# Robocon Electronics Quickstart Guide





Ki Chi Keung
Wong Long Sing (Sam)

# 0. Preface

As you might have known already, this crash course is dedicated to you, those who are ready to enjoy the year-long Robocon competition life. I tell you, it's going to be tough, the unofficially recognized workload is comparable, or even greater, than that of final year project, so prepared.

Most of the knowledge you need to learn to be a good player in this project is mostly out of university syllabus, or it is not going to be covered until Year 3, so do not blame us and yourself when you face something you had never learn. Yes! There are a lot of things in this Robocon project you had and will never learn in the university, but I tell you, they are not that hard.

In order to make you quickly familiar with the low level programming, we have prepared this crash course and the starter board, for you. As you might have known already, the main character of our starter board is the microcontroller, Atmel's ATMega128 to be exact, it can connect most of the hardware devices we used in previous competition. This starter course is focus to make you familiar with the ATMega128, and teach you most feature of it by experimenting with those devices.

It does not mean you have known everything and can win the game even after understanding every material of this course or the every aspect of datasheet and become a master of electronics. Why? Because our ultimate goal is to develop superb winning algorithm, not just blind coding. Electronics and programming are just the basic, we know that well, everyone knows that well, it's not a secret weapon. What we are missing in the winning puzzle is the algorithm. However, talking algorithm is meaningless without any programming knowledge and a good hardware to run on, so we designed this started course.

We, Chi Keung and I, were involved in previous Robocon competitions, from the very first one in Hong Kong to today. We know it may be a bit late to realize the needed of a well structured starter tutorial, nevertheless, here it is. Please enjoy! (Clap! Clap!)

Sam Wong

# 1. Microcontroller

Our robocon needs some sort of intelligent. It may be common for the robot to be fully controlled mechanically, when computers are just for big corps and scientist, it's obviously doesn't fit for our Robocon project. On the other hand, a full-blown PC is overkill for this project, because of its size, energy consumption and complexity. We need some form of direct control so that we can command the motor and sensors to do their work with very high precision. Microcontrollers come and save our life.

It's very common to see microcontrollers, or MCU, around, and in fact the line between MCU and PC is becoming more blur today. Or I should say the CPU of today becomes smaller and faster, which used to be a distinctive nature of MCU. Take a look at today cell-phone, or your wireless router, it has a CPU inside, yet the size and energy consumption is similar to that of microcontroller.

From Wikipedia, the definition of Microcontroller is that "It is a type of microprocessor emphasizing self-sufficiency and cost-effectiveness...A typical microcontroller contains all the memory and interfaces needed for a simple application, whereas a general purpose microprocessor requires additional chips to provide these functions". To me, MCU is graded as MCU because of its selling price. =P

## *i. Compare to the Computer*

Most MCU has flash memory and RAM in it, so it doesn't need a harddisk and external ram to run. It is designed that it can run itself without any OS, very much similar to your computer BIOS, you flash a program to it and it just runs.

In addition, MCU usually comes with a lot of general purpose IO pins, or GPIO, while CPU almost only carries address and data bus. With GPIO, and your program run on the chip without the OS, you have direct control to pin, I am talking about a precise High or Low voltage output, strict control to the timing. These two points make MCU very suitable in controlling the low level hardware like motors and sensors.

On the other hand, because now you have direct control, you, as a programmer, is also directly in charge of every low level stuffs which is usually handled by OS. For example, there is no multitasking, nothing prevent you from segfault'ing, you have to handle the interrupt and all low level timing stuff. Plus, because of the limitation of processing power and memory, you are responsible to save every juice that the MCU has.

As a side note, we used to use a Embedded ARM - Linux in previous Robocon competition, to calculate complicated algorithm and make Wifi works. We are not going to mess with that bastard in this starting course though.

