IOT Report

Ali BAYDOUN

March 2025

1 First Steps

First of all, i will understand and take notes regarding the initial files provided to use, that way i can use the notes i took as documentation in case i forget what a certain option does in the MakeFile for example.

I don't think i have to do this for the linker (kernel.ld), as it is already documented so i can just read the comments in the code in case needed.

2 The Makefile

The provided Makefile is used to automate the process of compiling, linking, and running our project for an ARM-based system. It defines compilation rules, handles dependencies, and sets up debugging options. Below is a breakdown of its key components.

2.1 Configurable Parameters

At the beginning of the Makefile, several variables are defined :

- `BOARD=versatile`: Specifies the target board. In this case, it is a "Versatile" board, which is an ARM-based development platform. (Documentation)
- `CPU=cortex-a8`: Defines the target CPU architecture, ensuring the correct instruction set is used. (Documentation)
- `TOOLCHAIN=arm-none-eabi`: Indicates the toolchain to be used for cross-compilation. Since the project targets an ARM processor, an appropriate cross-compiler is required.
- `DEBUG?=yes`: Enables or disables debugging features. If set to yes, additional flags are added for debugging.
- `BUILD=build/`: Specifies the directory where compiled output files will be stored.
- `objs= startup.o main.o exception.o uart.o`: Lists the object files that will be generated from their corresponding source files.

2.2 Handling Object Files

The Makefile constructs the list of object files dynamically using:

```
OBJS = $(addprefix $(BUILD), $(objs))
```

This prepends the build directory path to each object file name, ensuring they are placed in the correct location.

2.3 Conditional Compilation Based on Board

The Makefile includes conditional logic to configure compilation options based on the selected board. If the board is "versatile" (which it is in our case), it defines:

- QEMU-related settings:

  ```
  VGA=-nographic
  SERIAL=-serial mon:stdio
  ```

  These settings configure QEMU to run without a graphical interface and redirect serial output to the console.

- Memory settings:

  ```
  MEMSIZE=32
  MEMORY="$(MEMSIZE)K"
  ```

  The project is configured to allocate 32 KB of memory, this can easily be changed by modifying the MEMSIZE in the make file if we every need more memory size for our "kernel".

- QEMU machine type:

  ```
  MACHINE=versatileab
  ```

2.4 Compiler Flags (`CFLAGS`)

The `CFLAGS` variable contains options passed to the compiler when compiling C source files. The flags used in this Makefile are:

- `-c`: Tells the compiler to generate an object file (`.o`) instead of a complete executable.
- `-mcpu=$(CPU)`: Specifies the target CPU architecture.
- `-nostdlib`: Excludes the standard C library from the build process.
- `-ffreestanding`: Indicates that the code does not depend on a hosted environment (i.e., an operating system). So the software we are running is directly used "above" the hardware.
- `-DCPU=$(CPU)`: Defines a preprocessor macro `CPU` with the value specified in the Makefile.
- `-DMEMORY="($(MEMSIZE)*1024)"`: Defines a preprocessor macro `MEMORY` representing the memory size in bytes. This macro is used in the main.c file of our code to calculate the memory size.

If debugging is enabled (`DEBUG=yes`), the following additional flags are included:

- `-ggdb`: Generates debugging information that can be used by GDB.

2.5 Assembler Flags (`ASFLAGS`)

The `ASFLAGS` variable contains options passed to the assembler when assembling assembly source files:

- `-mcpu=$(CPU)`: Ensures the assembler generates code compatible with the specified ARM processor.

If debugging is enabled, the following flag is also included:

- `-g`: Includes debugging information in the assembled output to allow debugging at the assembly level.

2.6 Linker Flags (`LDFLAGS`)

The `LDFLAGS` variable contains options passed to the linker when linking object files into an executable:

- `-T kernel.ld`: Specifies the linker script to use, which will define the memory layout and how sections of the program (such as text, data, and stack) are organized.
- `-nostdlib`: Ensures the linker does not link against the standard C library.
- `-static`: Forces static linking, meaning no dynamic libraries are used.

If debugging is enabled, the following flag is also included:

- `-g`: Embeds debugging information into the final executable, allowing tools like GDB to perform source-level debugging.

2.7 Compilation Rules

The Makefile defines explicit rules for compiling source files:

- Compiling C files:

  ```
  $(BUILD)%.o: %.c
      $(TOOLCHAIN)-gcc $(CFLAGS) -o $@ $<
  ```

  This means any C source file (`%.c`) is compiled into an object file (`%.o`) and placed in the `BUILD` directory.

