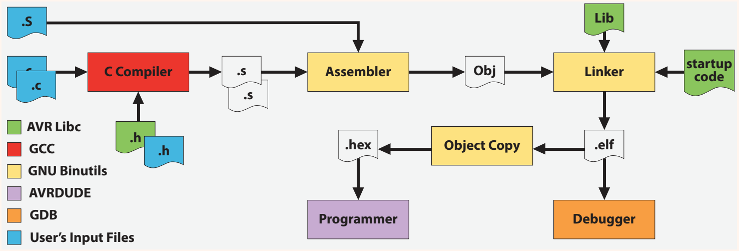
## GNU toolchain

- The GNU toolchain has played a vital role in the development of the Linux kernel, BSD, and software for embedded systems.
- The AVR toolchain was built upon the GNU toolchain and includes:
    - bash: (Bourne-again shell)
    - make: Automation tool for compilation and build
    - avr-gcc : GNU Compiler for AVR architecture
    - binutils: Suite of supporting tools for AVR
        - avr-as, avr-ld, avr-objcopy, avr-size, avr-ld
    - gdb: GNU debugger
    - avr_libc : Subset of standard C library for Atmel AVR
        - io.h, delay.h, ctype.h, math.h, string.h, interrupt.h, twi.h, etc.

## GNU toolchain



- C source files are compiled by avr-gcc and assembled by avr-as.
- Assembly files (.S) can be read in directly to avr-as
- Object files from the assembler are linked by avr-ld to produce a single relocatable object file.
- Physical memory addresses are assigned to the relative offsets within the relocatable program by avr-ld.
- The linker inserts the necessary startup and library code required.
- The `.elf` (Executable and Linkable Format) file output from the linker which has sufficient information to build the programming files.

# GNU toolchain

- The object file from the avr-ld contains separate sections containing code or data:
  - Executable code is in the `.text` section
  - Initialized global variables are in the `.data` section
  - Uninitialized global variables are in the `.bss` section
- To view these areas, invoke `avr-size` on the `.elf` file.

```
traylor-ubuntu$  avr-size  alarm_clock.elf

  text     data     bss     dec     hex     filename
  8646      317     342     9305    2459    alarm_clock.elf
```

- Total flash used will be (*.text* + *.data* + *.bss*.) or 9305 bytes.
- These are decimal values except for the *hex* column.
- Why are initalized and uninitalized variables being stored in flash?

## GNU toolchain

- *make* looks for a makefile in the current directory. The makefile describes the dependencies and actions using makefile *rules*
- A *rule* is formed as follows:

```
target        :  prerequisites
<hard tab>       commands
```

- *target*: usually the name of a file generated by the command below. It can also be the name of an action to carry out. (phony target)
- *prerequisites*: file(s) that used as input to create the target. A target usually depends on several files.
- *commands* : usually shell command(s) used to create the target. Can also specify commands for a target that does not depend on any prerequisites. (e.g.; clean)

## GNU toolchain

- An example rule:

```
sr.o   :   sr.c
           avr-gcc -g -c -Wall -O2 -mmcu=atmega128 -o sr.o sr.c
```

- This rule tells make that:
  - `sr.o` is dependent on `sr.c`
  - if `sr.c` is newer than sr.o, recreate `sr.o` by executing avr-gcc
- A rule then, explains how and when to remake certain files which are the targets of the particular rule. *make* carries out the commands on the prerequisites to create or update the target.

## GNU toolchain

- *make* does its work in two phases:
  1. Include needed files, initialize variables, rules, build dependency graph
  2. Determine which targets to rebuild, and invoke commands to do so
- The order of the rules is insignificant, except for determining the default goal.
- Rules tell make when targets are out of date and how to update them if necessary.
- Targets are out of date if they do not exist or if they are older than any of its prerequisities.

## GNU toolchain

- A simple makefile for edit

```
edit       : main.o kbd.o command.o display.o insert.o search.o files.
             utils.o
             cc -o edit main.o kbd.o command.o display.o insert.o sear
             files.o utils.o
files.o    : utils.o
main.o     : main.c defs.h
             cc -c main.c
kbd.o      : kbd.c defs.h command.h
             cc -c kbd.c
command.o  : command.c defs.h command.h
             cc -c command.c
display.o  : display.c defs.h buffer.h
             cc -c display.c
insert.o   : insert.c defs.h buffer.h
             cc -c insert.c
search.o   : search.c defs.h buffer.h
             cc -c search.c
files.o    : files.c defs.h buffer.h command.h
             cc -c files.c
utils.o    : utils.c defs.h
             cc    utils.c
clean :
             rm edit main.o kbd.o command.o display.o insert.o search.
             utils.o
```

## GNU toolchain

- A simple make file for sr.c including variables

```
Listing goes here
```

## GNU toolchain

- We can make the makefile more general purpose with some new rules.
  - *Implicit rules* - *make* implicitly knows how to make some files. Its has implicit rules for updating `*.o` files from a corresponding `*.c` file using gcc. We can thus omit commands for making `*.o` files from the object file rules.
  - *Pattern rules* - You can define an *implicit* rule by writing a pattern rule. Pattern rules contain the character %. The target is considered a pattern for matching names.

    The pattern rule:

    ```
    %.o   :   %.c
              command
    ```

    Says to create any file ending with `.o` from a file ending with `.c` execute `command`.

## GNU toolchain

- Automatic variables
  - These are variables that are computed afresh for each rule that is executed based on the target and prerequisites of the rule.
  - The scope of automatic variables is thus very limited. They only have values within the command script. They cannot be used in the target or prerequisite lists.
  - Some automatic variables we will use:

| | |
|---|---|
| `$@` | The file name of the target of the rule |
| `$<` | The name of the first prerequisite |
| `$?` | The names of all the prerequisites newer than the target |
| `$^` | The names of all the prerequisites |

## GNU toolchain

```
Big make file goes here
```

## GNU toolchain

- Generally, translational units are turned into object files. But header files are not standalone translational units, *make* has no way of knowing that it needs to rebuild a `.c` file when a `.h` file changes.
- gcc is able to read your source file and generate a list of prerequisite files for it with the -MM switch.
- We can take this output and create another makefile from it using some shell commands. There will be one makefile for each source file. Each makefile is then included in our top level makefile so that the dependencies are taken care of.
- Here is the makefile code for this:

```
%.d: %.c
        @set -e; rm -f $@; \
        $(CC) -M $(CFLAGS) $< > $@.$$$$; \
        sed 's,\($*\)\.o[␣:]*,\1.o␣$@␣:␣,g' < $@.$$$$ > $@; \
        rm -f $@.$$$$

-include ($SRCS:.c =.d)
```