Sources of interrupts

Hardware Interrupts on Raspberry Pi

Linux (and Unix) in-general never really expected user-land code to handle hardware interrupts (that's the kernels job!), and that's the issue at-hand—there really is **no API for handling interrupts in user-land code**.

So in the traditional sense, there isn't "ISR" - Interrupt Service Routine.

So what can you do - many things. One, write your own operating system. A bit drastic that one! Use another OS - well the only other viable choice right now is RiscOS... You could write your own kernel module - this is the closest you'll get to a real ISR. You write the module, poke the hardware to tell it to deliver an edge triggered interrupt and tell Linux to register your driver with the interrupt handler and off you go. It actually is a real ISR - within the Linux kernel interrupt handler - this is the same mechanism Linux uses to access all interrupt driven hardware, disk controllers, serial ports, and so on. All your code is running at the kernel level with all the implications that has - security, robustness, crashability, etc.

Great if you're a Linux kernel module writer, not so good for the other 99.9999% of us... (Although from what I've seen the learning curve isn't that steep for an accomplished C programmer, however, much though I wish I have dived in, I've never found time).

So we're left with Plan B. [Multithreading combined with sleeping]

Firstly, it is good to know that it is possible to get an edge detected interrupt on a GPIO pin delivered into a user-land program. It's not technically the same as a true ISR, but it does have the same effect and with care you can handle about 10K interrupts/second although realistically at that speed the overhead is so high, there's little cpu left for anything else.

The way to do it is to create a thread in your program - that's a function that you write, and which you then ask Linux to schedule concurrently with the main program. That thread then asks the kernel to make it sleep and wake it up when the interrupt happens.

Meanwhile, your main part of the program carries on executing, doing its thing, whatever that might be.

When the interrupt happens, the main program is suspended and the sleeping thread is woken up, it can do "stuff" to handle the interrupt, possibly use global variables to communicate with the main program, then go back to sleep, waiting for the next interrupt.

Note that it's not the same as polling as the thread is stopped and consumes zero cpu, (which polling wouldn't) however the mechanism inside the Linux kernel to suspend the main task and wake up the sleeping thread when the interrupt fires does have more overhead than something you might be used to on an embedded controller with no operating system as such.

https://raspi.tv/2013/how-to-use-interrupts-with-python-on-the-raspberry-pi-and-rpi-gpio-part-2

https://www.raspberrypi.org/forums/viewtopic.php?t=9207

1

Sources of interrupts

Counter-timer-system and Hardware interrupts—Arduino Uno

Most microcontrollers have a counter/timer system—special support hardware for higher speed operations.

Example, PWM with an `analogWrite` command on the Arduino.

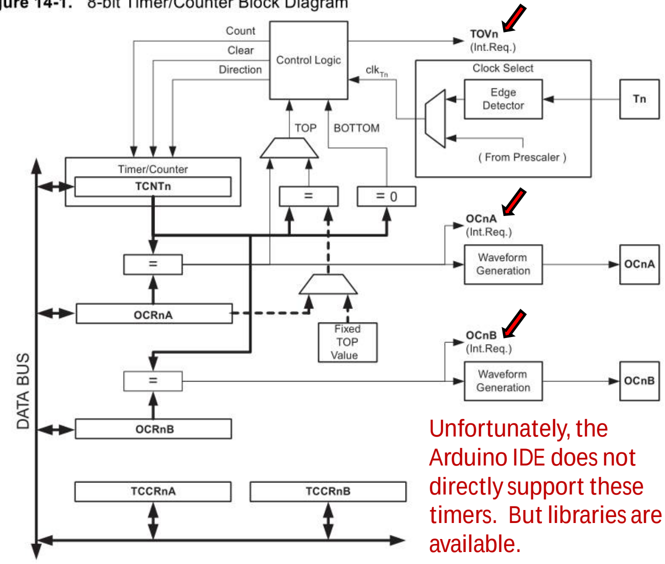
At the right, a bit-slice block diagram. Arduino Uno has three of these, Timer 1, Timer 2, Timer 3. Their output compare signals are directed to six different GPIO ports.

Timer output	Arduino Uno digital I/O pin	Chip pin	Pin name
OC0A	6	12	PD6
OC0B	5	11	PD5
OC1A	9	15	PB1
OC1B	10	16	PB2
OC2A	11	17	PB3
OC2B	3	5	PD3

https://www.arduino.cc/en/Tutorial/SecretsOfArduinoPWM

Each Timer can generate three different hardware interrupt signals.

Figure 14-1. 8-bit Timer/Counter Block Diagram



Unfortunately, the Arduino IDE does not directly support these timers. But libraries are available.

