

# Microchip PolarFire SoC tutorial

Created by: David Rubio G.

Blog post: <https://soceame.wordpress.com/2025/03/11/microchip-polarfire-soc-tutorial/>

Blog: <https://soceame.wordpress.com/>

GitHub: <https://github.com/DRubioG>

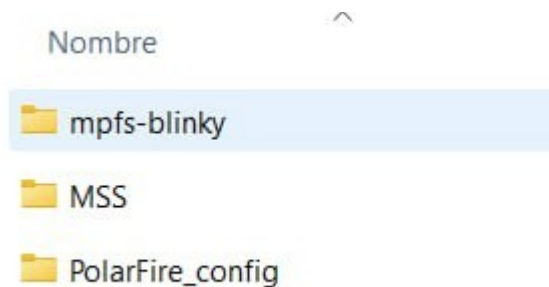
Last modification date: 11/03/25

## Before you start

If you also work with a PolarFire SoC Discovery Kit I recommend you download this GitHub repository. It has the base configuration to enable the MSS processor and also has an example project that we will work with, which also includes all the drivers you may need for a PolarFire. If you do not work with this kit I also recommend you download it because the SoftConsole base project is common to all boards.

[https://github.com/DRubioG/Polarfire\\_basic\\_project](https://github.com/DRubioG/Polarfire_basic_project)

We will only need the mpfs-blinky and the PolarFire\_config. The other file is like the PolarFire\_config but applied to the complete Discovery Kit.



## Libero

The first step is to create a new project in Libero.



We give the project a name and a location.

The screenshot shows the 'New project' dialog in the Libero IDE. The 'Project details' tab is selected in the left sidebar. The main area contains the following fields:

- Project name:** prueba\_PolarFire
- Project location:** C:/Libero\_test (with a 'Browse...' button)
- Description:** (empty text box)
- Preferred HDL type:** Verilog (dropdown menu)
- ☐ **Enable block creation**  
Block flow enables you to publish a reusable component that can be instantiated into another design. A block component may not contain I/O cells and cannot be programmed by itself. It could include timing constraints, physical constraints, placement or routing.

At the bottom, there are buttons for '< Back', 'Next >', 'Finish', and 'Cancel'. A 'Help' button is also present in the bottom left corner.

Then we choose the chip and click *Finish*.

The screenshot shows the 'New project' dialog in the Libero IDE, now on the 'Device selection' tab. The 'Selected part' is MPFS095T-1FCSG325E. The 'Part filter' section shows the following settings:

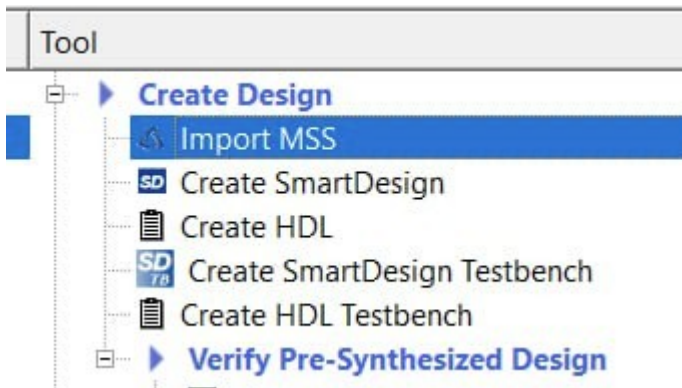
- Family:** PolarFireSoC
- Die:** All
- Package:** All
- Speed:** All
- Range:** All

A 'Reset filters' button is located below the filters. Below the filters is a 'Search part:' text box and a table of available parts.

Part Number	DFF	User I/Os	uSRAM	LSRAM	Math	H-Chip Globals	PLL
MPFS025T-FCVG484E	22956	108	204	84	68	24	8
MPFS025T-FCVG484I	22956	108	204	84	68	24	8
MPFS025TL-FCSG325E	22956	80	204	84	68	24	8
MPFS025TL-FCSG325I	22956	80	204	84	68	24	8
MPFS025TL-FCVG484E	22956	108	204	84	68	24	8
MPFS025TL-FCVG484I	22956	108	204	84	68	24	8
MPFS095T-1FCSG325E	93516	80	876	308	292	48	8
MPFS095T-1FCSG325I	93516	80	876	308	292	48	8

At the bottom, there are buttons for '< Back', 'Next >', 'Finish', and 'Cancel'. A 'Help' button is also present in the bottom left corner.

We click *Import MSS* and import a .cxz that is from the repository.



