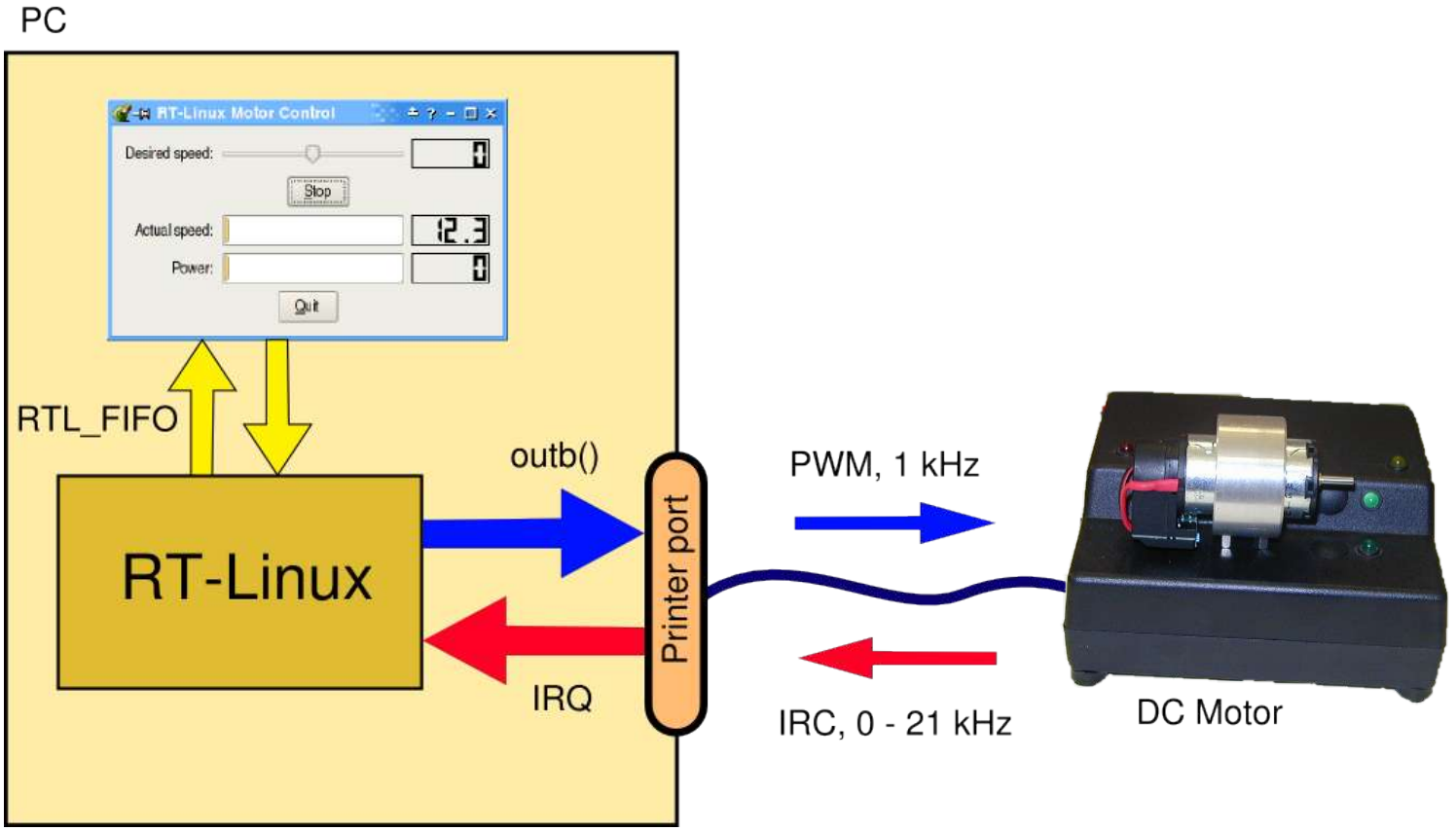
DC Motor Controller in RT-Linux

The goal is to create a servo-controller (to control the speed of the motor).



Steps to Create a Controller

1. Create basic RT-Linux module.
2. Try to rev up the motor at full speed.
3. Write a thread for PWM generation (period 1 ms)
4. Write an IRQ handler (position measuring).
5. Write a thread for speed measuring.
6. Design a controller (PID) for the speed control.
7. Allow communication with user-space.
8. Write a user-space interface for the controller.

