UNIVERSITY OF WATERLOO
Department of Electrical and Computer Engineering

Section III: Interfacing Software, Introduction to Synchronization, and Device Drivers

WATERLOO
ENGINEERING

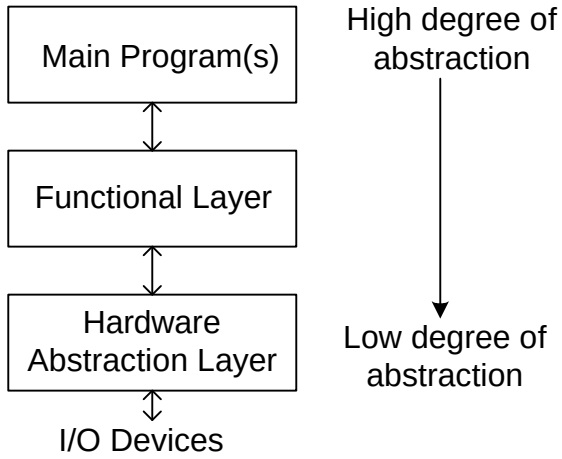
f2013-1.0

General Synchronization Introduction: Section Contents

This section of the notes contains the following subsections.

Software Development

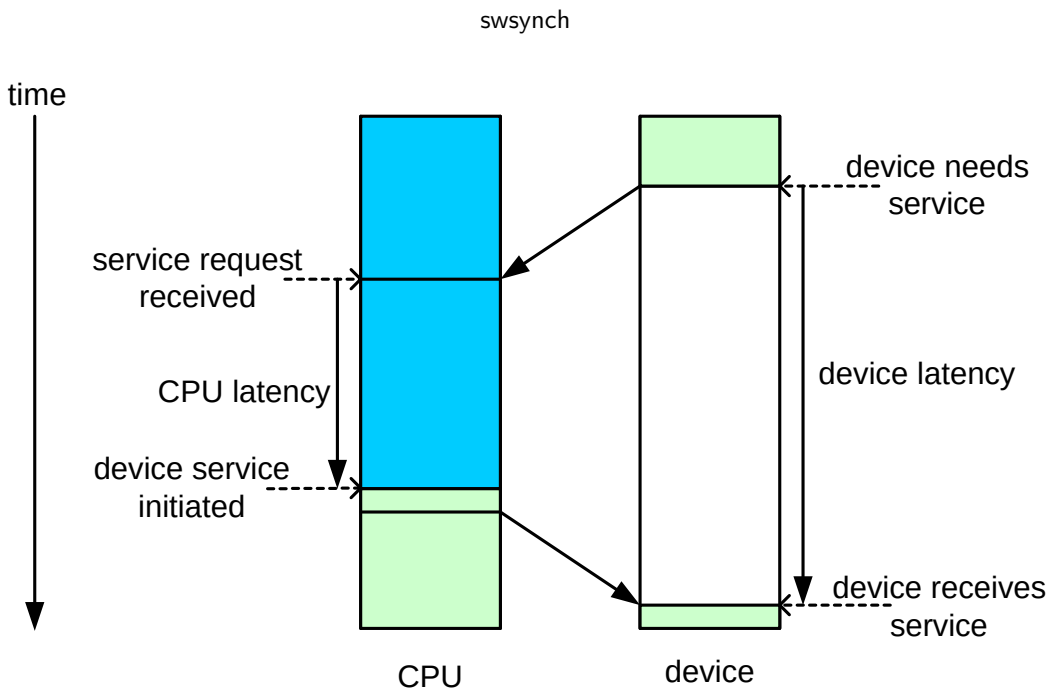
- A microprocessor system requires both hardware and software design.
- Good software is written in a modular manner.



- Low level functions interact directly with the system's hardware.
  - initialize device
  - read/write to the device
  - handle software synchronization through polling and/or interrupts

Software Synchronization

- When a device completes its task it must synchronize with the processor.
- Events may have a priority in order to ensure minimum latency.

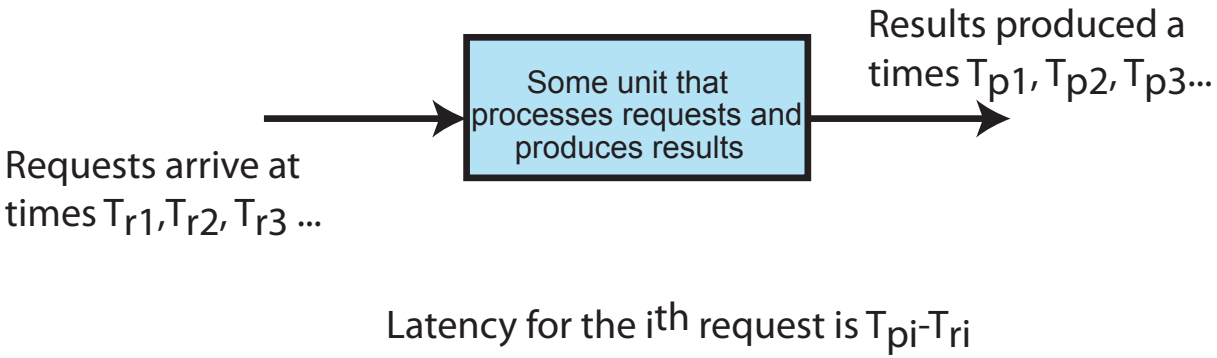


CPU Latency and Device Latency: is the time between a service request and when the service is initiated. Latency can involve both hardware and software delays.