## *ii. The Hardware*

The microcontroller we will be using is ATMega128, a AVR series MCU by Atmel. You can download the datasheet in its product info page. Working in hardware level, datasheet is very important to MCU programmers, very much like API menu for software development. Also, very similar to that of software development, you

would never read the whole JAVA API menu when you start working with Java, we are also not going to go through every aspect of those 391 pages, because there are details that we don't care! Although if you are going to take care the PCB design, as well as programming the chip, like what we did, you will eventually read the whole thing. 391 pages is not that long, trust me.

MCU comes with Flash, RAM and EEPROM, which you can load your program to there (Chapter 2). It comes with GPIO, we will do some input and output experiments with that (Chapter 3). To communicate with computers, the most common way to do it is by using the UART, which is speaking the computer's COM port protocol (Chapter 4). The timer is also very handy in doing time related task (Chapter 5). Our ATMega128 also comes with some other communication protocol hardware interface, very useful in connecting other electronics device (Chapter 6).

When power is given to the MCU, to the correct pin of course, it would start working. It would start fetching the machine code from the Flash memory address 0x0000, the very beginning of the memory, and decode the command, execute it. Unless the command is to instruct the MCU to make a jump, e.g. calling a function, or in a if-then-else branch, or in a for-loop, or it is interrupted by an event (Chapter 4), the MCU will continue process it one by one sequentially, very precisely.

The speed to process each command is very well-defined. In fact, every time it executes the segment of code, it takes exactly the same amount of time to do so. It is actually true to your PC CPU, but not true to PC programming. Why? Because in PC programming, there are OS and multitasking, the CPU is not dedicated to process your program. In MCU programming, the execution unit is 100% obey to your program, that's why the timing becomes deterministic.

The processing speed is defined by the hardware, and the reference clock. Just like your PC CPU, the ATMega128 is speed graded too. It can run at 16Mhz max. Okay, nothing prevent you from overclocking it, PC CPU used to have overclock prevention too, but if you really does, it may becomes malfunction, may not execute the instruction very correctly, or even may burn up itself in extreme case.

The MCU/CPU itself doesn't know the time, they all rely on one very important hardware to tell time—the crystal oscillator. If you are more familiar to the word quartz because of the Rolex watch, I can tell you that the oscillator we are using is genuine quartz crystal. And it's actually a very cheap thing that's included in every electronic gadgets.

Take home message, you have total control over the time and pin input/output.

## *iii. The Software*

Recall that the MCU runs the program in the flash memory, you may ask, how? The program in flash memory is actually the machine codes, a stream of binary value which is stored in the flash memory and read as high or low voltage signal by MCU and decoded by the AND, OR, NAND gates and transistors. We don't do programming in machine code, general speaking the lowest level of programming we would do is in the assembly level. Assembly language is very similar to the machine codes used in the chip, every assembly command, or Op Code, will be

eventually translated into the binary values, as specified in the datasheet, so we write in English and the compiler translates it into binary value.

In this project, we do not write our program is assembly, that's only needed usually when we are demanding a very timing critical response. We do our programming in C, and use GCC to compiler our code into assembly, and eventually into machine code. However, because of the very nature of MCU programming, the quality of the C program directly affect the performance of it, so in some sense we have to code it carefully with everything well-thought and optimized to be assembly friendly.

# 2. The Hardware and Tools

In making the tutorial material, it comes to chicken and egg problems. We understand that without one-person-one-hardware, your interest in learning would be tends to zero. But to make a suitable hardware, or PCB, for competition, we first need a basic algorithm and idea to understand how we are going to do thing and play the game. Previously we designed were using a powerful set board, but the system is too complicated for every starter to pick up. So we made this starter board, and it's dedicated for training purpose. It's for you, so please take good care of it.

By the way, please let me know if us know if you think you understand this starter document very well, you might actually be ready to do the real hardcore thing!

