Enable semihosting on ARM for debugging purpose

ARM semihosting is a distinctive feature of the ARM platform, that allows to use input and output functions on a host computer that get forwarded to the microcontrollers over a hardware debugger.

#arm #stm32 #debug #semihosting

Last update: May 4, 2021

Table of Content

Semihosting setup

1. Connect debugger
2. Include semihost lib in GCC linker `-l rdimon --specs=rdimon.specs`
3. Exlcude *syscall.c* implementation
4. Initialize in application `initialise_monitor_handles();`
5. Run OpenOCD with command `monitor arm semihosting enable`

Semihosting notes

Semihosting implementation in OpenOCD is designed so that every string must be terminated with the newline character (`\n`) before the string appears on the OpenOCD console.

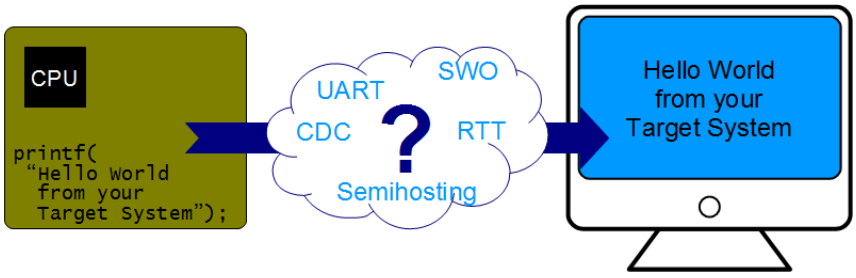
Semihosting only works during a deubug session, and it's slow and affects the system performance.

CPU is halt when semihosting is executing in host machine, therefore semihosting is not suitable for realtime application.

1. Debugging

There are some debug techniques used to inspect the firmware running on ARM-based MCUs:

- Semihosting: build-in to every ARM chips, need adding additional library and running in debug mode
- Redirection: forward to a UART port but need using GPIO and extra hardware (USB to Serial converter), or forward to a Virtual COM port but need USB peripheral
- Instrumentation Trace Macrocell (ITM): fast output over dedicated SWO pin, but it’s only available on Cortex-M3+



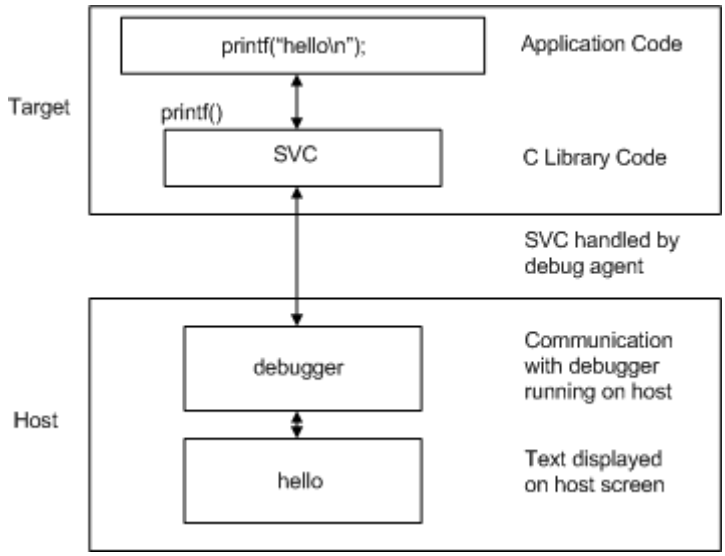
Ways to print debug

2. Semihosting

ARM semihosting is a distinctive feature of the ARM platform, that allows to use input and output functions on a host computer that get forwarded to the microcontrollers over a hardware debugger, such as `printf()` and `scanf()`, or even `fopen()`.

Semihosting is implemented by a set of defined software instructions, for example, `SVCs`, that generate exceptions from program control. The application invokes the appropriate semihosting call and the debugger then handles the exception by communicating with the debugging application on the host computer.

ARM processors prior to ARMv7 use the `SVC` instructions, formerly known as `SWI` instructions, to make semihosting calls. However, for an ARMv6-M or ARMv7-M, in a Cortex-M1 or Cortex-M3 processor, semihosting is implemented using the `BKPT` instruction.