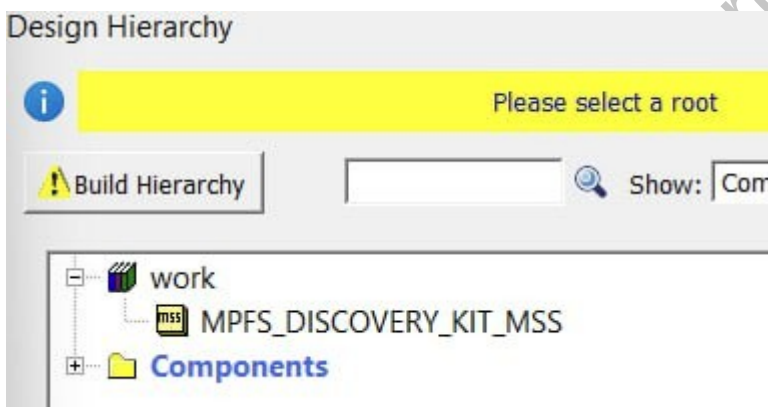
The one you need to import.

MPFS\_DISCOVERY\_KIT\_MSS.cxz

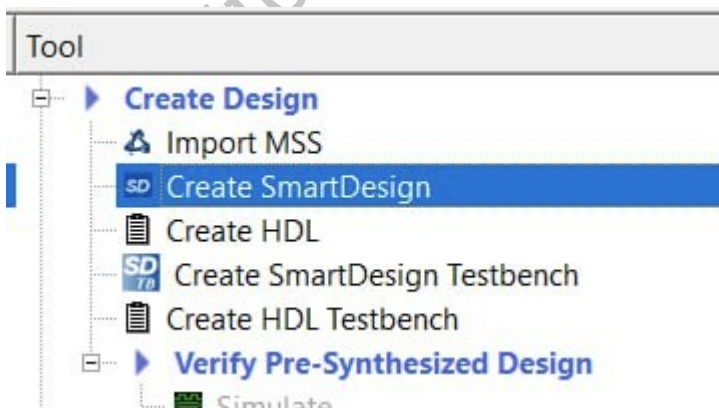
To create your own you have this tutorial.

<https://soceame.wordpress.com/2025/03/11/configuring-the-peripherals-of-a-polarfire-soc/>

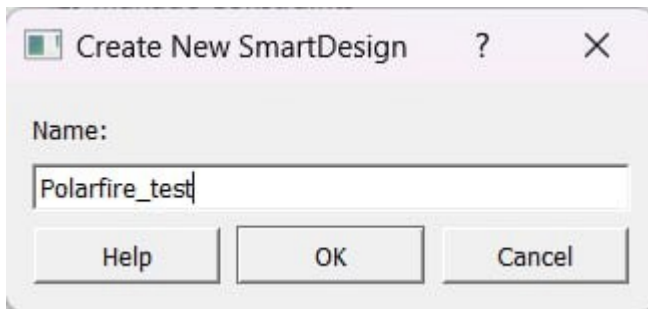
In the hierarchy we have the following.



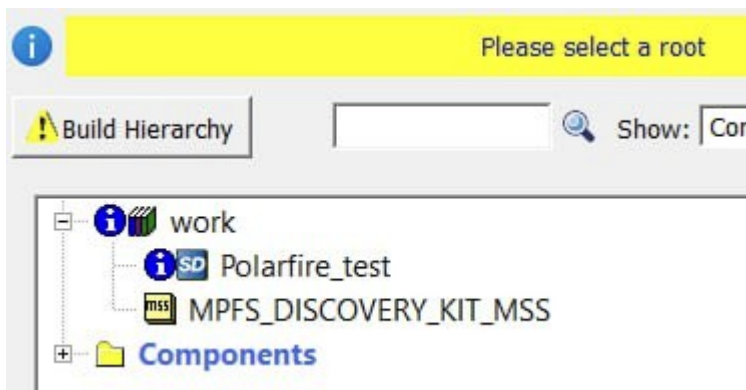
Now we create a SmartDesign.



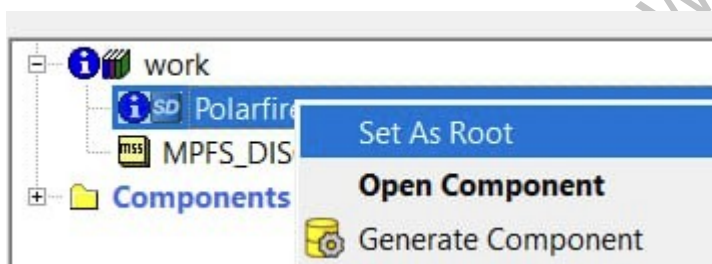
We call it *Polarfire\_test*.



