STM32 Nucleo

Board

- https://en.wikipedia.org/wiki/STM32#Development_boards
- NUCLEO-F302R8 board for STM32F302R8T6 MCU with 72 MHz **Cortex-M4F** core, 64 KB flash, 16 KB SRAM.
- RAM: `0x20000000` (16 KiB)
- FLASH: `0x08000000` (64 KiB)
- Start address (entry point): `0x08000400`

Documentation

- https://os.mbed.com/platforms/ST-Nucleo-F302R8/#getting-started
- MCU Reference manual
- Board User manual

Software - Void Linux

1. Install software

   ```
   xi stlink
   ```

2. Set up udev rules (as root)

   ```
   #/etc/udev/rules.d/stm32nucleo.rules
   SUBSYSTEM!="usb|usb_device", ACTION!="add", GOTO="stm32nucleo_end"
   ATTRS{idVendor}=="0483", ATTRS{idProduct}=="374b", SYMLINK+="stm32-%k", MODE="660",
   GROUP="input"
   LABEL="stm32nucleo_end"
   ```

3. Connect device
4. Probe for the device `st-info --probe`; should print something like this:

```
Found 1 stlink programmers
  version:    V2J33S25
  serial:     066FFF5155xxxxxxxxxxxx
  flash:      65536 (pagesize: 2048)
  sram:       40960
  chipid:     0x439
  dev-type:   STM32F301_F302_F318
```

5. Read boot or flash area

```
st-flash read /tmp/boot.bin 0x0 0x10000 # flash if booting from flash
st-flash read /tmp/flash.bin 0x8000000 0x10000 # flash
```

Debugging

1. Install tools

```
 xi cross-arm-none-eabi cross-arm-none-eabi-gdb
```

2. Using GDB
   1. Connect to device

```
 st-util
```

   2. Run GDB

```
 arm-none-eabi-gdb -ex 'target extended-remote localhost:4242'
```

Using OpenOCD

1. Install

```
xi openocd inetutils-telnet
```

2. Run server

```
openocd -f interface/stlink.cfg -f target/stm32f3x.cfg
```

3. It should print something like

```
Info : STLINK V2J33M25 (API v2) VID:PID 0483:374B
Info : Target voltage: 3.238345
```

```
Info : [stm32f3x.cpu] Cortex-M4 r0p1 processor detected
Info : [stm32f3x.cpu] target has 6 breakpoints, 4 watchpoints
```

4. Now you can telnet to the debugger: `telnet localhost 4444`

```
halt
reg
# read output register of port B
read_memory 0x48000414 32 1
# disable LED on PB-13
write_memory 0x48000414 32 0x0
# enable LED on PB-13
write_memory 0x48000414 32 0x2000
```

5. Or use **gdb**: `arm-none-eabi-gdb -ex 'target extended-remote :3333'`

Rust

- https://docs.rust-embedded.org/book/start/qemu.html

Tools:

```
rustup target add thumbv7em-none-eabihf
rustup component add llvm-tools-preview
cargo install cargo-binutils
cargo install cargo-generate
```

Template project

```
cargo generate --git https://github.com/rust-embedded/cortex-m-quickstart
```

Edit `.cargo/config.toml` and comment other and un-comment `target = "thumbv7em-none-eabihf" # Cortex-M4F and Cortex-M7F (with FPU)` .

Edit `memory.x`:

```
MEMORY
{
  /* NOTE 1 K = 1 KiBi = 1024 bytes */
  FLASH : ORIGIN = 0x08000000, LENGTH = 64K
```

```
  RAM : ORIGIN = 0x20000000, LENGTH = 16K
}
```

Set up **openocd**, edit: `openocd.cfg`:

```
source [find interface/stlink.cfg]
source [find target/stm32f3x.cfg]
```

It should now build:

```
cargo build
cargo readobj -- --file-headers
```

To see assembly (look for `00000484 <main>`:):

```
cargo objdump --release -- --disassemble --no-show-raw-insn --print-imm-hex
```

QEMU

Install QEMU:

```
xi qemu libnuma
```

Build example program:

```
#![no_std]
#![no_main]

use panic_halt as _;
use cortex_m_rt::entry;
use cortex_m_semihosting::{debug, hprintln};

#[entry]
fn main() -> ! {
    hprintln!("Hello, world!").unwrap();

    // exit QEMU
    // NOTE do not run this on hardware; it can corrupt OpenOCD state
    debug::exit(debug::EXIT_SUCCESS);

    loop {}
```

```
}
```

Build:

```
cargo build --release
```

Run (note we are using `netduinoplus2` device as it uses M4 CPU as well, see: https://qemu.readthedocs.io/en/master/system/arm/stm32.html):

```
qemu-system-arm \
  -cpu cortex-m4 \
  -machine netduinoplus2 \
  -nographic \
  -semihosting-config enable=on,target=native \
  -kernel target/thumbv7em-none-eabihf/release/stm32-nucleo-test
```

It should print `Hello, world!`.

