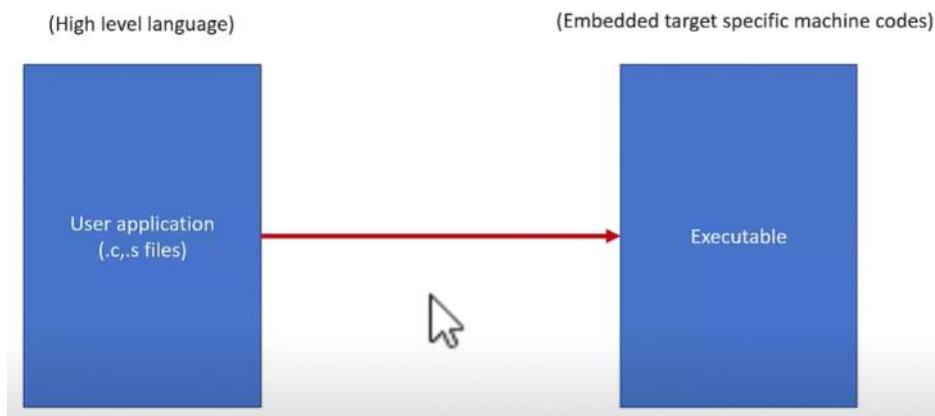
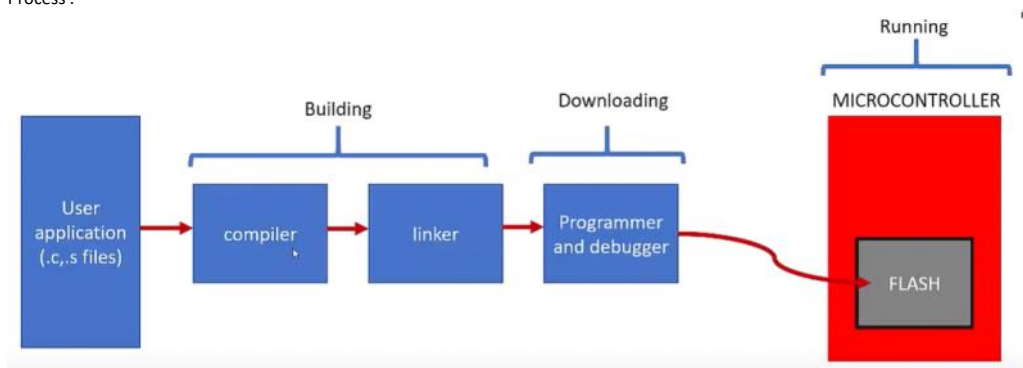


## Bare metal embedded lecture-1: Build process

### Section deliverables

- Toolchain installation
- Understand compiling a 'C' program for an embedded target without using an IDE
- Writing microcontroller startup file for STM32F4 MCU
- Writing your own 'C' startup code (code which runs before main())
- Understanding different sections of the relocatable object file (.o files)
- Writing linker script file from scratch and understanding section placements
- Linking multiple .o files using linker script and generating application executable (.elf, bin, hex)
- Loading the final executable on the target using OpenOCD and GDB client