A Basic RT-Linux Module

- The same kind of module which Linux uses for drivers etc.
- The code runs in the kernel-space (shares all code and data with the Linux kernel).
- Source: simple.c
- Makefile for compilation

```
all: simple.o

include /usr/rtlinux/rtl.mk
include $(RTL_DIR)/Rules.make
```

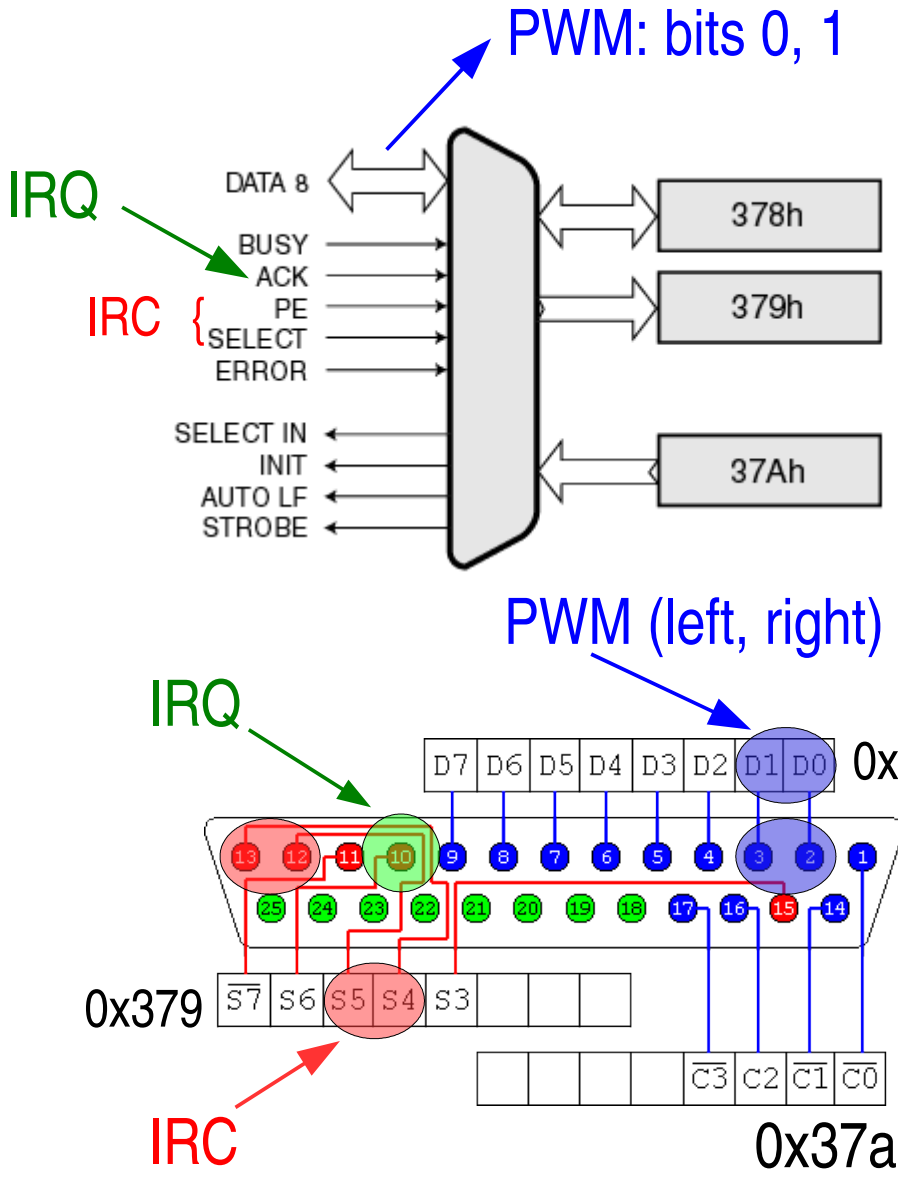
Running the application:

```
shell# insmod simple.o
```

```
#include <linux/module.h>
#include <linux/kernel.h>

int init_module(void)
{
        printk("Init\n");
        return 0;
}

void cleanup_module(void)
{
        printk("Cleanup\n");
}

MODULE_LICENSE("GPL");
```

Parallel Port

- Motor rotation:
  - left: `outb(1, 0x378);`
  - right: `outb(2, 0x378);`
- IRC signals:
  - `inb(0x379);`