Semihosting overview

2.1. Hardware setup

Semihosting need to be run under a debug session to communicate with semihosting-enabled debugger. In STM32, debugging channel maybe ST-LINK debugger (onboard, or external) which connects to the MCU via `SWCLK` and `SWDIO`.

On STM32F051 Discovery board, ST-LINK module is connected to MCU on pin PA13 and PA14.

2.2. Software setup

To use semihosting, it has to be set in linker options, and initialized in the main program.

2.2.1. Linker options

GNU ARM libs use `newlib` to provide standard implementation of C libraries. However, to reduce the code size and make it independent to hardware, there is a lightweight version `newlib-nano` used in MCUs.

However, `newlib-nano` does not provide an implementation of low-level system calls which are used by C standard libs, such as `print()` or `scan()`. To make the application compilable, a new library named `nosys` should be added. This library just provide an simple implementation of low-level system calls which mostly return a by-pass value.

> The lib `newlib-nano` is enabled via linker options `--specs=nano.specs`, and `nosys` is enabled via linker option `--specs=nosys.specs`. These two libs are included by default in GCC linker options in generated project.

There is a `rdimon` library that implements interrupt for some special system calls, which pause the processor and interact with debugger host to exchange data, such as `SYS_WRITE (0x05)` or `SYS_READ (0x06)`. This library provides low-level system calls to handling the the `newlib-nano` specs. Remove `nosys.specs` when use `rdimon`.

> The lib `rdimon` is enabled via linker option `--specs=rdimon.specs -l rdimon`

Example of using command line:

```
arm-none-eabi-gcc src.c --specs=nano.specs --specs=nosys.specs --
specs=rdimon.specs
```

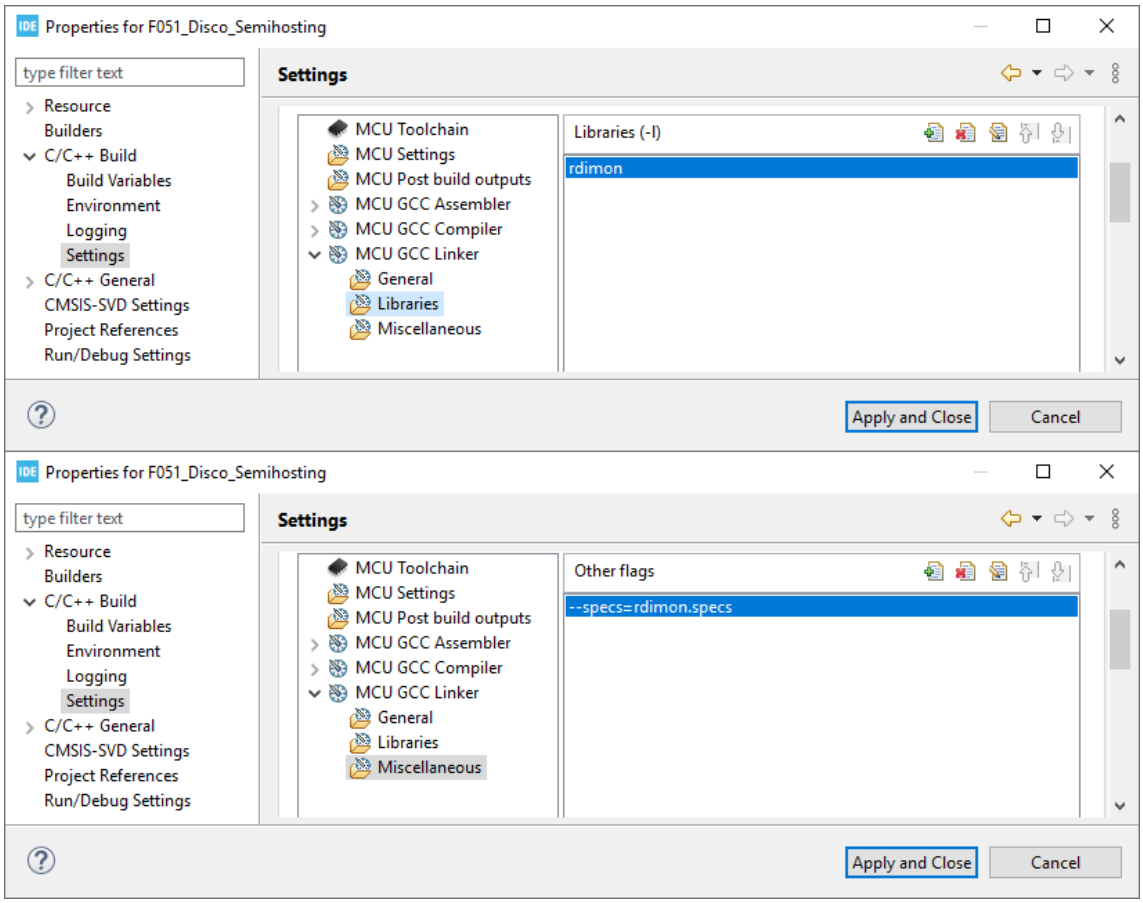
Example of using GUI config:

Open **Project Properties** >> **C/C++ Build** >> **Settings** >> **Tool Settings tab** >> **MCU GCC Linker**:

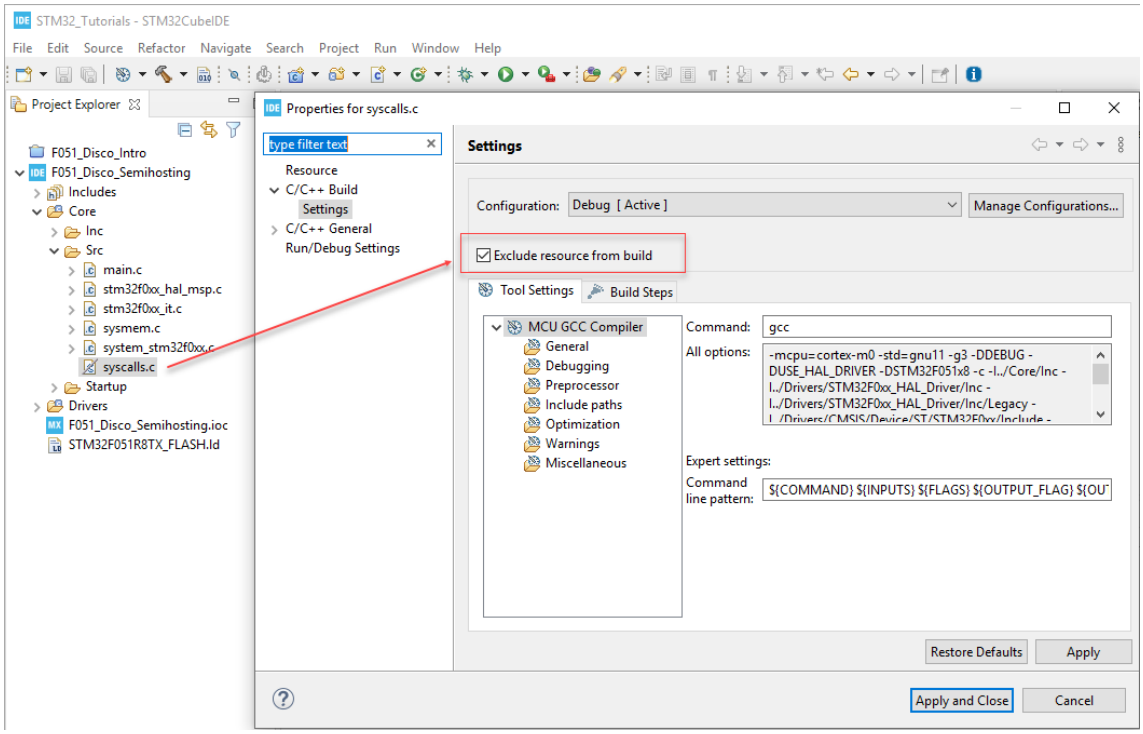
- Libraries: add `rdimon`
- Miscellaneous: add `--specs=rdimon.specs`

2.2.2. Exclude user system calls

In addition, STM32CubeIDE automatically generates `syscalls.c` with a simple implementation for `nosys.specs`. Must exclude `syscalls.c` from build to avoid compilation error of multiple definitions.



