Tutorial: Interrupt-Driven Event-Counter on the Raspberry Pi

--D. Thiebaut (talk) 19:57, 23 July 2013 (EDT)

The purpose of this tutorial is to illustrate how to implement a user-level interrupt in C on a Raspberry Pi to count events. You may want to start with the very first tutorial of this series, here.

Contents [hide]

Note that the interrupt service routine implemented by the WiringPi library has a huge overhead. In situations where a high frequency signal is connected to the pin that generates the interrupt, you may want to simply poll the pin using the method illustrated here. Sampling frequencies of 12MHz can be achieved that way.

Getting the Environment Ready

Install the WiringPi Library

- Get the WiringPi library from drogon.net
- Follow the directions on the Web site to download to the Pi. In my case the Pi is connected to my Mac through an ethernet cable, so I downloaded the tgz archive from https://git.drogon.net/?p=wiringPi;a=summary and sftp it over to the pi. I renamed the actual library to **wiring.tgz**, which is easier to type than its actual name.

(download wiringPi-cbf6d64.tar.gz)

```
sftp pi@169.254.0.2
sftp> pwd
Remote working directory: /home/pi
sftp> put wiring.tgz
Uploading wiring.tgz to /home/pi/wiring.tgz
wiring.tgz                                                         100%  108KB 107.8KB/s   00:00
sftp> quit
```

- Connect to the RPI and build the library

```
 tar -xzvf wiring.tgz
 cd wiringPi-cbf6d64/
 ./build
wiringPi Build script
=====================


WiringPi Library
make: Warning: File `Makefile' has modification time 1.4e+07 s in the future
[UnInstall]
[Compile] wiringPi.c
[Compile] wiringSerial.c
...
[Compile] drc.c
[Link (Dynamic)]
[Install Headers]
[Install Dynamic Lib]
make: warning:  Clock skew detected.  Your build may be incomplete.

WiringPi Devices Library
make: Warning: File `Makefile' has modification time 1.4e+07 s in the future
[UnInstall]
[Compile] ds1302.c
...
[Link (Dynamic)]
[Install Headers]
[Install Dynamic Lib]
make: warning:  Clock skew detected.  Your build may be incomplete.

GPIO Utility
make: Warning: File `Makefile' has modification time 1.4e+07 s in the future
[Compile] gpio.c
[Compile] extensions.c
[Compile] readall.c
[Link]
make: warning:  Clock skew detected.  Your build may be incomplete.
make: Warning: File `Makefile' has modification time 1.4e+07 s in the future
[Install]
make: warning:  Clock skew detected.  Your build may be incomplete.

All Done.

NOTE: This is wiringPi v2, and if you need to use the lcd, Piface,
  Gertboard, MaxDetext, etc. routines then you must change your
  compile scripts to add -lwiringPiDev
```

- Add the new library to the libray path, as explained in the INSTALL file of the wiringPi distribution.

```
sudo nano /etc/ld.so.conf
```