Periodic Threads

```
#define MS (1000000)

void *thread_func(void *arg)
{
    pthread_make_periodic_np(pthread_self(), gethrtime(), 2*MS);
    while (1) {
        /* do something */
        pthread_wait_np();
    }
    return NULL;
}

int init_module(void)
{
    pthread_t thr;

    pthread_create(&thr, NULL, &thread_func, NULL);
    return 0;
}
```

start time

period

wait for the start of the next period

Thread Priorities

- Rate Monotonic Priority Assignment
  - the lesser task period the higher assigned priority
- **In RT-Linux:** The higher number the higher priority

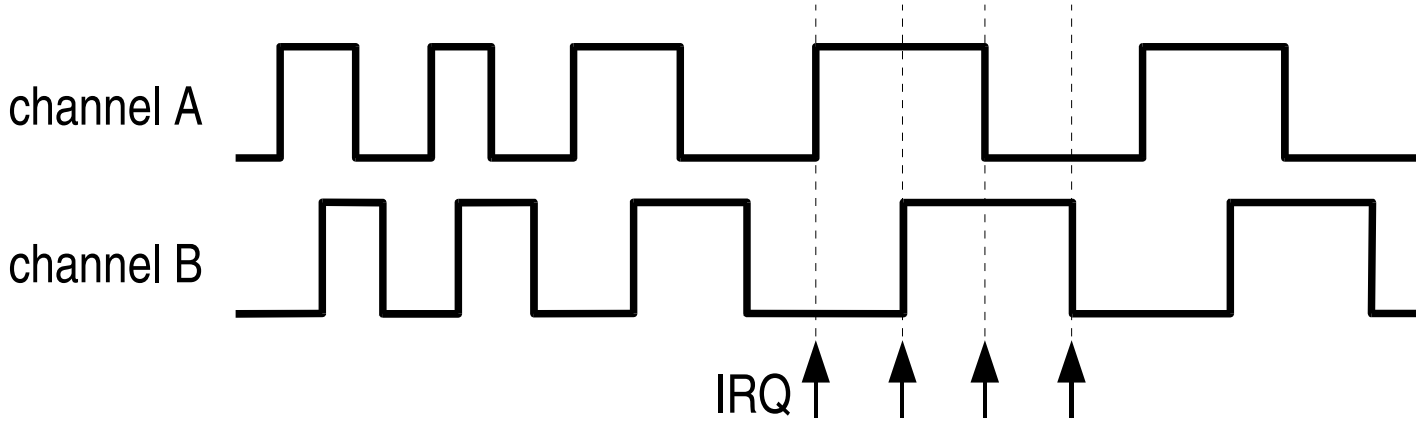
```
int init_module(void)
{
    pthread_attr_t attr;
    struct sched_param param;

    pthread_attr_init(&attr);
    param.sched_priority = 1;
    pthread_attr_setschedparam(&attr, &param);
    pthread_create(&thr, &attr, &thread_func, NULL);
    return 0;
 }
```

IRQ Handling

- Parallel port: IRQ 7
- Interrupts reception should be reenabled in the handler!

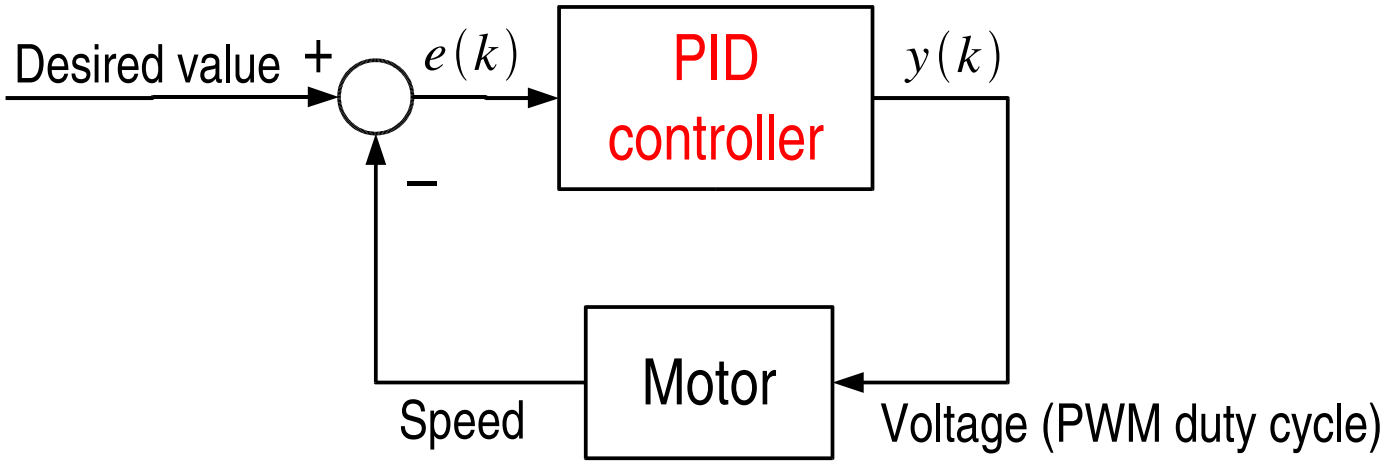
```
unsigned int irq_handler(unsigned int irq, struct pt_regs * regs)
{
    /* do something */

    rtl_hard_enable_irq(irq);
    return 0;
}

status = rtl_request_irq(irq_number, irq_handler);
```