- Assembling assembly files:

  ```
  $(BUILD)%.o: %.s
      $(TOOLCHAIN)-as $(ASFLAGS) -o $@ $<
  ```

  This rule processes assembly source files in a similar manner.

2.8 Building and Linking the Kernel

To create the final executable, the Makefile defines the `all` target which ensures that:

- The `build` directory is created.
- The kernel is linked into an ELF executable.
- The ELF file is converted into a binary file.

The ELF file is built then converted into a binary format with:

```
$(BUILD)kernel.bin: $(BUILD)kernel.elf
    $(TOOLCHAIN)-objcopy -O binary $(BUILD)kernel.elf $(BUILD)kernel.bin
```

2.9 Creating the Build Directory

Since all compiled files are placed in the `build` directory, it must be created before compilation starts. The `build` target does this:

```
build:
    @mkdir $(BUILD)
```

2.10 Cleaning Up

To remove all compiled files and reset the project, the `clean` target is used:

```
clean:
    rm -rf $(BUILD)
```

This ensures that a fresh compilation is performed the next time the Makefile is run.

2.11 Running the code

```
run: all
    @echo "\n\nBoard: Versatile Board...\n"
    $(QEMU) $(QEMU_ARGS) -device loader,file=$(BUILD)kernel.elf
```

This compiles the project and runs it in QEMU.

2.12 Debugging with GDB

To debug the kernel, the `debug` target is used:

```
debug: all
    @echo "\n\nBoard: Versatile Board...\n"
    $(QEMU) $(QEMU_ARGS) -device loader,file=$(BUILD)kernel.elf -gdb tcp::1234 -S
```

This starts QEMU in debugging mode, waiting for a connection on TCP port 1234. The `-S` option ensures that execution is halted until the debugger is attached.

3 Handling Keyboard Inputs

The first thing I will try to do is receive inputs from the keyboard and display them on the screen.

To achieve this, i spent some time reading about UART, which stands for Universal Asynchronous Receiver/Transmitter. It is a simple, two-wire protocol for exchanging serial data, commonly used for communication between a computer and embedded systems. In this project, i will use UART as a hardware communication protocol to send and receive data between the system and a serial terminal.

I found the base addresses of UART0, UART1, and UART2, which are the three available UART interfaces, in the documentation of the Versatile board here. I used these addresses to define the corresponding base values in the `uart-mmio.h` file.

Additionally, I found detailed information about the PL011 UART registers in the ARM documentation here. This documentation provides the offset, name, type, and detailed descriptions of each register. For now, I will focus on two key registers: `UARTDR` and `UARTFR`, as they are essential for basic input and output operations.

`UARTDR` (UART Data Register) is used for reading received data and writing data to be transmitted. This is the primary register for sending and receiving characters.

`UARTFR` (UART Flag Register) contains various status flags that indicate the current state of the UART. The flags I will be using are:

- **TXFF (Transmit FIFO Full)**: Indicates whether the transmit FIFO buffer is full. If this flag is set, no more data can be written to `UARTDR` until space becomes available.
- **RXFE (Receive FIFO Empty)**: Indicates whether the receive FIFO buffer is empty. If this flag is set, there is no data available to read from `UARTDR`.

- **TXFE (Transmit FIFO Empty)**: Indicates whether the transmit FIFO buffer is empty. This is useful for checking if all outgoing data has been sent.
- **BUSY**: Indicates whether the UART is currently transmitting data. This flag can be used to ensure the UART has finished its operation before sending new data.

To check if a specific flag is set in the code, I need to use a bitmask operation. Each flag corresponds to a specific bit position in the `UARTFR` register. For example, the `RXFE` flag is located at bit position 4. To check whether this bit is set, I create a mask using `(1 << 4)`, which results in the binary value `0b00010000`. I then perform a bitwise AND operation between this mask and the value of the `UARTFR` register. If the result is nonzero, it means the flag is set; otherwise, it is cleared.

Then, for receiving data, I need to check if the receive FIFO buffer is not empty by examining the `RXFE` flag in the `UARTFR` register. This is done using a bitwise AND operation. If the flag is not set, it means data is available, and I can read the received character from the `UARTDR` register and store it in the pointer passed to the `receive` function.

For sending data, I will check if the transmit FIFO buffer is not full by examining the `TXFF` flag. If the buffer has space available (i.e., `TXFF` is not set), I can write the character to be sent into the `UARTDR` register. This ensures that the data is properly queued for transmission without overwriting any ongoing transmissions.