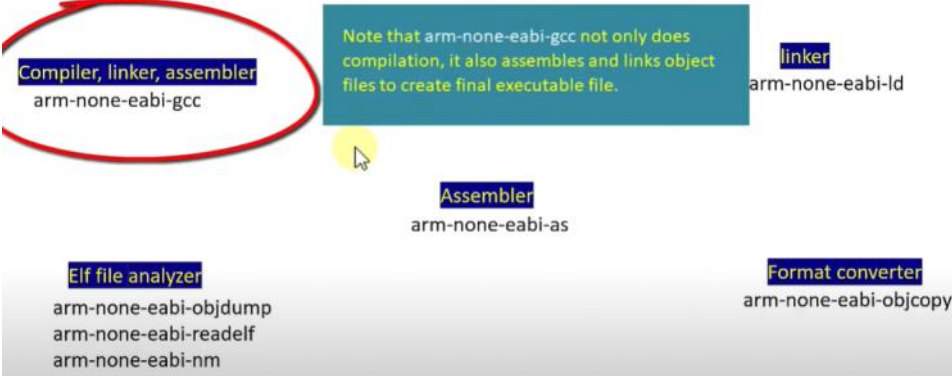
Process :-



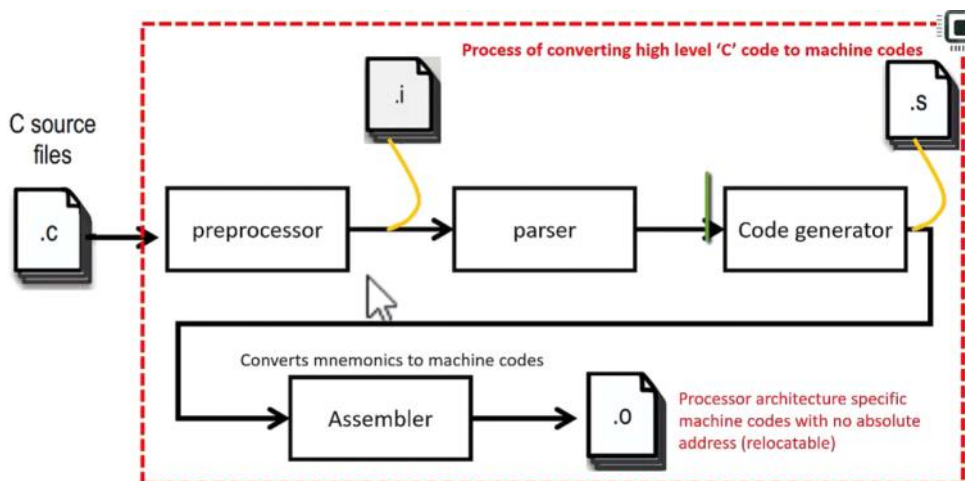
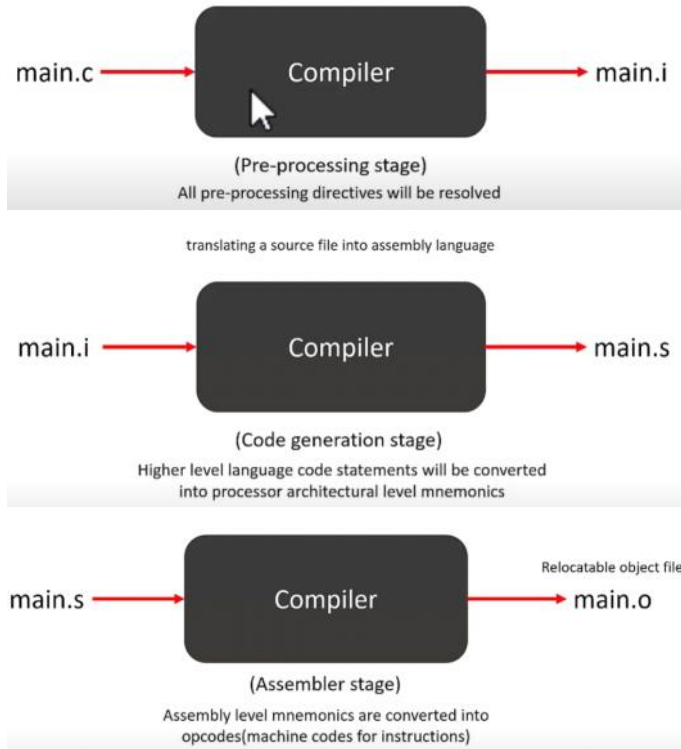
### What is cross compilation ?