Add semihosting in GCC Linker



Exclude syscalls.c

2.2.3. Initialize semihosting

`rdimon` has to be initialized before it can run properly. It exposes a function to do that, then use it:

```
extern void initialise_monitor_handles(void);
```

in the `main()` function:

```
int main(void)
{
  /* USER CODE BEGIN 1 */
  initialise_monitor_handles();
  /* USER CODE END 1 */
}
```

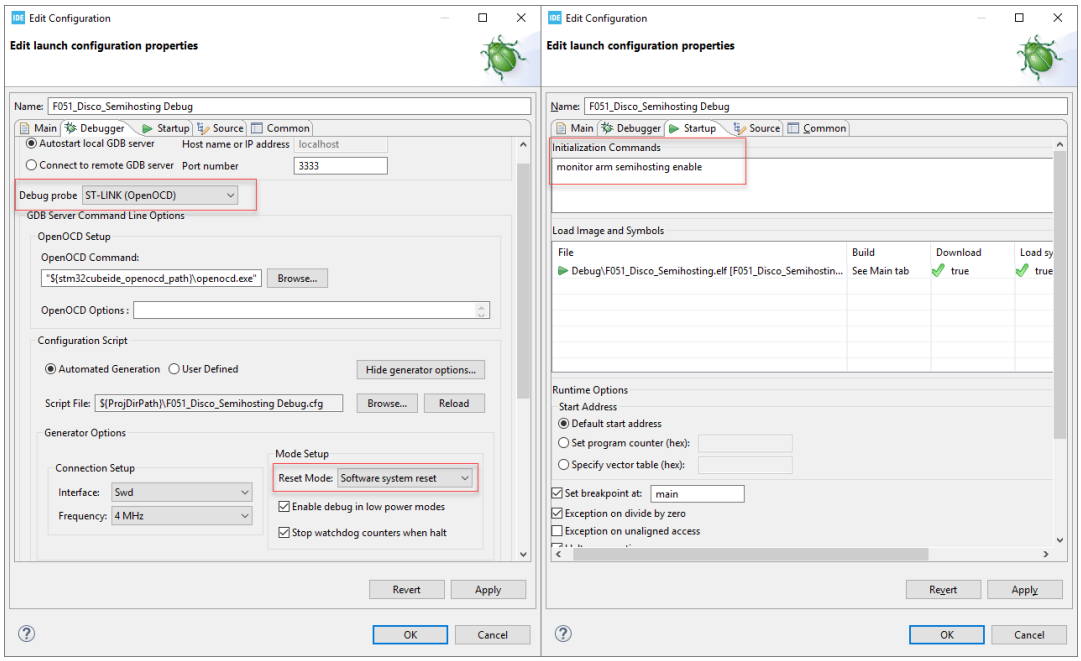
After that, the application can use `printf()`, `scanf()`, or `gets()`.

2.3. Debugger option

The final thing is to enable semihosting on debugger that will handle the interruption fired from MCUs. Debugger has to communicate with MCUs and the host computer.

Use ST-LINK over OpenOCD with the start up command `monitor arm semihosting enable`.

> In some cases, it has to use Software Reset!