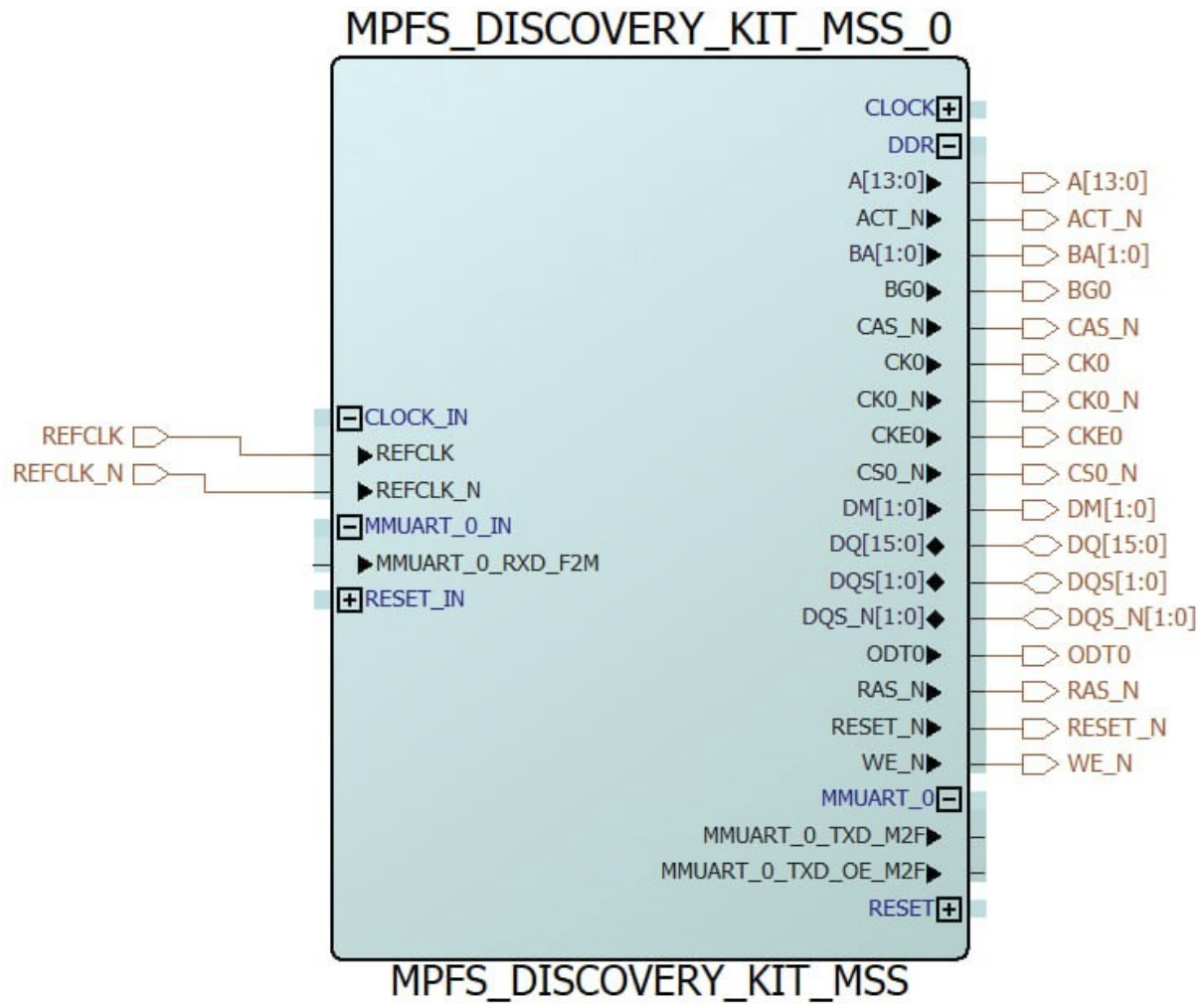
We get a structure like this.



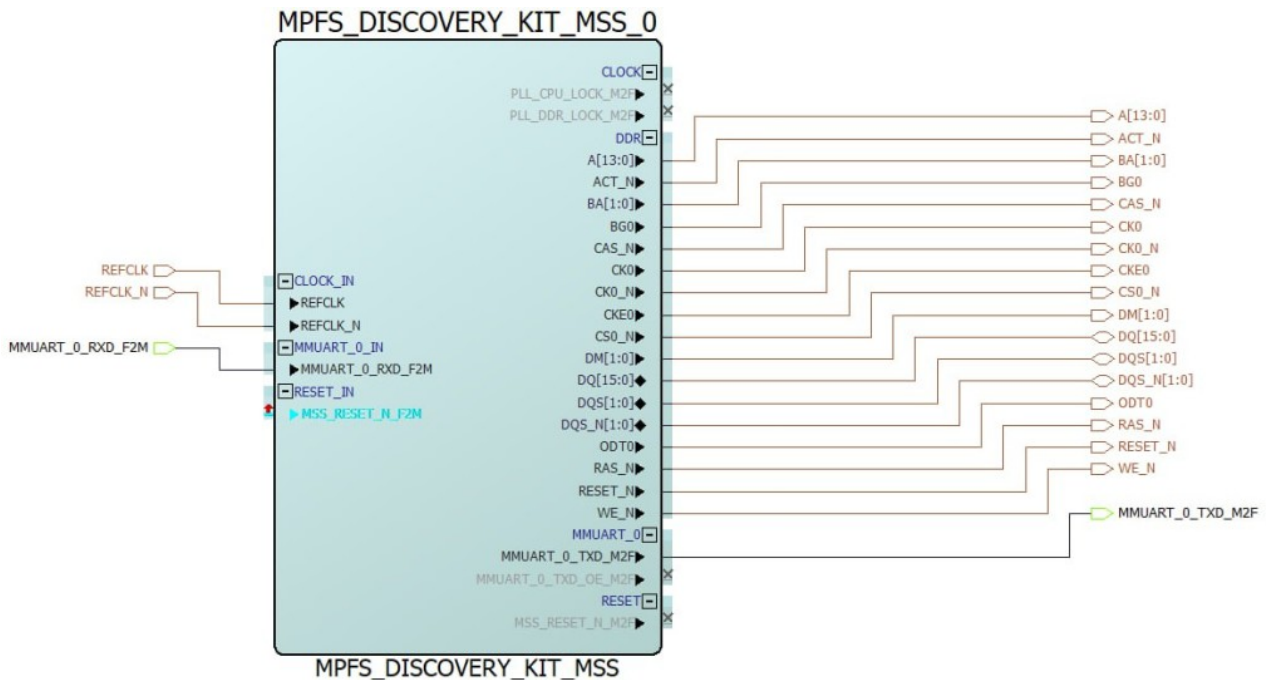
We select the *SmartDesign* as *Root*.



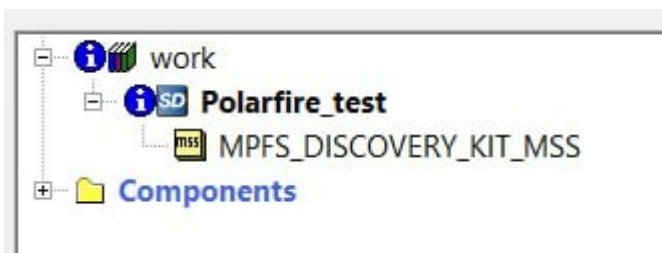
Now we drag the PolarFire SoC model to the SmartDesign.



Now we connect only the UART ports that are as Fabric.



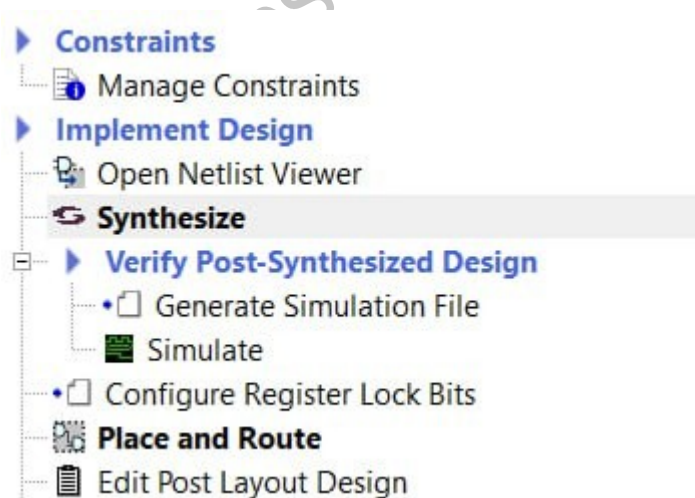
Now we have a structure like the following.



If you want to connect other IP blocks you can do it by following this.

<https://soceame.wordpress.com/2025/03/11/how-to-connect-an-ip-block-in-libero/>

When we have everything, we synthesize.





Now in *Manage Constraints* we can configure the UART pins as Fabric.

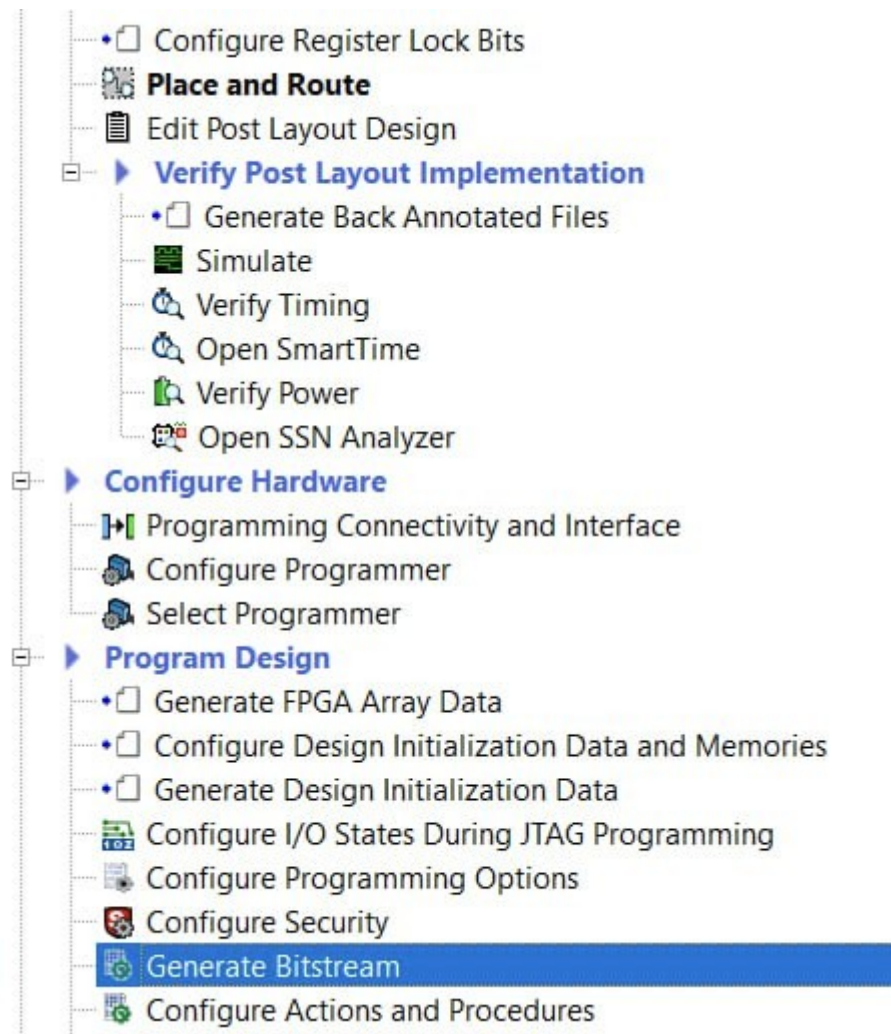
47	DQS_N[0]	INOUT	POD12I	Y10	<input checked="" type="checkbox"/>	BIBUF_DIFF
48	DQS[1]	INOUT	POD12I	V12	<input checked="" type="checkbox"/>	BIBUF_DIFF
49	DQS_N[1]	INOUT	POD12I	U12	<input checked="" type="checkbox"/>	BIBUF_DIFF
50	MMUART_0_RXD_F2M	INPUT	LVC MOS18		<input type="checkbox"/>	INBUF
51	MMUART_0_TXD_M2F	OUTPUT	LVC MOS18		<input type="checkbox"/>	OUTBUF
52	ODT0	OUTPUT	HSTL12I	T2	<input checked="" type="checkbox"/>	IOPAD_TRI
53	RAS_N	OUTPUT	HSTL12I	AA4	<input checked="" type="checkbox"/>	IOPAD_TRI
54	REFCLK	INPUT	LVDS25	P5	<input checked="" type="checkbox"/>	INBUF_DIFF
55	REFCLK_N	INPUT	LVDS25	P4	<input checked="" type="checkbox"/>	INBUF_DIFF

Now we select them.

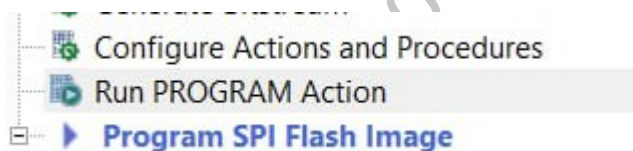
47	DQS_N[0]	INOUT	POD12I	Y10
48	DQS[1]	INOUT	POD12I	V12
49	DQS_N[1]	INOUT	POD12I	U12
50	MMUART_0_RXD_F2M	INPUT	LVC MOS18	W21
51	MMUART_0_TXD_M2F	OUTPUT	LVC MOS18	Y21
52	ODT0	OUTPUT	HSTL12I	T2

Once selected we can generate the bitstream.





