there can be a variable number of elements stored in the FIFO. The order in which data are removed is the same as the order the data are entered.

The busy-wait method is classified as unbuffered because the hardware and software must wait for each other during the transmission of each piece of data. The interrupt solution (shown in the right part of Figure 11.1) is classified as buffered, because the system allows the input device to run continuously, filling a FIFO with data as fast as it can. In the same way, the software can empty the buffer whenever it is ready and whenever there is data in the buffer. The buffering used in an interrupt interface may be a hardware FIFO, a software FIFO, or both hardware and software FIFOs. We will see the FIFO queues will allow the I/O interface to operate during both situations: I/O bound and CPU bound.

Help

For an output device, a status flag is set when the output is idle and ready to accept more data. The "busy to ready" state transition causes a busy-wait loop to complete, see the middle part of Figure 11.2. Once the software recognizes the output is idle, it gives the output device another piece of data to output. It will be important to make sure the software clears the flag each time new output is started. Figure 11.5 contains a state versus time plot of the activities of the output device hardware and software. For an output device, the software begins by generating data then sending it to the output device. When the output device is busy it is processing the data. Normally when the software writes data to an output port, that only starts the output process. The time it takes an output device to process data is usually longer than the software execution time. When the output device is done, it is ready for new data. When the output device makes the transition from busy to ready, it releases the software to go forward. In a similar way, when the software writes data to the output, it releases the output device hardware. The output interface illustrated in Figure 11.5 is also I/O bound because the time for the output device to process data is longer than the time for the software to generate and write it. Again, I/O bound means the bandwidth is limited by the speed of the I/O hardware.

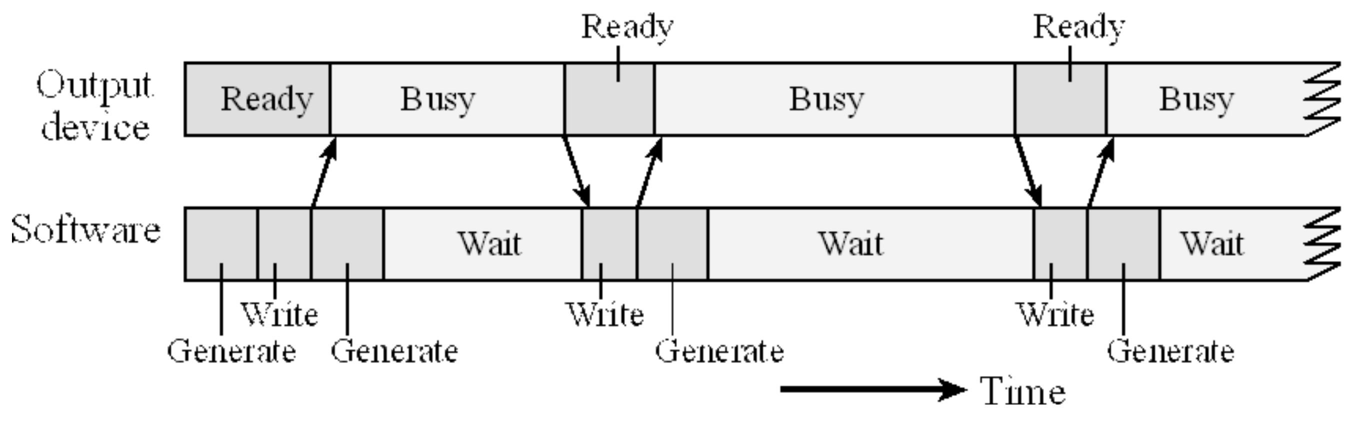

Figure 11.5. The software must wait for the output device to finish the previous operation (I/O bound).

The busy-wait solution for this output interface is also unbuffered, because when the hardware is done, it will wait for the software and after the software generates data, it waits for the hardware. On the other hand, the interrupt solution (shown as the right part of Figure 11.2) is buffered, because the system allows the software to run continuously, filling a FIFO as fast as it wishes. In the same way, the hardware can empty the buffer whenever it is ready and whenever there is data in the FIFO. Again, FIFO queues allow the I/O interface to operate during both situations: I/O bound and CPU bound.

On some systems an interrupt will be generated on a hardware failure. Examples include power failure, temperature too high, memory failure, and mechanical tampering of secure systems. Usually, these events are extremely important and require immediate attention. The Cortex™-M processor will execute special software (**`fault`**) when it tries to execute an illegal instruction, access an illegal memory location, or attempt an illegal I/O operation.