Real-Time System: is one that guarantees a worst-case latency for critical events.

Performance, Latency and Throughput

latthroug



Two terms that are used in discussing system performance are Latency and Throughput.

Latency: The delay between the arrival of the request and the completion of service. One could also consider the average latency or the maximum latency by considering more results. (Note: on the surface this appears to be slightly different than the definition of CPU Latency defined earlier. However, this difference can be resolved by noting that the system is viewed as completing the request once the ISR has started.)

Throughput: is a measure of how many items can be processed per unit of time. For example, a system could have a very high latency (5 years for UW Engineering) but still have a throughput of 900 graduates per year.

Synchronization Mechanisms

Adapted from J.W. Valvano, *Embedded Microcomputer Systems: Real Time Interfacing*

- *Blind Cycle*: software waits a fixed amount of time and then acts on the data whether or not the device is ready
- *Periodic Polling*: device status is checked after a pre-determined amount of time and this repeats until the device is done. This is usually implemented with a timer interrupt.
- *Occasional Polling*: device status is checked at the convenience of the designer.
- *Polling Loop (Gadfly or Busy Waiting)*: software continuously checks the I/O status, waiting for the device to be done. Although this is often implemented as a very tight loop (one status register test and then loop if not yet ready), it could be implemented as a series of tests (say test 5 I/O status registers and service any that are requesting service).

Synchronization Mechanisms – Cont.

- Interrupts
  - when device is done it generates a hardware interrupt
- First 3 options are one-sided, as the synchronization is dependent primarily upon the CPU timing
- Polling loop and interrupts are more device oriented, the latency seen by the device is minimized

Polling Loop Synchronization – Input

- poll device and wait until data is available

```
while ( not data_available ) loop //Tight polling loop
read data
clear data_available flag (usually by hardware when data read)
process data
return
```

Polling Loop Synchronization – Output

- poll before sending data

```
while ( not ready_to_output ) loop
clear ready_to_output flag (usually by hardware when data output)
output data
return
```

- poll after sending data (this assumes that the interface is initially ready to output)

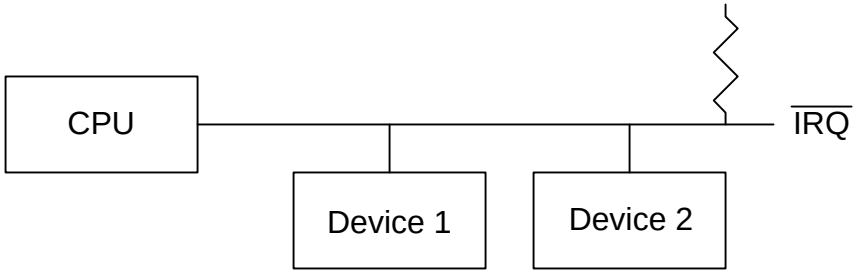
```
clear ready_to_output flag (usually by hardware when data output)
output data
while ( not ready_to_output ) loop
return
```

Interrupt Synchronization

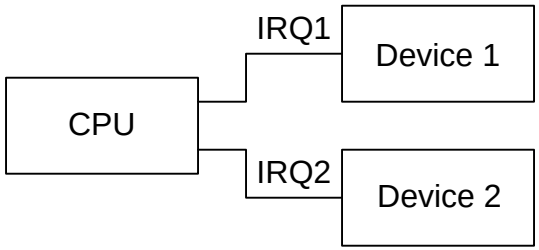
1. a device notifies the CPU of an interrupt request
2. CPU completes execution of the current instruction
3. execution of the main program is suspended
4. interrupts are disabled (processor specific)
5. some internal registers are saved (including the program counter)
6. device may be acknowledged
7. interrupt service routine is selected
8. interrupt service routine is executed (more on this later)
9. registers are restored, if required, including the program counter
10. interrupts are enabled (processor specific)
11. execution of the main program resumes

CPU Notification

- Interrupts must be handled from multiple sources
- Single interrupt request line