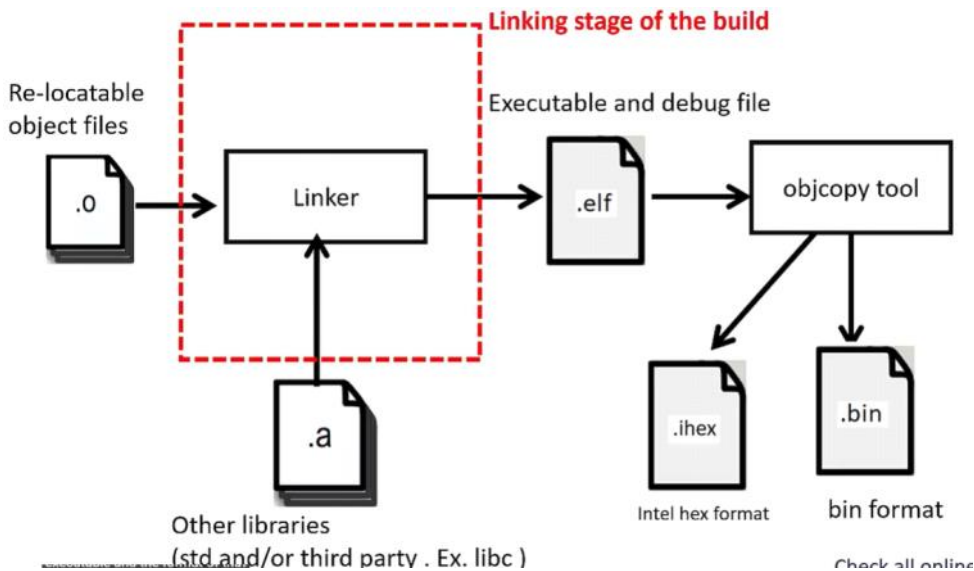
Cross-compilation is a process in which the cross-toolchain runs on the host machine(PC) and creates executables that run on different machine(ARM )

# Cross toolchain important binaries



1st Stage :-





This is just for compilation, there is no linking :- (We have stopped linking using -c)  
`arm-none-eabi-gcc -c -mcpu=cortex-m4 -mthumb main.c -o main.o`

If we want to just create assembly file, stop compilation at code generator state .s file then we can use -S instead of -c  
`arm-none-eabi-gcc -S -mcpu=cortex-m4 -mthumb main.c -o main.s`

## Bare metal embedded lecture-2: Makefile and analyzing relocatable obj file

Simple Makefile

```
CC=arm-none-eabi-gcc
MACH=cortex-m4
CFLAGS= -c -mcpu=$(MACH) -mthumb -std=gnu11 -O0
```

```
main.o:main.c
$(CC) $(CFLAGS) -o $@ $^
```

We can run Makefile using make command

## Analyzing .o files (Relocatable object files)

- main.o is in elf format(Executable and linkable format)
- ELF is a standard file format for object files and executable files when you use GCC
- A file format standard describes a way of organizing various elements(data, read-only data, code, uninitialized data, etc.) of a program in different sections.

## Relocatable object files



Analyze Object file :

```
arm-none-eabi-objdump -h main.o
arm-none-eabi-objdump -d main.o > main_log
arm-none-eabi-objdump -s main.o
arm-none-eabi-objdump -D main.o > main_log // to get disassembly of all the sections
```

## Bare metal embedded lecture-3: Writing MCU startup file from scratch

## Importance of start-up file

- The startup file is responsible for setting up the right environment for the main user code to run
- Code written in startup file runs before main(). So, you can say startup file calls main()
- Some part of the startup code file is the target (Processor) dependent
- Startup code takes care of vector table placement in code memory as required by the ARM cortex Mx processor
- Startup code may also take care of stack reinitialization
- Startup code is responsible of .data, .bss section initialization in main memory

## Start-up file

1. Create a vector table for your microcontroller . Vector tables are MCU specific
2. Write a start-up code which initializes .data and .bss section in SRAM
3. Call main()



Our own created data will go to data section, we need to instruct the compiler not to put this data in data section, rather in a different section (called user-defined section) `section ("section-name")`

Normally, the compiler places the objects it generates in sections like `.data` and `.bss`. Sometimes, however, you need additional sections, or you need certain particular variables to appear in special sections, for example to map to special hardware. The `section` attribute specifies that a variable (or function) lives in a particular section. For example, this small program uses several specific section names:

```
struct duart a __attribute__((section ("DUART_A"))) = { 0 };
struct duart b __attribute__((section ("DUART_B"))) = { 0 };
char stack[10000] __attribute__((section ("STACK"))) = { 0 };
int init_data __attribute__((section ("INITDATA")));
```

To check created section:- (file\_name = stm32\_startup.c)  
`arm-none-eabi-objdump -h stm32_startup.o`

## Function attribute : weak and alias

Weak :

Lets programmer to override already defined weak function(dummy) with the same function name

Alias :

Lets programmer to give alias name for a function

Reset handler is the first function which gets executed whenever the processor undergoes reset. Will write code in reset handler of startup code which initializes .data and .bss section in SRAM.

[Bare metal embedded lecture-4: Writing linker scripts and section placement](#)

# Linker scripts

- Linker script is a text file which explains how different sections of the object files should be merged to create an output file
- Linker and locator combination assigns unique absolute addresses to different sections of the output file by referring to address information mentioned in the linker script
- Linker script also includes the code and data memory address and size information.
- Linker scripts are written using the GNU linker command language.
- GNU linker script has the file extension of .ld
- You must supply linker script at the linking phase to the linker using `-T` option.

## Linker scripts commands

- ENTRY
- MEMORY
- SECTIONS
- KEEP
- ALIGN
- AT>

### ENTRY command

- This command is used to set the "**Entry point address**" information in the header of final elf file generated

- Syntax : `Entry(_symbol_name_)`  
`Entry(Reset_Handler)`

`ENTRY(Reset_Handler)`

### Memory command

- This command allows you to describe the different memories present in the target and their start address and size information

#### Syntax :

MEMORY

```
{  
  name (attr) : ORIGIN = origin, LENGTH = len  
}
```

defines origin address of the memory region

Defines the length information

Defines name of the memory region which will be later referenced by other parts of the linker script

```
MEMORY  
{  
  FLASH(rx):ORIGIN=0x08000000, LENGTH=1024K  
  SRAM1(rwx):ORIGIN=0x20000000, LENGTH=116K  
  SRAM2(rwx):ORIGIN=0x20000000+116K-4, LENGTH=16K  
}
```

### Sections command

- SECTIONS command is used to create different output sections in the final elf executable generated.

```

/* Sections */
SECTIONS
{
    /* This section should include .text section of all input files */
    .text :
    {
        //merge all .isr_vector section of all input files
        //merge all .text section of all input files
        //merge all .rodata section of all input files

    } >(vma) AT>(!ma)

    /* This section should include .data section of all input files */
    .data :
    {
        //here merge all .data section of all input files

    } >(vma) AT>(!ma)
}

```

```

SECTIONS
{
    .text :
    {
        *(.isr_vector)
        *(.text)
        *(.rodata)
    } > FLASH

    .data :
    {
        *(.data)
    } > SRAM AT> FLASH

    .bss :
    {
        *(.bss)
    } > SRAM
}

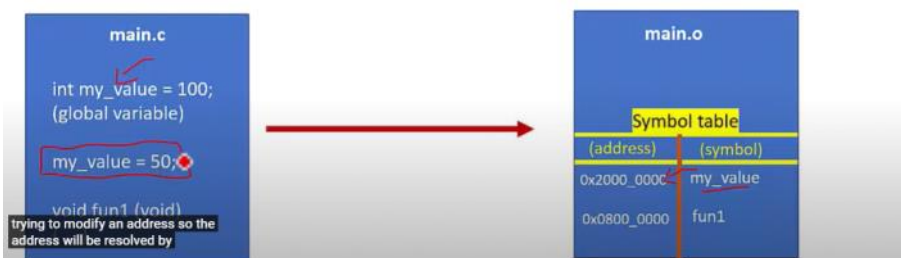
```

## Location counter (.)

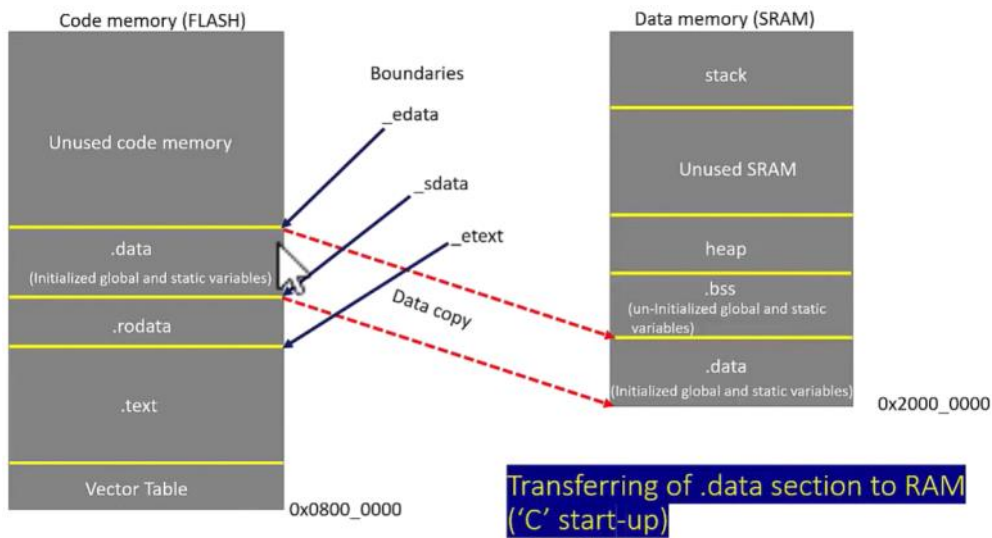
- This is a special linker symbol denoted by a dot ‘.’
- This symbol is called “location counter” since linker automatically updates this symbol with location(address) information

## Linker script symbol

- A symbol is the name of an address
- A symbol declaration is not equivalent to a variable declaration what you do in your ‘C’ application







A symbol is the name given for an address.

#### Linker Script:-

ENTRY(Rest\_Handler)

#### MEMORY

```
{
  FLASH(rx):ORIGIN=0x08000000, LENGTH=1024K
  SRAM(rwx):ORIGIN=0x20000000, LENGTH=128K
}
```

#### SECTIONS

```
{
  .text :
  {
    *(.isr_vector)
    *(.text)
    *(.rodata)
    _etext = .;
  }> FLASH

  .data
  {
    _sdata = .;
    *(.data)
    _edata = .;
  }> SRAM AT> FLASH

  .bss
  {
    _sbss = .;
    *(.bss)
    _ebss = .;
  }> SRAM
}
```

#### Commands :

Make clean

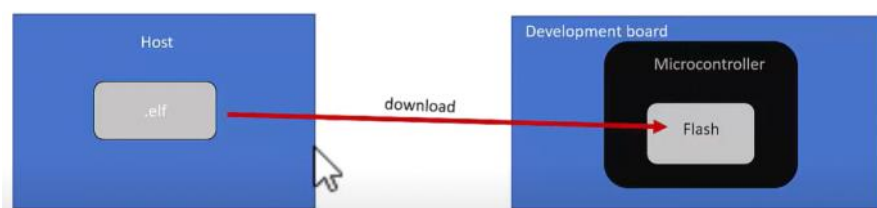
Make all

arm-none-eabi-gcc -nostdlib -T stm32\_ls.ld \*.o -o final.elf

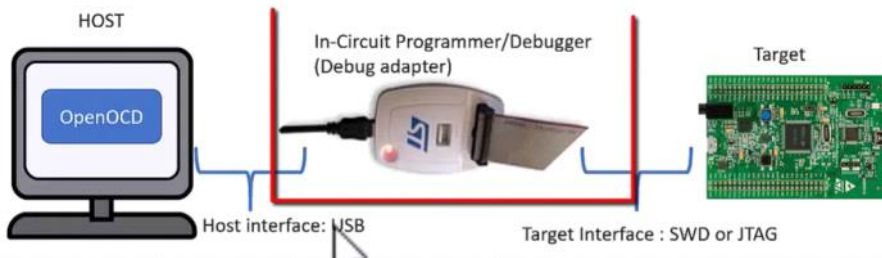
arm-none-eabi-objdump -h final.elf

### Bare metal embedded lecture-6: Downloading and debugging executable

## Downloading and debugging executable



# Downloading executable to Target



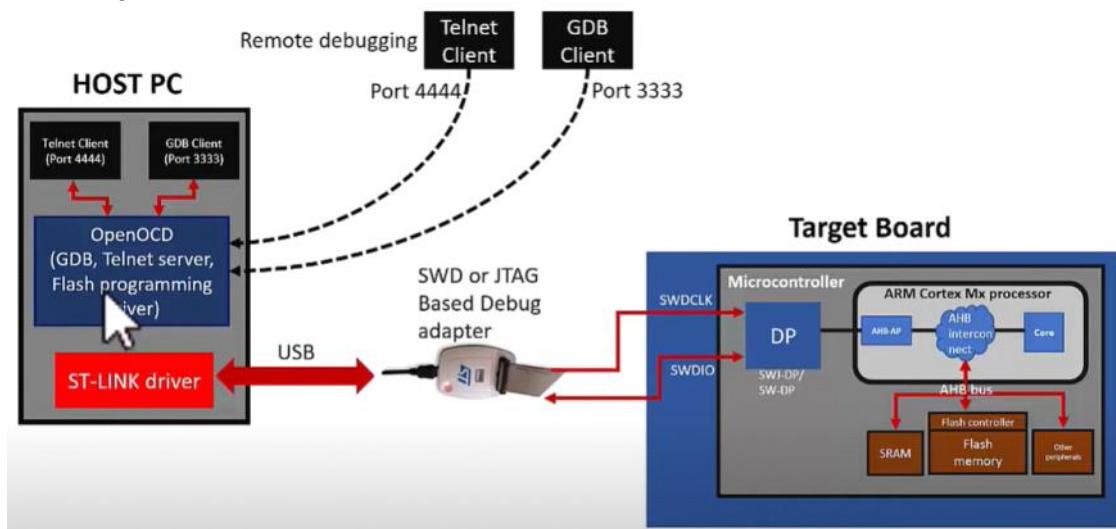
## OpenOCD(Open On Chip Debugger)

- The Open On-Chip Debugger (OpenOCD) aims to provide debugging, in-system programming, and boundary-scan testing for embedded target devices.
- Its free and opensource host application allows you to program, debug, and analyze your applications using GDB
- It supports various target boards based on different processor architecture
- OpenOCD currently supports many types of debug adapters: USB-based, parallel port-based, and other standalone boxes that run OpenOCD internally
- GDB Debug: It allows ARM7 (ARM7TDMI and ARM720t), ARM9 (ARM920T, ARM922T, ARM926EJ-S, ARM966E-S), XScale (PXA25x, IXP42x), Cortex-M3 (Stellaris LM3, ST STM32, and Energy Micro EFM32) and Intel Quark (x10xx) based cores to be debugged via the GDB protocol.
- Flash Programming: Flash writing is supported for external CFI-compatible NOR flashes (Intel and AMD/Spansion command set) and several internal flashes (LPC1700, LPC1800, LPC2000, LPC4300, AT91SAM7, AT91SAM3U, STR7x, STR9x, LM3, STM32x, and EFM32). Preliminary support for various NAND flash controllers (LPC3180, Orion, S3C24xx, more) is included

## Programming adapters

- Programming adapters are used to get access to the debug interface of the target with native protocol signaling such as SWD or JTAG since HOST doesn't support such interfaces.
- It does protocol conversion. For example, commands and messages coming from host application in the form of USB packets will be converted to equivalent debug interface signaling (SWD or JTAG) and vice versa
- Mainly debug adapter helps you to download and debug the code
- Some advanced debug adapters will also help you to capture trace events such as on the fly instruction trace and profiling information

How are code gets downloaded into the internal flash of the microcontroller :-





## Openocd

<https://openocd.org/pages/getting-openocd.html>  
<https://github.com/XPack-dev-tools/openocd-xpack/releases>  
<https://github.com/ilg-archived/openocd/releases>


## Bare Metal Embedded on Arduino :

[https://baremetalmicro.com/tutorial\\_avr\\_digital\\_io/index.html](https://baremetalmicro.com/tutorial_avr_digital_io/index.html)

Introduction:- (Blink LED)

<https://create.arduino.cc/projecthub/milanistef/introduction-to-bare-metal-programming-in-arduino-uno-f3e2b4>  
<https://maker.pro/arduino/tutorial/introduction-to-bare-metal-programming-for-arduino>  
<https://medium.com/geekculture/avr-c-for-arduino-uno-6118bbf3a951>

Interrupts:-

[https://baremetalmicro.com/tutorial\\_avr\\_digital\\_io/06-Pin-Change-Interrupts.html](https://baremetalmicro.com/tutorial_avr_digital_io/06-Pin-Change-Interrupts.html)  
[https://baremetalmicro.com/tutorial\\_avr\\_digital\\_io/07-External-Interrupts.html](https://baremetalmicro.com/tutorial_avr_digital_io/07-External-Interrupts.html)  
Interrupts  ATmega328P Programming #6 AVR microcontroller with Atmel Studio  
<https://forum.arduino.cc/t/solved-int0-external-interrupt-register-based-approach-leonardo/291414>  
<https://www.arxterra.com/10-atmega328p-interrupts/>

<http://www.elproducts.com/understanding-hex-files.html>

UART :-

<http://www.rjhcoding.com/avrc-uart.php>  
<https://www.xanthium.in/how-to-avr-atmega328p-microcontroller-usart-uart-embedded-programming-avrgcc>  
<https://embedds.com/programming-avr-usart-with-avr-gcc-part-1/>  
<https://embedds.com/programming-avr-usart-with-avr-gcc-part-2/>

- 1) try same code with rising/falling edge
  - 2) explanation of isr
  - 3) analyse the hex binary
  - 4) create a binary and understand b/w hex and binary file
- avr-objcopy -O binary -R .eeprom interrupts\_external interrupts\_external.bin

-S            Compile only; do not assemble or link.  
-c            Compile and assemble, but do not link.  
-o <file>    Place the output into <file>.  
avr-gcc -Os -DF\_CPU=16000000UL -mmcu=atmega328p -S -o interrupts\_external.  
interrupts\_external.c

Check size : du -shc /var/\*

## Understanding Hex Files

:BBaaAATDDCC

BB contains the number of data bytes on line.

aaAA is address in memory where the bytes will be stored. This number is actually doubled which I'll cover in a bit. Also the lower byte is first aa followed by the upper byte AA.

TT is the data type.

00 - means program data.

01 - means End of File (EOF).

04 - means extended address. It indicates the data value is the upper 16 bits of the address.

DD is actual data bytes which contain the machine code that your program created. There can be numerous bytes in one line. The BB value indicates how many bytes are included in the line.

CC is calculated checksum value for error monitoring. It's a 2s-complement calculation of: BB + AAAA + TT + DD.

:100000000C9434000C943E000C943E000C943E0082

10 - number of bytes on the line. 16 bytes.

0010 - is the memory address to place the bytes but its value was multiplied by 2. So normally we would divide the address by 2 but in this case 0010 / 2 = 0005. So the address is 0005.

00 - means program data.

0C94 - the first two data bytes. But again these are reversed since the lower byte is first so this is really 940C hex.

This byte is followed by 14 more:

003E

940C

0040

940C

003E

940C

003E

66 - is the 2's complement checksum.

:100010000C943E000C9440000C943E000C943E0066

:100020000C943E000C943E000C943E000C943E0058

:100030000C943E000C943E000C943E000C943E0048

:100040000C943E000C943E000C943E000C943E0038

:100050000C943E000C943E000C943E000C943E0028

:100060000C943E000C943E0011241FBECFEFD8E04C

:10007000DEBFCDBF0E9446000C9455000C940000DA

:100080004A9B02C02D9818952D9AFDCF259A52981B

:100090005A9A80916D00846080936D008091680011

:0E00A0008460809368007894FFCFF894FFCFBF

:00000001FF