Enable semihosting in debugger

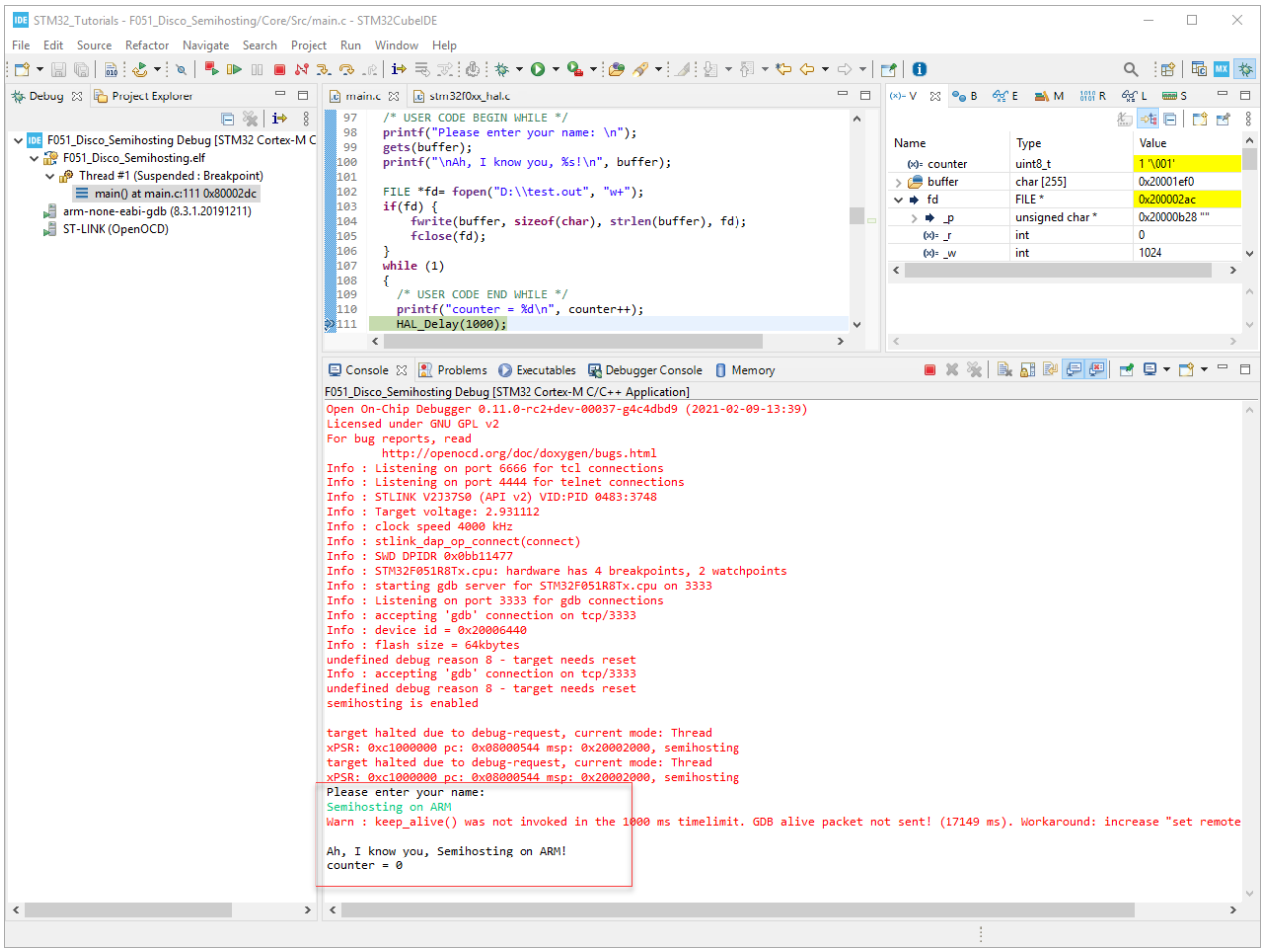
3. Debug with Semihosting

Run the project in debug mode and then interact with MCUs. Here are some lines of code to print a message, get a string, and write to a file on the host machine:

main.c

```
#include "main.h"
#include <stdio.h>
#include <string.h>

extern void initialise_monitor_handles(void);

int main(void)
{
    uint8_t counter = 0;
    char buffer[255];

    initialise_monitor_handles();

    printf("Please enter your name: \n");
    gets(buffer);
    printf("\nAh, I know you, %s!\n", buffer);

    // test.out will be created in the host machine
    FILE *fd= fopen("D:\\test.out", "w+");
    if(fd) {
        fwrite(buffer, sizeof(char), strlen(buffer), fd);
        fclose(fd);
    }

    while (1)
    {
        printf("counter = %d\n", counter++);
        HAL_Delay(1000);
    }
}
```

When using `fopen()`, should specify the location, such as ~/test.out or D:\test.out, if not, the target file will be created in the folder containing openocd.exe.



Interact with semihosting