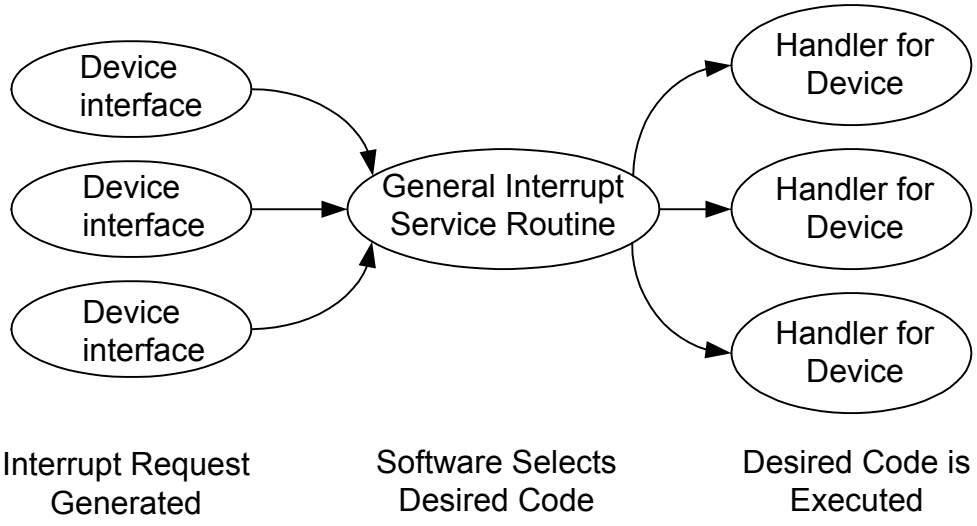


- Multiple interrupt request lines



Interrupt Service Routine (ISR) Selection

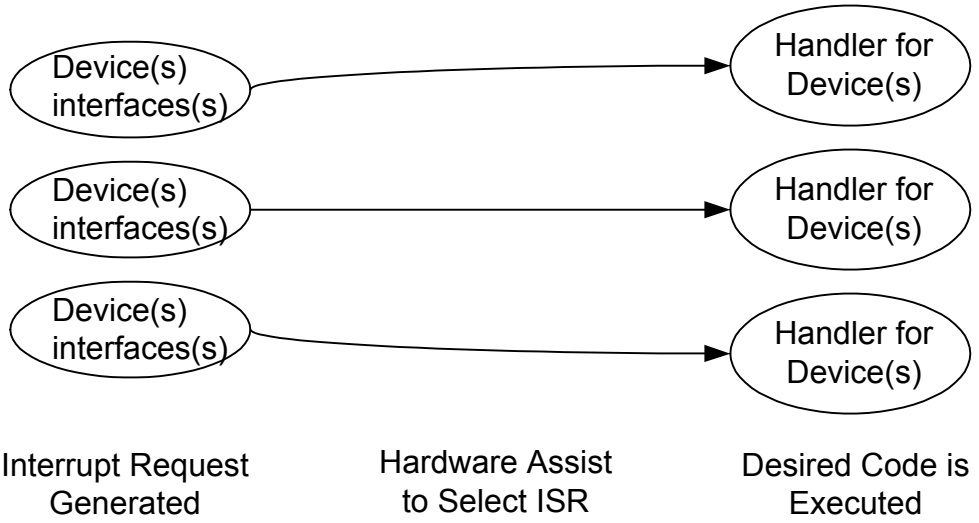
- Non-vectored interrupts
  - devices are polled to determine source
  - priority must be determined (in software)



oneirq

ISR Selection

- Vectored Interrupts
  - requests are associated with an interrupt vector
  - fixed priority associated with the interrupt vector
  - interrupt service routine (ISR) at vector address is executed



Interrupt Service Routine (ISR)

- should execute as fast as possible

1. any registers used by the function should be saved
2. device may be acknowledged
3. interrupts may be enabled to allow higher (and possibly same) priority interrupts (design specific)
4. test for valid interrupt and/or determine the source of the interrupt
5. complete desired action
6. restore registers (interrupts may need to be disabled during this step)
7. return from interrupt

Interrupt Initialization

The following steps need to be taken when initializing a system which uses interrupts.

1. disable all interrupts
2. enable device interface interrupts by setting appropriate device interface registers
3. set interrupt mask to allow interrupts from device
4. initialize interrupt vector with address of ISR
5. enable interrupts as required

Device Drivers

A device driver is the software associated with a particular device. It will include:

- Data Structures
  - variables needed to access the device interface registers
  - variables associated with the state of the device
  - data buffers
- Initialization Functions
  - device initialization
  - synchronization initialization
  - initialization of driver variables
- I/O Functions
  - functions to input and/or output to the device
- interrupt service routine

The NIOS system provides the data structures and some of the driver functions for its devices as part of the custom SDK associated with each design. This information can be found in `nios.h` and `nios_peripherals.h`.

Note: Terms used in Interfacing Software, Synchronization, Generalized IO, and Device Drivers

Terms: