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.

3.1 Update for Week 2

Regarding the registers mentioned in Section 3, I ultimately used only `TXFF` and `RXFE`, as the other registers were not necessary for my current implementation.

4 Implementing Interrupts

4.1 Setting up the registers

To begin implementing interrupts, I first identified the Vector Interrupt Controller (VIC) used by the board. The Versatile board features the PL190 VIC, as listed in the board's supported peripherals documentation here.

The base address for the VIC is `0xFFFFF000`, according to the VIC programming model. Additionally, I noted the offsets of key registers that may be used:

- `VICIRQSTATUS` (`0x000`) – Indicates which interrupt sources are currently active and enabled for IRQ handling.
- `VICFIQSTATUS` (`0x004`) – Shows which interrupts are active and configured as FIQ (Fast Interrupt Request).
- `VICRAWINTR` (`0x008`) – Displays all active interrupt sources before any masking is applied.
- `VICINTSELECT` (`0x00C`) – Configures each interrupt source as either IRQ or FIQ.
- `VICINTENABLE` (`0x010`) – Enables specific interrupt sources by setting corresponding bits.
- `VICINTENCLEAR` (`0x014`) – Disables specific interrupts by clearing corresponding bits.

I also added the corresponding IRQ addresses to `isr.h`, allowing me to define and manage different interrupt sources. The file now includes the following IRQ definitions:

- **UART Interrupts:** These correspond to the three UART controllers on the board.
  - `UART0_IRQ` (IRQ 12) – Mask: `(1 << UART0_IRQ)`

- `UART1_IRQ` (IRQ 13) – Mask: `(1 << UART1_IRQ)`
- `UART2_IRQ` (IRQ 14) – Mask: `(1 << UART2_IRQ)`

- **Timer Interrupts:** These correspond to the system timers.
  - `TIMER3_IRQ` (IRQ 5) – Mask: `(1 << TIMER3_IRQ)`
  - `TIMER2_IRQ` (IRQ 5) – Mask: `(1 << TIMER2_IRQ)`
  - `TIMER1_IRQ` (IRQ 4) – Mask: `(1 << TIMER1_IRQ)`
  - `TIMER0_IRQ` (IRQ 4) – Mask: `(1 << TIMER0_IRQ)`

4.2 Implementing Interrupt Handling Functions

The process of enabling interrupts in the system was a multi-step approach that involved modifications at both the assembly and C levels, as well as updates to the linker script. Below is are the steps i did in order to achieve it:

4.2.1 Assembly Modifications and Interrupt Vector Setup

To start with, I modified the assembly startup code and exception handler to ensure that interrupts could be properly handled the C code:

- **Enabling IRQs in Startup:** Originally, the startup code disabled IRQ interrupts using a CPSR flag. I removed the disablement of the IRQ (by omitting the `CPSR_VIC_FLAG`) in the following line:

  ```
  msr        cpsr_c,#(CPSR_SYS_MODE | CPSR_FIQ_FLAG),
  ```

  This change allows IRQ interrupts to be enabled from the start, enabling the system to respond to them immediately.

- **Integrating the ISR Call:** I modified the exception handler in `exception.s` to call the C-level `isr()` function. The updated IRQ handler looks like:

  ```
  _isr_handler:
      sub lr, lr, #4
      stmfd sp!, {r0-r12, lr}
      bl isr
      ldmfd sp!, {r0-r12, pc}
  ```

  This sequence ensures that when an IRQ occurs, the processor saves the necessary registers, calls `isr()` to dispatch the interrupt to the appropriate handler, and then restores the registers before resuming execution.

4.2.2 Linker Script Updates

I also updated the linker script (`kernel.ld`) to clearly define the memory layout and reserve dedicated memory for the interrupt stack. Key changes include:

- Defining the interrupt vector table within the `.text` section to include the exception handlers.
- Reserving a specific memory region for the IRQ stack:

  ```
  . = ALIGN(8);
  . = . + 0x1000; /* 4KB of stack memory */
  irq_stack_top = .;
  ```

This organization ensures proper alignment and prevents stack overflows while allowing the CPU to switch stacks upon an interrupt.

4.2.3 C-Level Interrupt Handling Functions

Once the assembly and linker modifications were in place, I implemented the interrupt handling functions in C:

- `isr()`: This is the main Interrupt Service Routine (ISR) called from assembly when an interrupt occurs. It:
  1. Disables interrupts to prevent nested calls.
  2. Reads the active interrupts from the VIC using `vic_load_irqs()`.
  3. Iterates over all potential interrupts, calling the registered callback for any active interrupt.
  4. Re-enables interrupts after processing.
- `vic_setup_irqs()`: This function initializes the VIC by disabling all interrupts and then configuring the interrupt system via `_irqs_setup()`.
- `vic_enable_irq()` **and** `vic_disable_irq()`: These functions manage individual interrupt lines. `vic_enable_irq()` registers a callback (with associated data) and enables the interrupt by setting the corresponding bit in the `VICINTENABLE` register. Conversely, `vic_disable_irq()` clears the handler and disables the interrupt.