## *i. Our Starter Board*

There are only a few chips on the board, well in fact it's true even for the boards used in competition. The square shaped in the middle is the ATMega128. The rectangular one with a relatively big heatsink is the DC-DC converter, responsible to convert high voltage to a 5V standard working voltage. Another long rectangular one with pins on two side is the MAX232, responsible in converting the signal level from UART to RS232, that is the COM port protocol your computer speaks, to facilitate communication between the computer and the MCU. Others are just passive components and connectors to make things work, no analog or mathematic calculation involved.

You will need to refer to the schematic from time to time, you will at least need to know which connectors connected to which pins.

To power up the starter board, connect a 6V to 30V DC source. Don't mess with the positive and negative, there is just physical barrier to prevent you from doing that, but no electronic design to prevent you from frying the board.

## *ii. Development Tools*

To compile the C code and flash your compiled machine code binary file to the MCU, you need the GCC, the GNU C compiler, and related toolchain, i.e. the libraries and helper programs. We used and will use WinAVR for this course and project. It is a nice Windows package of gcc and avr-libc, downloadable at http://winavr.sourceforge.net/.

avr-libc is a library that provides AVR specific functions, included in WinAVR and the document is avaiable at http://www.nongnu.org/avr-libc/user-manual/. You will need to reference its *Library Reference* from time to time.

AVRStudio, the development tools by Atmel, downloadable at http://www.atmel.com/dyn/products/tools_card.asp?tool_id=2725 is nice to have too. Basically we are going to use its STK500 plugins only, but not using it as development.

A nice text editor like vim would be cool, but strictly optional =P. The one comes with WinAVR is cool enough.

## *iii. Bootloader*

To flash a program, you need to erase the contents of the flash memory first. In old days you need some high energy to do so. In real old days, you have to use a UV light box to erase the program. In not so old days, you can program it with a programmer, a.k.a. parallel programming, it generally needs 12V and around 16pins connection, to erase the chip content, however if the MCU is soldered on board, because 12V and those connections would interfere other component, this method is not practical and there was no mean to change the program at all, nevertheless, the AVR series still support this old day method. Today, the most common form of programming is ISP, a.k.a. in system programming or serial programming, which only requires 5V and some pins to do the work, and of course AVR does support this too. An even more advanced form of programming is flashing over UART, or RS232, directly from the com port of your computer, which unfortunately the AVR does not support.

Despite of that, our AVR does support bootloader programming. In short, it allows the program on flash rewriting the flash. So once a starter program, or bootloader, is flashed into the chip, we can do what we want. People had prepared bootloaders for flashing over UART, we adopted one of them. So now you can enjoy flashing the program just with your COM port, without the need of special hardware to do ISP. All the boards are loaded with the bootloader.

## *iv. Compilation and Development*

Compile is easy. `Tools -> Make All` in *Programmers Notepad* or `make all` under command prompt is all you needed.

To flash the MCU, holds the *PROG* button while resetting it, then it will enter Bootloader mode when it boots. The LED would flash periodically when the bootloader is ready. By using `Tools -> Program` or `make program` under prompt, you would have the program flashed to the board. Of course you need to connect the serial cable first. In case your COM port number allocated is different from that of the one specified in the Makefile, you have two choices: 1. Modify the Makefile, 2. Change the port number in Hardware Manager.

# 3. LED, Buzzer and Button

In this chapter, we will cover the basics of simple programs and GPIO operations. I bet most of you have the experience with the LED, buzzer as well as button. They are just very simple input and output device, requires no special protocol to control and they just work.

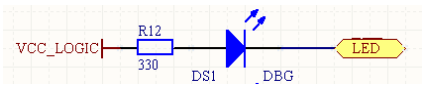
## *i. Schematic Reading*

Still with me? Now we want to turn on the LED, what should we do? We should first check the schematic to see where the LED is connected to. Look at the *Core Board* **schematic** first, the overview of our board. I hope you can find the word *LED* in the *DEBUG* **sheet**. This leads you to the *DEBUG* schematic, Bingo! Can you spot two LEDs there? *DS1* and *DS2*. Tell me which one you can control, and how. Try to locate them on the board. Given these information, *DS1*-the **Designator** and *DBG*-the **Comment**, you should also be able to find that LED on your board.