Illustration is from the AVR datasheet

2

The agenda—understanding interrupt-driven I/O (and by extension, multitasking)

An example to give some context

Memory capabilities needed for subroutines (functions, procedures, interrupts, are types of subroutines)

Sources of interrupts including counter-timer systems

Advantages of using interrupt-driven I/O—so obvious this section is hardly needed.
- --Alternatives to interrupt driven I/O are gadfly (uncontrolled—annoying) I/O or various polling techniques, all of which waste processor cycles prodigiously.
- --Interrupts are foundational to object-oriented programming
- --Many embedded systems that use interrupts have very little other code to run!

Risks of interrupt-driven I/O
- --density limit
- --latency and resolution limits
- --interval restrictions
- --critical regions in code
- --deadlock

3

The agenda—understanding interrupt-driven I/O (and by extension, multitasking)

An example to give some context

Memory capabilities needed for subroutines (functions, procedures, interrupts, are types of subroutines)

Sources of interrupts including counter-timer systems

Advantages of using interrupt-driven I/O—so obvious this section is hardly needed.
- --Alternatives to interrupt driven I/O are gadfly (uncontrolled—annoying) I/O or various polling techniques, all of which waste processor cycles prodigiously.
- --Interrupts are foundational to object-oriented programming
- --Many embedded systems that use interrupts have very little other code to run!

Risks of interrupt-driven I/O
- --density limit
- --latency and resolution limits
- --interval restrictions
- --critical regions in code
- --deadlock

4

Advantages of interrupts—foundational to object-oriented programming

The concept of object-oriented programming illustrated in the programming language *Scratch* (aimed at K-12!) Scratch is bundled with Raspbian on the Raspberry Pi. Unfortunately, it completely ignores the GPIO port ;-(

An illustration of an interrupt-driven game. (There is no `main()` or `setup()` or `loop()` routine in scratch.)
https://scratch.mit.edu/projects/360841305/editor/

Scratch is purely is object-oriented.

Each "sprite" is an object in memory and available at all times.

The objects can talk to each other via global variables, here called "broadcasts."

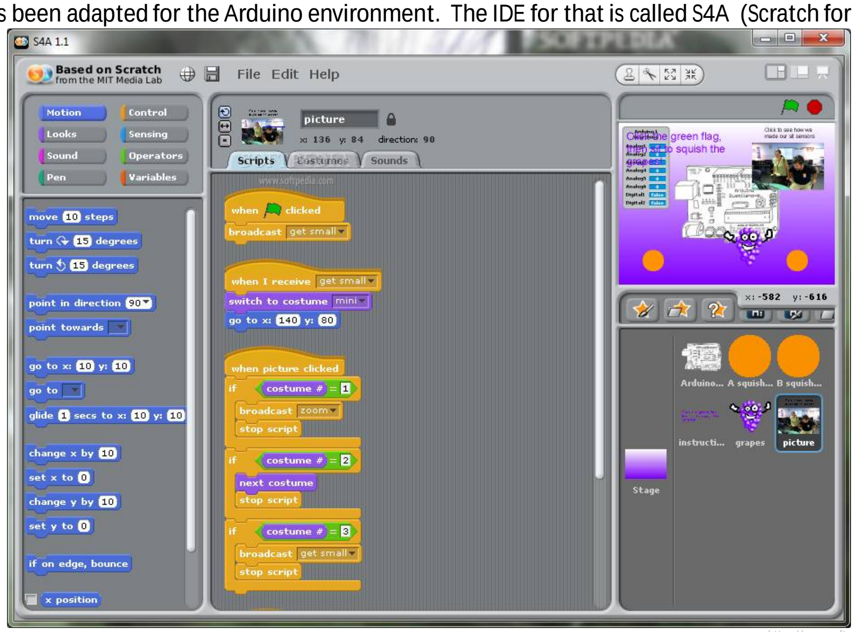
The objects have "methods" that can be invoked from other objects. Once invoked the method runs to completion without further supervision or return to the invoking program.

Methods can be invoked by interrupts like key-presses, mouse clicks, etc.

5

Advantages of interrupts—foundational to object-oriented programming

Scratch has been adapted for the Arduino environment. The IDE for that is called S4A (Scratch for Arduino.)