Can also setup `cargo run` to run in QEMU in `.cargo/config.toml`:

```
[target.thumbv7em-none-eabihf]
runner = "qemu-system-arm -cpu cortex-m4 -machine netduinoplus2 -nographic -semihosting-config
enable=on,target=native -kernel"
```

Debugging on hardware

Run `openocd`, it should print like:

```
Info : [stm32f3x.cpu] Cortex-M4 r0p1 processor detected
Info : [stm32f3x.cpu] target has 6 breakpoints, 4 watchpoints
Info : starting gdb server for stm32f3x.cpu on 3333
Info : Listening on port 3333 for gdb connections
Info : accepting 'gdb' connection on tcp/3333
```

Connect debugger pointing it to the built binary (cargo build --release):

```
arm-none-eabi-gdb -ex 'target extended-remote :3333' -q target/thumbv7em-none-
eabihf/release/stm32-nucleo-test
```

Now you can flash the program using `load` command and run with `cont`.

If using **semihosting** (to print stuff in **openocd** window from the device), enable support with `monitor arm semihosting enable` before running the program.

Can also setup `cargo run` to run on device in `.cargo/config.toml`:

```
[target.'cfg(all(target_arch = "arm", target_os = "none"))']
runner = "arm-none-eabi-gdb -q -x openocd.gdb"
```

HAL

For STM32F302R8T6 MCU we can use `stm32f302x8` feature of stm32f3xx-hal crate (x denotes any a to z) see: https://docs.rs/stm32f3xx-hal/latest/stm32f3xx_hal/#target-chip-selection.

Example `Cargo.toml`:

```
[package]
authors = ["Jakub Pastuszek <jpastuszek@protonmail.com>"]
edition = "2018"
readme = "README.md"
name = "stm32-nucleo-test"
version = "0.1.0"

[dependencies]
cortex-m = { version = "0.7.7", features = ["critical-section-single-core"]}
cortex-m-rt = "0.7.3"
cortex-m-semihosting = "0.5.0"
critical-section = "1.1.2"
panic-halt = "0.2.0"
panic-semihosting = "0.6.0"
stm32f3xx-hal = { version = "0.10.0", features = ["stm32f302x8", "rt"] }

# Uncomment for the panic example.
# panic-itm = "0.4.1"

# Uncomment for the allocator example.
# alloc-cortex-m = "0.4.0"

[[bin]]
name = "stm32-nucleo-test"
test = false
```

```
bench = false

[profile.release]
codegen-units = 1 # better optimizations
debug = true # symbols are nice and they don't increase the size on Flash
lto = true # better optimizations
```

Blinking LED2, `src/main.rs`:

```
#![no_std]
#![no_main]

use cortex_m::asm;
use cortex_m_rt::entry;
// pick a panicking behavior
// use panic_halt as _; // you can put a breakpoint on `rust_begin_unwind` to catch panics
// use panic_abort as _; // requires nightly
// use panic_itm as _; // logs messages over ITM; requires ITM support
use cortex_m_semihosting::{dbg, hprintln};
use panic_semihosting as _; // logs messages to the host stderr; requires a debugger

use stm32f3xx_hal::{self as hal, pac, prelude::*};

#[entry]
fn main() -> ! {
    let dp = pac::Peripherals::take().unwrap();

    let mut rcc = dp.RCC.constrain();
    let mut gpiob = dp.GPIOB.split(&mut rcc.ahb);

    let mut led2 = gpiob
        .pb13
        .into_push_pull_output(&mut gpiob.moder, &mut gpiob.otyper);

    loop {
        // hprintln!("Hello, world!");
        led2.toggle().unwrap();
        asm::delay(1_000_000);
    }
}
```

More examples: https://github.com/stm32-rs/stm32f3xx-hal/tree/master/examples

---

Revision #39
Created 2024-02-18 14:18:47 GMT by hxd
Updated 2024-03-12 22:57:38 GMT by hxd