You should have figured out that *DS2* is simply a power on indicator. It has fixed *VCC* and *GND* connection. So our hop goes to the *DS1*.

The anode of the LED is connected to the VCC, and the cathode is connected to the *LED* **port**. Let's see where that port is connected to. Let's Go back to the *Core Board* schematic, the word LED is actually an schematic **sheet entry**, i.e. the LED **port** is logically connected to this LED **sheet entry**. The *E3* you see on the wire is the **net** name, conductor of the same net would be connected together. Let see what else is connected to this net, as you should might have expected already, it should be connected to our MCU.

Take a look at the *MCU* sheet at the upper corner. *E[0..7]* is a **bus** which actually means it contains 8 net, namely E0, E1, E2...E7. Now follow this and go to the *MCU* schematic. I bet you should have located the net *E3*, it's pin 5 of the ATMega128. Got the idea? Can you figure out how to turn on and off the *DS1* LED by now?



Recap - port *LED* connects to the net *E4* which is connected to the pin 5 of ATMega128.

The LED would be turned on if there is current flow through, current flows where there is a potential difference. If we can put the port *LED* at 0V (Low), the LED would be turned on. On the other hand, if we can put it at 5V (High), it would be turned off because the potential difference across is 0V. By the way, the function is active when you give it a 0V, this design is called **active low**. For those which activated when High is given, it's **active high**. Usually, a bar would be added to the logic name when it is active low, but this is not always the case (due to the laziness of the specification writer).

Then, how should we set the voltage of the pin 5? Now it's time to read the specification. Page 1 of the spec shows that we have 53 GPIO lines to use, E2 is one of them, there are **port** A, B, C, D, E, F x 8 pins and port G x 5 pins. Page 2 shows the pin location. You need this information because sometimes you would

really want see, or probe, the voltage with multimeter.

## *ii. Special Function Register*

A MCU behavior is mainly controlled by a special register simply called Special Function Registor (SFR), page 365 of the ATMega128 spec has a comprehensive list of register. From the view of programmer, reading and writing a register just work exactly the same as reading and writing a variable. Each register in AVR is 8-bit wide (It's a 8-bit MCU after all) and each SFR has a name, if you want to assign a value 0x12 to SFR named XXX, you can write

```
XXX = 0x12;
```

And to read, let say assign it to the variable *c*,

```
unsigned char c = XXX;
```

By the way, 0x12 is the **hexadecimal** representation of a number, which is the same as 18 in decimal in this case. You will see why we want to stick with hex in a moment.

## *iii. ATMega128 Specification Reading*

Now we want to control GPIO lines, we determined that we have to mess with some specific SFR. Go back to the spec, bookmark *I/O Ports* seems match our question. Because we want to have a quick answer, we would skip all the overview and description, all the way to the *Register Description for I/O Ports*.

Ok, we see a list of PORTx, DDRx and PINx, we have no idea what the heck are these. Sometimes the spec would explains what does each bit do bit by bit, e.g. Page 53. It seems that the PORTx stuff are just trivial that does not deserve any further explanation. We are forced to read the real thing.

Besides **register description**, the second useful information is **truth table/matrix**, then **timing diagram** and **schematic**. Just with these eye candies, you should get the general idea of the covered topic, after that we go to the main text. Remember, our current focus is to find out the solution to output a High voltage and Low voltage on a specified pin, other minor behavior could be ignored at this moment. Reading them is left as an exercise for the readers.

Table 25 on page 67 should catch your attention. See? Output High, Output Low, DDR*xn*, PORT*xn*! Bingo!

Let us spend a few minutes here to explain the essential:
GPIO pins of the ATMega128 can be configured into four mode, let x is A-G, n is 0-7, also assume PUD in SFIOR is 1, which is the boot-up default.