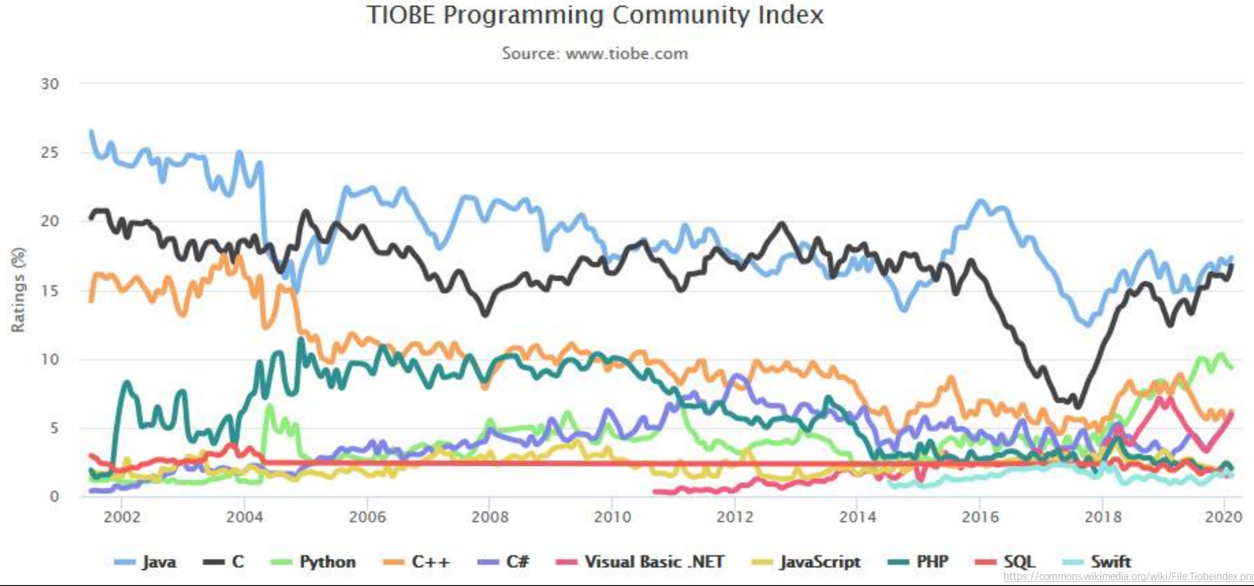


https://www.softpaz.com/software/download-s4a-windows-71755.htm

6

Advantages of interrupts—foundational to object-oriented programming

But Scratch and S4A are toys. C/C++ has it all, an that is the language of the Arduino IDE.
Python also has it all, save for drivers for the hardware you connect to the GPIO port!



7

The agenda—understanding interrupt-driven I/O (and by extension, multitasking)

An example to give some context

Memory capabilities needed for subroutines (functions, procedures, interrupts, are types of subroutines)

Sources of interrupts including counter-timer systems

Advantages of using interrupt-driven I/O—so obvious this section is hardly needed.
--Alternatives to interrupt driven I/O are gadfly (uncontrolled—annoying) I/O or various polling techniques, all of which waste processor cycles prodigiously.
--Interrupts are foundational to object-oriented programming
--Many embedded systems that use interrupts have very little other code to run!

Next

Risks of interrupt-driven I/O
--density limit
--latency and resolution limits
--interval restrictions
--critical regions in code
--deadlock

8

Risks of interrupt-driven I/O

Object-Oriented programming (OOP) is not without its critics.

OOP puts emphasis on data and objects (in our case, buttons, sensors, displays, and the related data). That's good. But OOP takes emphasis away from thoughtful use of algorithms and procedures.

While OOP is conceptually valuable for programmers, simplifying their work, it can lead to a chaotic and wasteful flow of code execution, commonly called "bloat." When OOP programs suffer from bugs the call is often to throw more hardware (memory, clock speed) at the problem without serious thought of figuring out exactly how the code is executing and what is using the hardware resources. This is not a competitive method in an embedded system. (But OOP with "more hardware" has been profitable in the general-purpose and smartphone markets.)

https://en.wikipedia.org/wiki/Object-oriented_programming#Criticism

https://commons.wikimedia.org/wiki/File:Tiobeindex.png

9

Risks of interrupt-driven I/O

Object-Oriented programming (OOP) is not without its critics.

OOP puts emphasis on data and objects (in our case, buttons, sensors, displays, and the related data). That's good. But OOP takes emphasis away from thoughtful use of algorithms and procedures.

While OOP is conceptually valuable for programmers, simplifying their work, it can lead to a chaotic and wasteful flow of code execution, commonly called "bloat." When OOP programs suffer from bugs the call is often to throw more hardware (memory, clock speed) at the problem without serious thought of figuring out exactly how the code is executing and what is using the hardware resources. This is not a competitive method in an embedded system. (But OOP with "more hardware" has been profitable in the general-purpose and smartphone markets.)

https://en.wikipedia.org/wiki/Object-oriented_programming#Criticism

For us, responding to the criticism means being aware of. . .

--density limit
--latency and resolution limits
--interval restrictions
--critical regions in code
--deadlock

To be seriously competitive with large-scale products one cannot just throw more hardware at the problem. Just the opposite. . . Getting the best out of limited hardware matters. This is very different from general computing.

https://commons.wikimedia.org/wiki/File:Tiobeindex.png

10