and add the following line to it:

```
/usr/local/lib
```

- Tell the system to configure the libraries:

```
sudo ldconfig
```

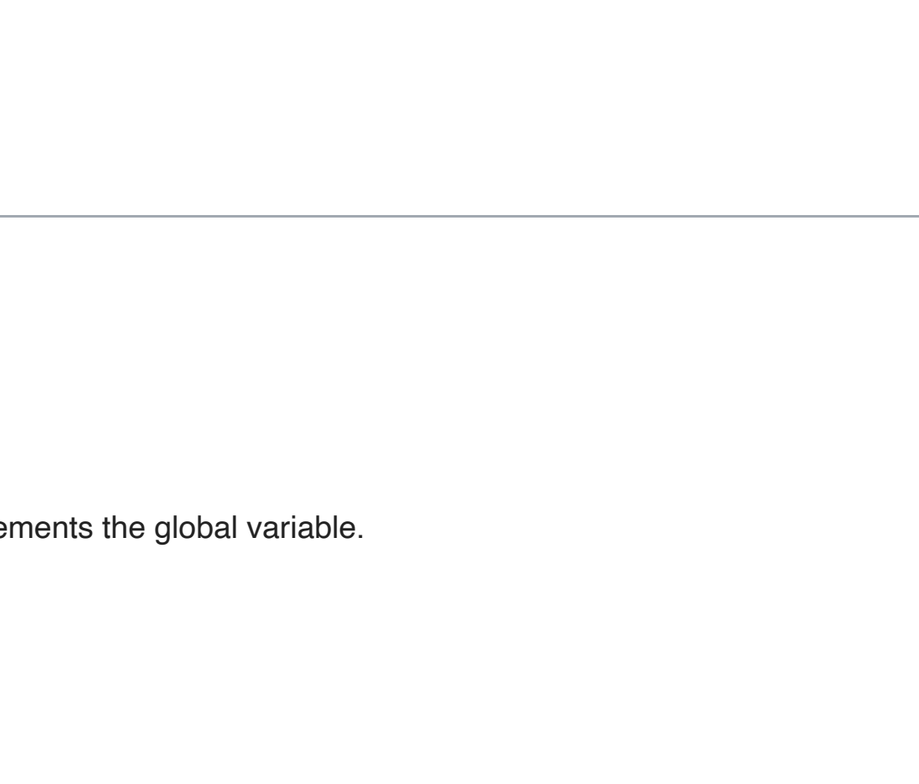
Hardware Setup

- Our hardware setup is the same as that presented in Introduction to accessing the Raspberry Pi's GPIO in C++ (Linux Way / SYSFS) on hertaville.com. We have only implemented the switch and will rely on **printf** statements to indicate whether activating the button triggers the ISR or not.
- We use PIN 17 of the GPIO, available on the RPI 26-pin connector Pin 11.
- Connecting the momentary switch is simple:
  - 1 lead connects to GND
  - its other lead connects to two places:
    1. to one side of a a 10KΩ resistor, the other side of the resistor to 3.3V
    2. to Pin 17 of the GPIO

Interrupt Service Routine

Picking the Right Constant for GPIO Pin 17

- The wiringPi library labels GPIO Pin 17 as Pin 0 (see drogon.net), as illustrated in the table below taken from their Web site:



wiringPi Pin	BCM GPIO	Name	Header	Name	BCM GPIO	wiringPi Pin
–	–	3.3v	1 | 2	5v	–	–
8	R1:0/R2:2	SDA0	3 | 4	5v	–	–
9	R1:1/R2:3	SCL0	5 | 6	0v	–	–
7	4	GPIO7	7 | 8	TxD	14	15
–	–	0v	9 | 10	RxD	15	16
0	17	GPIO0	11 | 12	GPIO1	18	1
2	R1:21/R2:27	GPIO2	13 | 14	0v	–	–
3	22	GPIO3	15 | 16	GPIO4	23	4
–	–	3.3v	17 | 18	GPIO5	24	5
12	10	MOSI	19 | 20	0v	–	–
13	9	MISO	21 | 22	GPIO6	25	6
14	11	SCLK	23 | 24	CE0	8	10
–	–	0v	25 | 26	CE1	7	11
wiringPi Pin	BCM GPIO	Name	Header	Name	BCM GPIO	wiringPi Pin

- Our switch is connected to Pin 17 of the GPIO, so we'll use **0** to refer to this pin when using the **wiringPi** library.

ISR Code

The code for the Interrupt Service Routine is given below. Its operation is simple:

1. it defines Pin 0 (GPIO Pin 17) as the pin which will receive the events
2. it declares a global variable that counts events
3. it defines a function that will be called by the interrupt triggered by Pin 0. The function simply increments the global variable.
4. it initializes the wiringPi library
5. it attaches the function to the interrupt
6. it starts an endless loop that displays the global variable and then clears it.
7. it repeats the previous step after a 1-second wait.

```
/*
isr4pi.c
D. Thiebaut
based on isr.c from the WiringPi library, authored by Gordon Henderson
https://github.com/WiringPi/WiringPi/blob/master/examples/isr.c

Compile as follows:

    gcc -o isr4pi isr4pi.c -lwiringPi

Run as follows:

    sudo ./isr4pi

 */
#include <stdio.h>
#include <string.h>
#include <errno.h>
#include <stdlib.h>
#include <wiringPi.h>


// Use GPIO Pin 17, which is Pin 0 for wiringPi library

#define BUTTON_PIN 0


// the event counter
volatile int eventCounter = 0;

// -------------------------------------------------------------------------
// myInterrupt:  called every time an event occurs
void myInterrupt(void) {
   eventCounter++;
}


// -------------------------------------------------------------------------
// main
int main(void) {
  // sets up the wiringPi library
  if (wiringPiSetup () < 0) {
      fprintf (stderr, "Unable to setup wiringPi: %s\n", strerror (errno));
      return 1;
  }

  // set Pin 17/0 generate an interrupt on high-to-low transitions
  // and attach myInterrupt() to the interrupt
  if ( wiringPiISR (BUTTON_PIN, INT_EDGE_FALLING, &myInterrupt) < 0 ) {
      fprintf (stderr, "Unable to setup ISR: %s\n", strerror (errno));
      return 1;
  }

  // display counter value every second.
  while ( 1 ) {
    printf( "%d\n", eventCounter );
    eventCounter = 0;
    delay( 1000 ); // wait 1 second
  }

  return 0;
}
```