1. DDR*xn* = 1, PORT*xn* = 1: Voltage driver driving High.
   The MCU would use a CMOS to drive the pin up to the VCC.
2. DDR*xn* = 1, PORT*xn* = 0: Voltage driver driving Low.

The MCU would use a CMOS to drive the pin down to the GND.

3. DDR*xn* = 0, PORT*xn* = 1: Voltage driver disabled, *pull-up* resistor enabled. The driving CMOS is turned off, but turns on a PMOS that connected to the VCC and a resistor to the pin. This would define a Weak High on the pin, i.e. it will stay at VCC if it is not driven otherwise.
4. DDR*xn* = 0, PORT*xn* = 0: Voltage driver disabled, no pull-up enabled. The driving CMOS and pull-up system is turned off. If it is not defined by other chips, the pin is *float* and we cannot say anything about the voltage of the pin. It is undefined.

One point you should keep in mind, we would usually always want to define the voltage of every pin, and we want everything defined. Refer to page 68, *Unconnected Pins*, it is best to be avoided because the undefined voltage could be in the embarrassing range that could drive the gate of both PMOS and NMOS of the input circuit. Besides, driving a high or low does not consume any power theoretically, because the voltage is static and no current flow through and hence no power loss. Power is needed only when the CMOS switches.

Each bit in the PORT, DDR and PIN is corresponding to a physical pin. The most significant bit, or **MSB**, is identified as bit 7, and **LSB** is identified as bit 0. Now we know in order to turn on the *DS1* LED, we want to have bit 2 of DDRE set to 1, and bit 2 of PORTE to 0 (Remember that it's active low?). Set bit 2 of PORTE to 1 to turn it off.

As a side note, keep in mind that if you want to read the logic (High or Low) of the pin, you should use save the value of register PIN*x*, instead of PORT*x*. Reading PORT*x* would only let you know the current pin mode, but not the pin logic detected. And there is no way to detect whether a pin is float or not, as explained that the voltage is undefined in that case.

## *iv. The Very First Program and Bitwise Operation*

How can I set a bit of a byte without affecting other bit? We usually want to modify a specify bit only, without affecting the operation of other bytes. Now you need bitwise operation. You MUST read the http://en.wikipedia.org/wiki/Bitwise_operation if you have no idea about this.

The avr-libc defined a **macro** _BV(*n*) == 1 << n, which is very useful. To get you familiar with the bitwise operation, let's take a look at the following statements. They are evaluated as true in C.

```
0x01 == _BV(0)
0x02 == _BV(1)
0x04 == _BV(2)
0x08 == _BV(3)
0x40 == _BV(6)
0x80 == _BV(7)
0x12 == 0x10 | 0x02
0x03 == 0x01 | 0x02   // In binary, 0b0011 == 0b0001 |
0b0010
0x55 == ~0xAA
_BV(6) | _BV(5) == (_BV(7) | _BV(6) | _BV(5)) &
~_BV(7)   // Precedences: ~, &, |
```

Now do you understand why we usually want to stick with hex? because it has

direct relationship with their binary equivalent.

Together with register operation, the following writing style is very common

```
DDRE |= _BV(2);       // DDRE = DDRE | _BV(2);, i.e. To set
bit 2
PORTE &= ~_BV(2);     // PORTE = PORTE & ~_BV(2); i.e. to
unset bit 2
```

Did I just mention the way to set and unset a given bit in a byte without ever touching others or knowing others bit values? This would actually translate into one very fast assembly operation which is cool!

Now let's write a simple program, a desktop-hello-world-equivalent! Start from the code skeleton provided, look at the main cpp file, it just includes some essential headers file and contains main subroutine. Same as writing C for desktop, the MCU start running from `main()`, but please note that return from `main()` is meaningless, because the CPU has nothing else to do.

Now insert the correct DDRE and PORTE statements. Let's try it out!

## *v. Deterministic Speed*

If you try

```
int main() {
    DDRE |= _BV(2);
    while (1) {
      PORTE &= ~_BV(2);
      PORTE |=  _BV(2);
   }
    return 0;
}
```

Don't expect that you can see the LED blinking. You might see that the LED glows darker though. Why? Because it's actually blinking at the rate of megahertz.

We need to slow it down, or delay it specifically. Because we only have one thread on this MCU, the timing becomes deterministic. Let's read about <util/delay.h> in avr-libc document, it provides some useful functions that we want to use it now!

Beware that the *_delay_ms()* has limitation, it couldn't be forever. Because the crystal oscillator we used is rated as 14.7456Mhz (See MCU schematic), *_delay_ms()* could delay up to 262.14 / 14.7456 = ~17.53ms at most. If we want to delay up to half a second, we could use a for-loop.

Now let us try again

```
// Don't forget to include this or else very likely the
compiler would complain.
#include <util/delay.h>
```

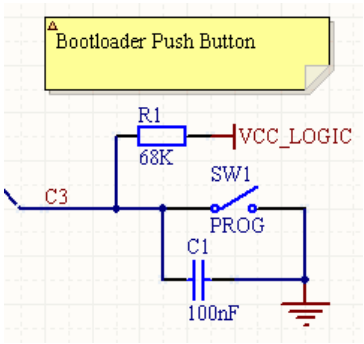
```
int main() {
    uint8_t i;
    DDRE |= _BV(2);
    while (1) {
        PORTE &= ~_BV(2);
        for (i = 0; i < 50; i++) _delay_ms(10);
     PORTE |=  _BV(2);
        for (i = 0; i < 50; i++) _delay_ms(10);
   }
    return 0;
}
```

Now the LED would blink at the rate of 1Hz.

**uint8_t**? Is that what you want to ask? See <stdint.h> of avr-libc. Because the memory and processing power of the MCU is limited, we want to have precise control about the memory size; Donation like `uint8_t` makes it very clear that it is a 8-bit variable, unsigned, the smallest addressable unit of 8-bit processor like ATMega128. Using `int` here would be overkill. By the way, `int` is only 16-bit wide here, instead of 32-bit wide in desktop computing.

Donation like `uint8_t` is actually defined in common C programming environment and available to desktop programming too. If you have done kernel programming before, you should be aware of this.

## *v. Reading a Button*




You should be able to figure out where is the button connected? let's focus on the *SW1 PROG* button. The *R1* in this case is a pull-up resistor, even if the pull-up and output is disable in the MCU, the net *C3* is defined as high by the *R1*.

So is the button active low or active high? When you activate it (push it), it would short two pins, which bring net *C3* to the GND.

The capacitor *C1* is to debounce the signal. Read http://en.wikipedia.org/wiki/Switch#Contact_bounce for details. Because we have hardware debouncing now, you do not have to do that in the software when using this switch. Practically, if you are rescanning the button status slow enough, like at 20Hz, you can ignore the debouncing issue, because usually the switch bounces duration is limited to milliseconds only.

Here is the code segment for turning on the LED when the switch is pressed. Recall that you should use PIN*x* for reading a pin logic. You need input mode in this case, pull-up is not important because we have external pull-up already, we will disable it to save some juice.

```
int main() {
   uint8_t i;
    DDRC &= ~_BV(3);
    PORTC &= ~_BV(3);
```

```
   DDRE |= _BV(2);

   while (1) {
      if (~PINC & _BV(3)) {
            // Pressed and On
         PORTE &= ~_BV(2);
         } else {
         // Release and off
         PORTE |=  _BV(2);
      }
   }
   return 0;
}
```

## *vi. State*

How about this: you want to turn on or off the LED when the switch is pressed. You would found the following code segment does not work:

```
while (1) {
   if (~PINC & _BV(3)) {
         PORTE ^= _BV(2);  // ^ means XOR-Exclusive Or
    }
}
```

Why? Because this code only does **level** detection, but in this case, we need **edge** detection. Specifically, we need to detect the moment from Unpressed to Pressed, the High to Low transition, or **falling edge** (The Low to High transition is called **rising edge**), but not the level detection.