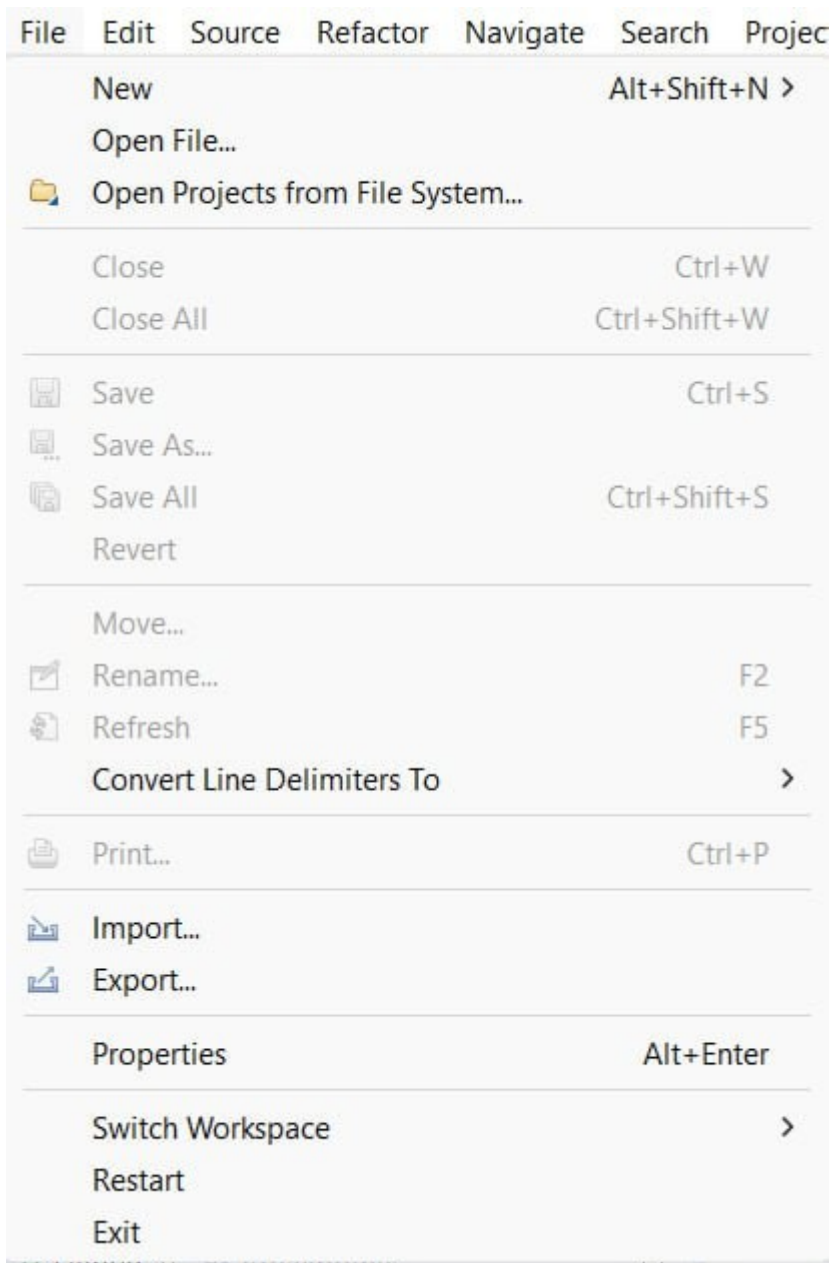
Once we have it we can record it to the SoC.