Signals From an IRC sensor



- Whenever the value of any IRC sensor channel changes, electronics in the motor generates an IRQ.
- The motor contains IRC with 100 pulses per rotation and there are 4 IRQs per one pulse.

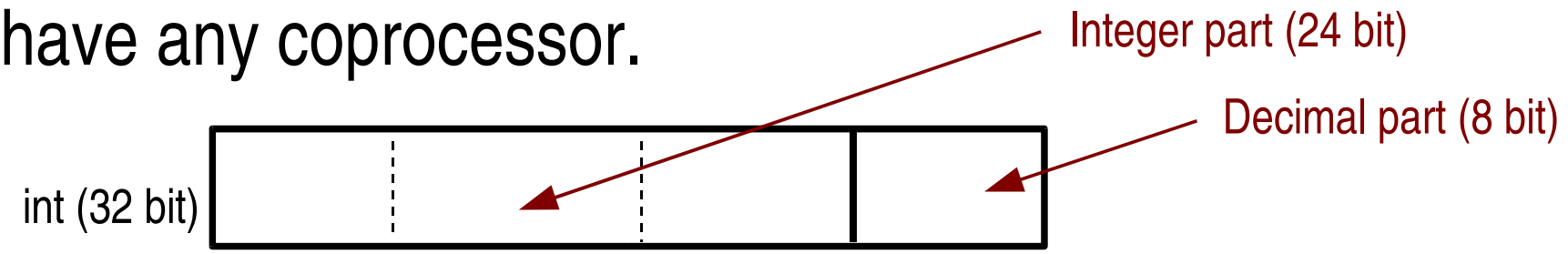
PID Controller



$$y(k) = P \cdot e(k) + I \cdot \sum_{i=0}^{k-1} e(i) + D \cdot (e(k) - e(k-1))$$

Fixed Point Arithmetic

- We need to use **decimal numbers** in calculations
- For this simple task we don't need to use a **mathematical coprocessor**. Smaller processors don't have any coprocessor.



- 5.0 ~ 0x500, 2.5 ~ 0x280
- Addition: 5.0 + 2.5 ~ 0x500 + 0x280 = 0x780 ~ 7.5
- Multiplication: 5.0 * 2.5 ~ 0x500 >> 4 * 0x280 >> 4 = 0x50 * 0x28 = 0xC80 ~ 12.5

RT FIFOs

- Communication between RT-Linux and user-space.
- Unidirectional communication, for bidirectional communication we need two fifos.

RT-Linux side

```
#include <rtl_fifo.h>

int fifo = 0;

rtf_create(fifo, 1000);
rtf_create_handler(fifo, &read_handler);

retval = rtf_put(fifo, &variable, sizeof(variable));

int read_handler(unsigned int fifo)
{
    int reference;
    rtf_get(fifo, &reference, sizeof(reference));
    return 1;
}
```

We use the FIFO number 0

RT FIFOs – User-Space Side

- From the user-space the FIFO looks like an ordinary file.

```
int i, j;

if ((fifo_out = open("/dev/rtf0", O_WRONLY)) < 0)
{
   perror("/dev/rtf0");
   exit(1);
}

write(fifo_out, &i, sizeof(i));

read(fifo_in, &j, sizeof(j));
```

We use the FIFO number 0