To do that, you need a variable to store the previous input, so that you can compare the current input with the previous input and detect the edge. The code for doing that is left as an exercise for the readers.

## *vii. Conclusion*

After this chapter, you should know how to read the schematic, as well as finding revalent information from the datasheet. The unique style of MCU programming such as the use of bitwise operations, strict memory control as well as deterministic speed. Last but not least, interfacing basic hardware with GPIO lines.

# 4. UART

UART is a very simple, mature, old days but yet commonly used serial **communication protocol**. UART actually just defines **data link layer**, i.e. how the bytes are represented in a serial way and how it works **logically**, but it cannot be used without a **physical layer**. In general, UART is used with **RS-232** when connected to computer. RS-232 defines the wire, the receptor (plug) and most importantly the voltage levels that represent different logical value. In fact, you can do UART over Fiber, over Wireless, over IR, but there are no such standard for these or at least not commonly used.

Perhaps we should go through the protocol basic first.

## *i. Protocol and Layers of Protocol*

Protocol is like human language, both party has to speak the same language to communicate. We have standard protocol because we want to transfer something to somewhere easily. A protocol is needed whenever you want to transfer something to somewhere else; Something could be a byte, a stream of data, a movie, a web page, etc; Somewhere could be a MCU, a motor, a LCD display, a computer, etc. People have designed different protocol for different needs, so the stuff made by any people can communicate in between if they both implement the standard.

In our project, we need to interface some other devices that are smarter than LED and Push button, e.g. LCD, motor controller, keyboard or even a mini-computer. We have to use standard protocol that they have adopted to communicate with them. We would have to invent some high level protocol for our application to communicate too.

UART is a very commonly encountered protocol. It has been used extensively for transferring low bitrate byte stream in between connected processor like MCU and Computer. We will also use that in our project, and it's a good starting point to learn more about protocol.

As mentioned earlier, UART just defines the data link layer, luckily there is RS-232, a very common physical layer standard that defines how is the UART logic is carried to the other connected device. If we are going to use UART to let different programs and application talk, we also need to define the protocol for the **application layer**, this would actually be done in the Robocon project. In this chapter, we will use UART to transfer message to ourselves, a.k.a. Human, so we will use another very common standard that we know-English for the application protocol.

## a. Physical Layer

Physical layer usually comes with two flavors, in **parallel** or in **serial**. Parallel is only possible when you can distinguish the signal from its physical interface, such as over wires fiber or multiple wireless channel. The advantages of parallel is that it is faster and simpler, consider using one or two GPIO ports for parallel connection, communication would be as easy as writing a byte to the PORT and reading from the PIN. It is much easier to decode than serial. However, parallel electronics connection suffers from **cross-talk** and signal quality deteriorates

quickly at high speed. Besides, more wires mean more point of failures. Another major disadvantage which is crucial in embedded application is that it takes up more precious real estate.

As the processing power of MCU increases, decoding serial protocol is affordable, as you can see in the ATMega128 specification, it supports **Serial Peripheral Interface SPI** and **Two-wire Serial Interface TWI**, a.k.a. **I2C**.

A very famous serial protocol example is **USB**, while others like **Parallel ATA** and **DDR DIMM** is based on parallel bus.

## b. Data Link Layer

How fast are the data sent over the link, how multiple devices share the same bus, how to retry a packet, acknowledge a packet, detecting receiving errors, checksum, etc. these are taken care in the data link layer.

## c. Application Layer

With the first two layer, you have got a usable basic communication channel. Then it would be your job to define how to use it. For example, you can send English ASCII text over UART for human to read. For communication between applications and programs, you need to build a application communication protocol.

That's enough protocol overview. If you know about OSI Model, a.k.a. 7 Layer model, you should aware that the above description is overly simplified. But in practice, because the communication in embedded application is usually device to device or in a small network at most, network layer is pointless and others upper layers can be done in application level.