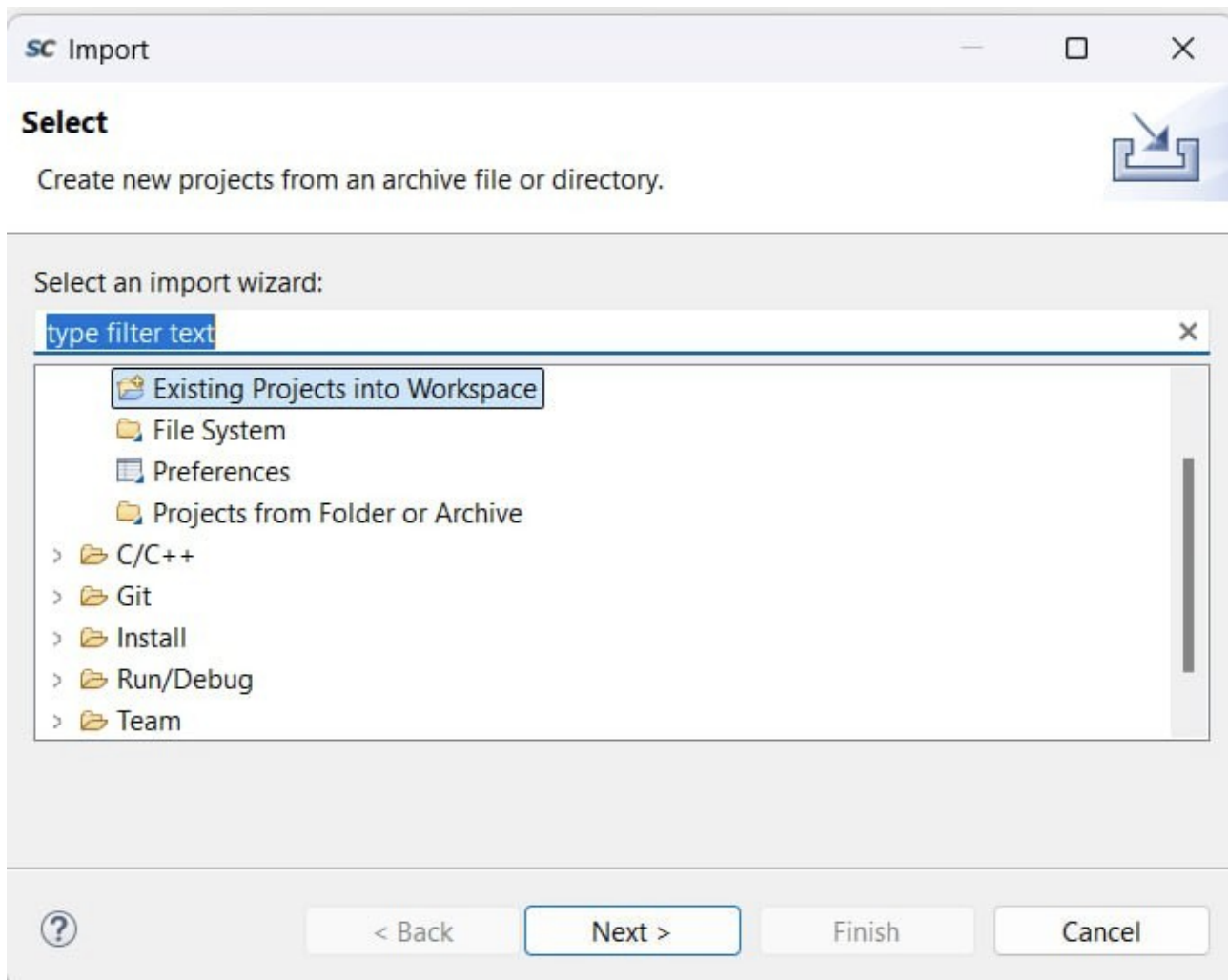
Now we go to the SoftConsole.

## SoftConsole

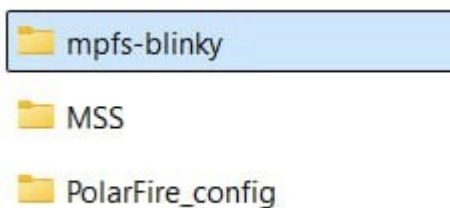
Now we import a project, to do this we click on *Import...*



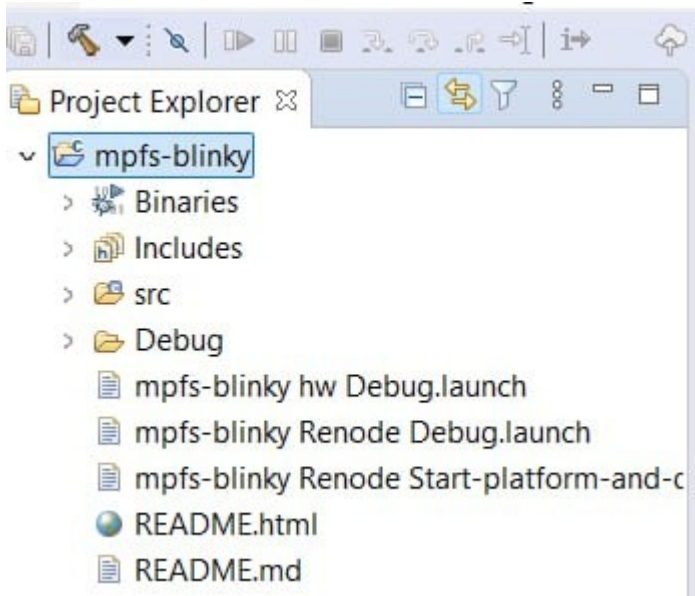
Then we click on Importing an *Existing Projects into Workspace*.