4.2.4 Testing VIC Interrupt Enabling

To verify that the VIC-level interrupt enabling was working correctly:

- I added the following line to `main.c`:

  ```
  vic_enable_irq(UART0_IRQ, uart_receive, (UART0, &c));
  ```

- Running the code in debug mode (with a breakpoint at the start of `_start()`), I checked the initial state of the `VICINTENABLE` register:

  ```
  x/1xw 0x10140000+0x10
  ```

  The output was:

  ```
  0x10140010: 0x00000000
  ```

  This confirmed that all interrupts were initially disabled.

- After executing `vic_enable_irq()`, running the same command yielded:

  ```
  0x10140010: 0x00001000
  ```

  Converting `0x00001000` to binary confirms that the 12th bit is set to `1` (as expected for `UART0_IRQ`). This verified that the function successfully enabled the correct interrupt.

4.2.5 Enabling UART Interrupts

To allow UART0 to generate interrupts upon receiving data, I completed the UART enable and disable functions:

- The `uart_enable()` function writes to the `UART_IMSC` register to enable the `RXIM` bit.
- The `uart_disable()` function clears the `UART_IMSC` register.

To test the UART interrupt functionality:

- I ensured that UARTs were initialized with `uarts_init()`.
- Then, I invoked `uart_enable(UART0)` in `main.c`.
- Using GDB, I first checked the state of the `UART_IMSC` register:

x/1xw 0x101F1000+0x038

which initially returned:

0x101f1038: 0x00000000

- After calling `uart_enable(UART0)`, the same command yielded:

  0x101f1038: 0x00000010

  Converting `0x00000010` to binary shows that the 4th bit is set, corresponding to the `RXIM` bit being enabled. This confirmed that UART0 receive interrupts is successfully activated.

4.2.6 Putting It All Together in `main.c`

In the final integration, I updated `main.c` to tie all the previous modifications together. In this file, I initialized the UARTs and the IRQ system, enabled the UART interrupts, and set up the IRQ parameters structure. These changes were made so that the system could properly handle incoming data. When the system starts, the following occurs:

- **Stack and System Checks:** The `check_stacks()` function verifies that both the C and IRQ stacks are within the allocated memory bounds.
- **UART Initialization:** The `setup_uarts()` function initializes the UARTs by setting their base addresses and enabling UART0. A confirmation message is sent to indicate that the UART setup is complete.
- **IRQ Setup:** The `setup_irqs()` function initializes the VIC and enables the system-wide interrupts, including the UART interrupt. This is achieved by calling `vic_setup_irqs()` followed by `vic_enable_irqs()`, which registers the UART interrupt callback.
- **Runtime Behavior:** Once all the initialization is complete, the system prints a "The system is now running..." message. From this point on, the main loop halts the CPU with `core_halt()` until an interrupt occurs.

With these modifications, when a button is pressed (or data is received via UART), the hardware triggers an IRQ. The interrupt is caught by the updated assembly handler, which calls the C-level `isr()` function. The `isr()` function then dispatches the interrupt to the registered callback (in this case, `uart_interrupt()`), which processes the incoming data by reading from the UART data register and echoing the character back to the screen.

4.2.7 Taking It Further: Implementing a Minimal Shell

To extend the use of UART interrupts beyond simple input and output, I implemented a minimal shell that processes user commands. This shell uses the interrupt-driven UART input to collect characters, allowing users to enter commands. The commands I added are:

- `echo <message>`**:** Prints the given message back to the console.
- `clear`**:** Clears the terminal screen using ANSI escape sequences.
- `curs <on|off>`**:** Enables or disables the terminal cursor.
- `help`**:** Displays a list of available commands.

I also implemented basic character processing features, such as handling the backspace key (`\b` or `0x7F`) to allow users to correct their input and processing the enter key (`\r` or `\n`) to execute commands.

One issue i'm encountering during development is handling arrow key inputs. When pressing an arrow key, the system receives a sequence of three characters corresponding to ANSI escape codes (e.g., `/033[D` for the left arrow). While I was able to detect these sequences correctly, an unexpected behavior occurres: after receiving the three characters, the system would trigger an additional interrupt that causes a call to `_reset_handler`, which resets the entire system by invoking `_start`.