# *ii. UART and RS-232*

Let’s come back to the topic. We will go into the UART and RS-232.

The UART protocol defined two signal state, **SPACE** and **MARK**. Mark represents a binary 1 or off. Space represents binary 0 or on. The UART output of ATMega128 use VCC to presents Mark, and GND to represent Space. When the UART interface is enabled, it would stay at Mark state, i.e. VCC.

RS-232 defines the Space to be represented > 3V to 27V, and Mark to be represented < -3V to -27V. The higher voltage difference allows longer cable length. But because the voltage requirement is different in UART, a transceiver, which is something that deal with physical layer differences, is needed. A very famous UART<->RS-232 transceiver is MAX232 series, and Maxim-IC is the leader of making MAX232 chips. In particular, we are using MAX3222 on this board. The transceiver takes care of the not-so-standard voltage requirement of RS-232.

As mentioned earlier, UART is a byte-stream oriented protocol, it does not distinguish provide facility for dividing stream into packets. Although the transmission is almost error-free and collision-free because of its point to point

nature, it does not provide any form of acknowledgement.

The basic transmission unit is a byte. A start of transmission is donated by a Space bit, a.k.a. the start bit, then bits of byte follows, with LSB transmitted first, then a optional parity bit and a compulsory Mark bit signal the stop. The time of one bit time is defined by baudrate, which must be known by both parties beforehand.

## *iii. ATMega128 UART Interface*

You do not have to generate the Space and Mark signal, and doing *_delay_us()* to simulate the baudrate with GPIO lines, because there is a **specialized interface** to save your life. By using some SFR, every nasty work is handled by the MCU hardware. Specialized hardware supports allow you to read a byte, write a byte, almost without caring any low level stuff.

There is also specialized interfacing hardware for SPI and $I^2C$, a.k.a. TWI in the spec to avoid trademark infringement issue. Once you know how to use UART, you will probably master how to use the others. While sometimes there are protocols, such as **Keyboard PS/2** or **IR Demodulator** that we might use later on, are not supported by hardware, and we have to do it with the GPIO way, this technique is called **bit-banging** and indeed a very common way to do thing too.

Time for specification reading! Go straight to page 190, the *Register Description*. This time each bit is further explained because of the operation complexity. By the way, if you are not aware yet, the SFR can not just be read and written by your program, the MCU hardware can also access and modify the content of SFR, so you can get updated information by reading the SFR. How and when the MCU do this is of course clearly specified in the specification.

## *iv. Code Implementation*

Please refer to the sample code files.

## *v. Interrupt Operation*

So you may ask, what can you do when it has finished sending the data? This is a very common question in programming, i.e. waiting for something. There are two common answers to this:

- **Polling**. This can be applied to all situations. Including waiting for a button to be pressed. This is exactly what you have done in last chapter, using a while loop to keep an eye over the button status.
- **Interrupted.** This needs hardware support. An interrupt is generated when something deserve your attention. You can program the corresponding register to enable or disable a particular interrupt. UART, as well as all the hardware communication interface on ATMega128, provides you a selection of interrupts to notify you that a transmission is done, so you no longer need to actively check the SFR status in a while loop, the MCU would tell you by interrupting you instead.

The ATMega128 comes with 35 different kinds of interrupt, as you can see in page 59. A particular interrupt is enabled only when you have set a corresponding bit in the matching register, as well as enabled the **global interrupt**. When an enabled interrupt does occur, the MCU will halt your current operation and execute the interrupt service routine, a.k.a. **ISR**. How it is done is exactly like force your programming calling the ISR, and when the ISR finish it returns back to your program.

Interrupt can be happened anytime. Your main program would not notice that it has occurred. So it is common practice is to avoid long ISR so not to interfering main program so much.

Communication, i.e. passing the data between ISR and main program actually deserve a chapter for discussion. A singular linked list or its fixed size array version-ring buffer is a good choice in most case for passing data in FIFO fashion. We will leave out the details in this tutorial.