Compilation

- Compile the code above against the WiringPi library as follows:

```
gcc -o isr4pi isr4pi.c -lwiringPi
```

Test

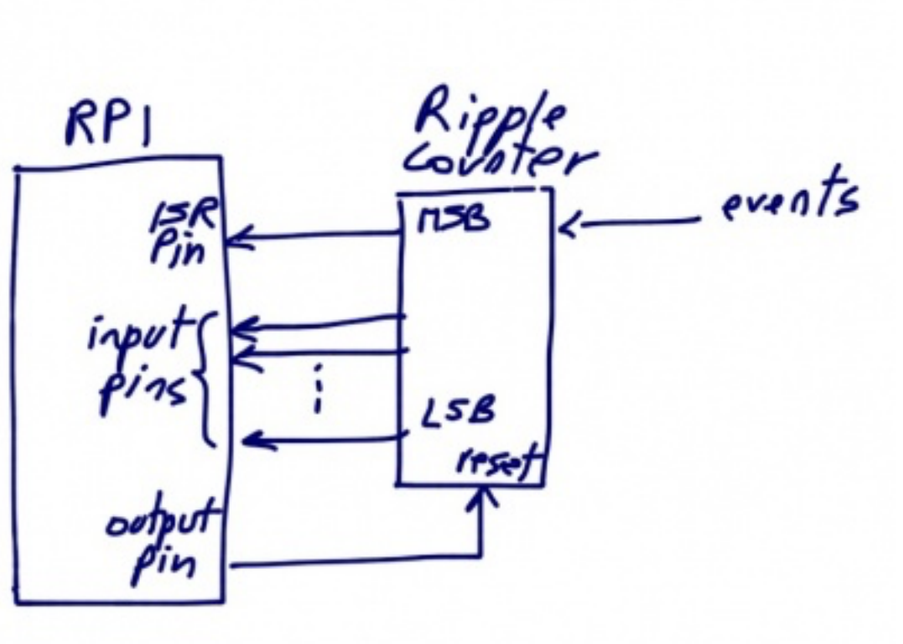
Now comes the time to test the setup. We launch the program on the RPI and press the button several times. Note that because there is no debouncing on the button, spurious spikes are generated when we activate it and the number printed on the screen is quite a bit larger than how often we press the button. For example, pressing the button once generates counts of 2, 3, and even 6. This is not a flaw. Just a property of mechanical switches. There are several good solutions on the Web ways to debounce switches.

```
pi@raspberrypi ~ $ sudo ./isr4pi
0
0
0
14
13
10
2
3
2
6
2
^C
pi@raspberrypi ~ $
```

Note that you have to Control-C out of the program to stop it...

Increasing the Resolution of the Event Counter

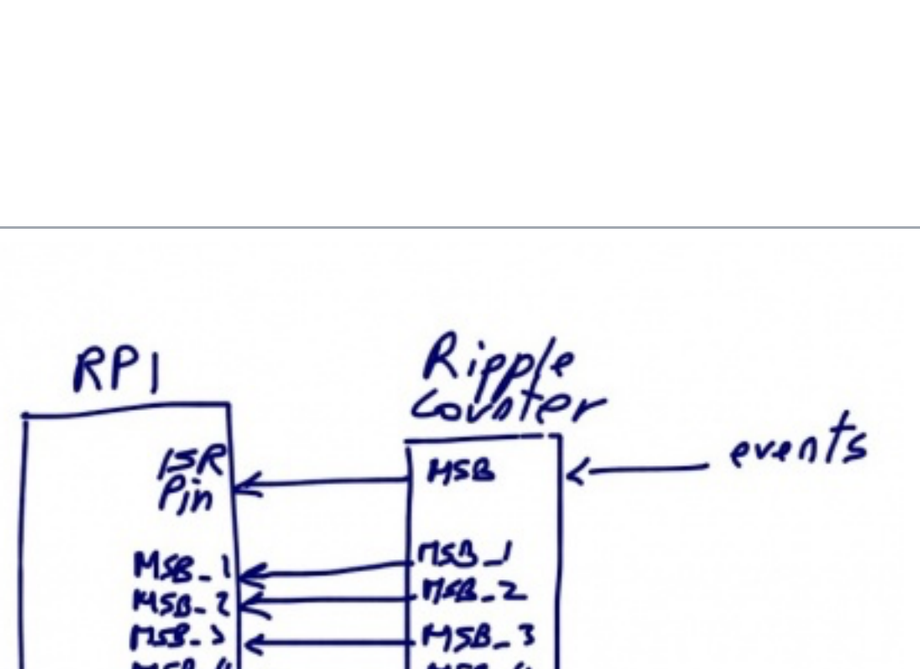
- Raspberry Pi users have reported the important latency with which user-level interrupts are serviced. One user reports as much as 75 μs latency on some of the interrupts, and an other indicates that dynamic refresh of the RAM may also create longer delays.
- One solution is to increase the priority of the interrupt, which requires making it a *kernel-level* interrupt and recompiling the kernel, which is not for the faint-of-heart.
- Another solution that is inexpensive and requires just one external chip can save the day and still work with user-level interrupts while providing better accuracy. The figure below illustrates the concept:



Theory of Operation

- The ripple counter is a CMOS binary counter, possibly 10 or 12 bit counter. The events activate the clock signal which in turns increments the counter.
- The top *n* most-significant bits of the counter are connected to GPIO input pins.
- The true MSB of the counter, the bit that changes the least frequently is connected to a GPIO pin setup as an interrupt pin.
- The ripple counter's *reset* pin is connected to a GPIO output pin
- A typical algorithm for counting events would operate as follows:
  1. define a function as in this tutorial that increments a global variable*eventCount*.
  2. attach this function as an ISR to the pin attached to the MSB of the ripple counter. Make the ISR called when the GPIO pin goes *low*.
  3. activate the *reset* pin on the counter, setting its contents to 0.
  4. wait some period of time, say 1 second if we're interested in a frequency counter.
  5. after this delay, read the *n-1* MSBs of the ripple counter. Convert to their binary equivalent.
  6. multiply the value of the global variable *eventCount* by 2^k, where k is the number of bits of the ripple counter. For example, if the counter is a 10-bit counter, its MSB is going to go from 1 to 0 every 1024 clock ticks, so if the ISR is attached to the low transitions of the MSB, every increment of the global variable represents 1024 events.
  7. if the number of MSB bits read from the ripple counter (excluding the top MSB) is k, with $k < n$, then convert them to a binary number m and multiply it by $2^{(n-k)}$. Add the resulting number to the number obtained in the previous step. The total is the approximate number of events that the ripple counter saw on its clock input.
  8. Go back to Step 3.

Case Example



- A 10-bit ripple counter (for example the 74HC4040) is used to interface a digital signal that is fed to its clock input
- The MSB of the counter representing 2^{10} is connected to the pin of the RPI acting as the interrupt input.
- The next top 4 output bits of the counter, referenced *MSB_1*, *MSB_2*, *MSB_3*, *MSB_4*, and corresponding to weights 2^9, 2^8, 2^7, and 2^6, are connected to 4 input pins of the RPI, which we also refer to as *MSB_1*, *MSB_2*, *MSB_3*, and *MSB_4* in the code.
- Whenever it is time to compute the number of events received by the counter, we simply add up these quantities:

```
numberOfEvents = eventCounter * 1024 + MSB_1 * 512 + MSB_2 * 256 + MSB_3 * 128 + MSB_4 * 64;
```

Note that not connecting the least-significant bit of the counter to the RPI means that your accuracy is fixed to 2^{n-k}, and the error is always positive. Any real count of events between 0 and 63 will be reported as 0. Any real count between 64 and 127 as 64. In our case our measurements will be off by at most 2^{10-4} = 64. Of course, the higher the frequency of the events, the lower the relative error.

Comments

The *n*-bit ripple counter decouples the Raspberry Pi from the events, so that an interrupt is generated only every 2^n events. Being able to read some of the other bits of the ripple counter allows one to get a resolution finer than just having a multiple of 2^n.

Assume that the maximum frequency of events that a Raspberry Pi can service via a user-level ISR is 10 KHz (as some users have reported), then a 10-bit ripple counter will push this limit to 10 MHz. A 14-bit ripple counter to 160 Mhz. Two cascaded 10-bit counters to 10 GHz (assuming one can find counters that operate at that frequency!).

Updating to Raspberry Pi3B

This section is still under debugging... Use at your own risk...

```
cd temp
ls
git clone --depth=1 https://github.com/raspberrypi/linux
uname -r
git clone --depth=1 --branch rpi-4.14.y https://github.com/raspberrypi/linux
cd linux
KERNEL=kernel7
make bcm2709_defconfig
make -j4 zImage modules dtbs
sudo make modules_install
sudo cp arch/arm/boot/dts/*.dtb /boot/
sudo cp arch/arm/boot/dts/overlays/*.dtb* /boot/overlays/
sudo cp arch/arm/boot/dts/overlays/README /boot/overlays/
sudo cp arch/arm/boot/zImage /boot/$KERNEL.img
```

Categories: Tutorials | Raspberry Pi