Now we look for the mpfs-blinky, and we import it (you may have to go down one more directory so that the SoftConsole project is supported).



Once we have imported it, we compile it.

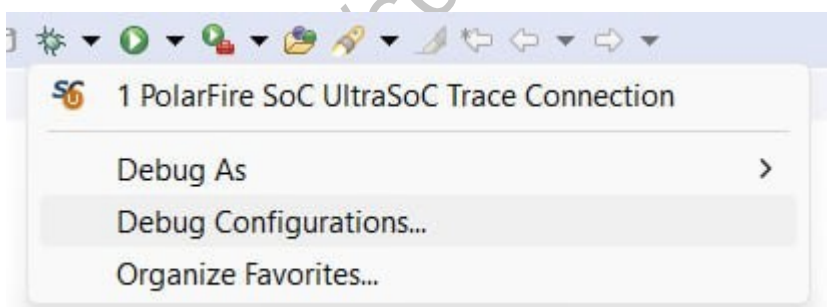


Once we have compiled it we can debug it.

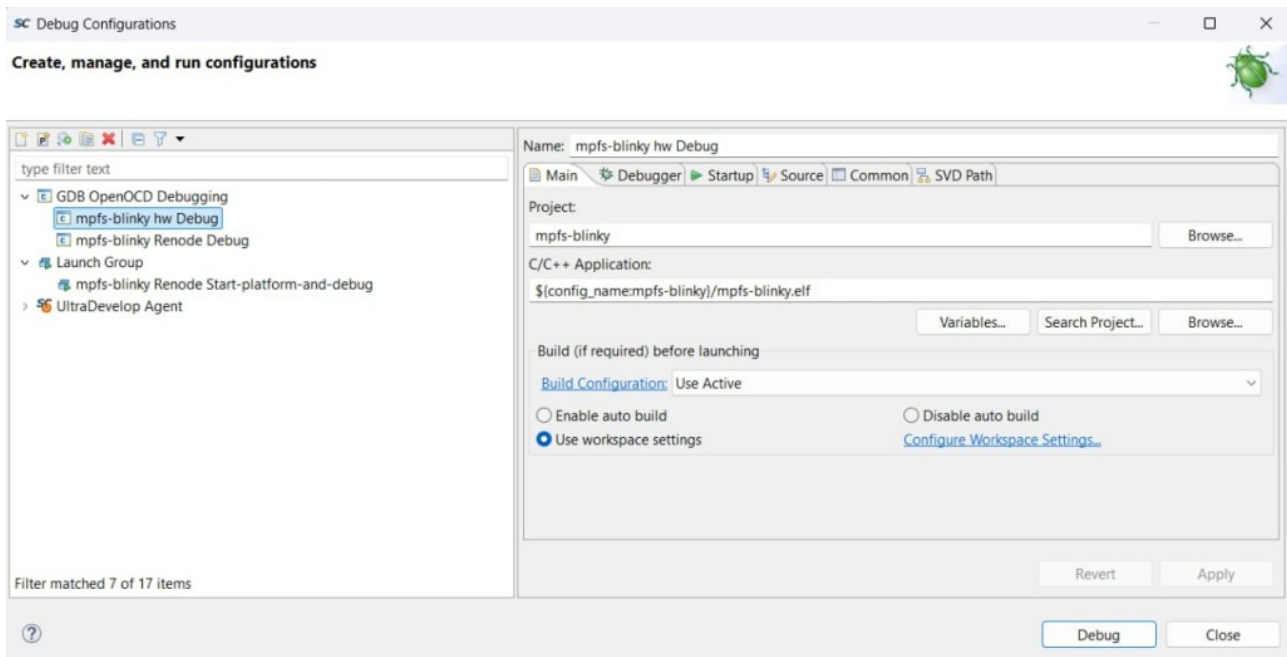
```
Invoking: GNU RISC-V Cross Print Size
riscv64-unknown-elf-size --format=berkeley "mpfs-blinky.elf"
   text    data    bss     dec     hex filename
 19056   3648   12560   35264   89c0 mpfs-blinky.elf
Finished building: mpfs-blinky.siz
```

23:03:50 Build Finished. 0 errors, 0 warnings. (took 5s.689ms)

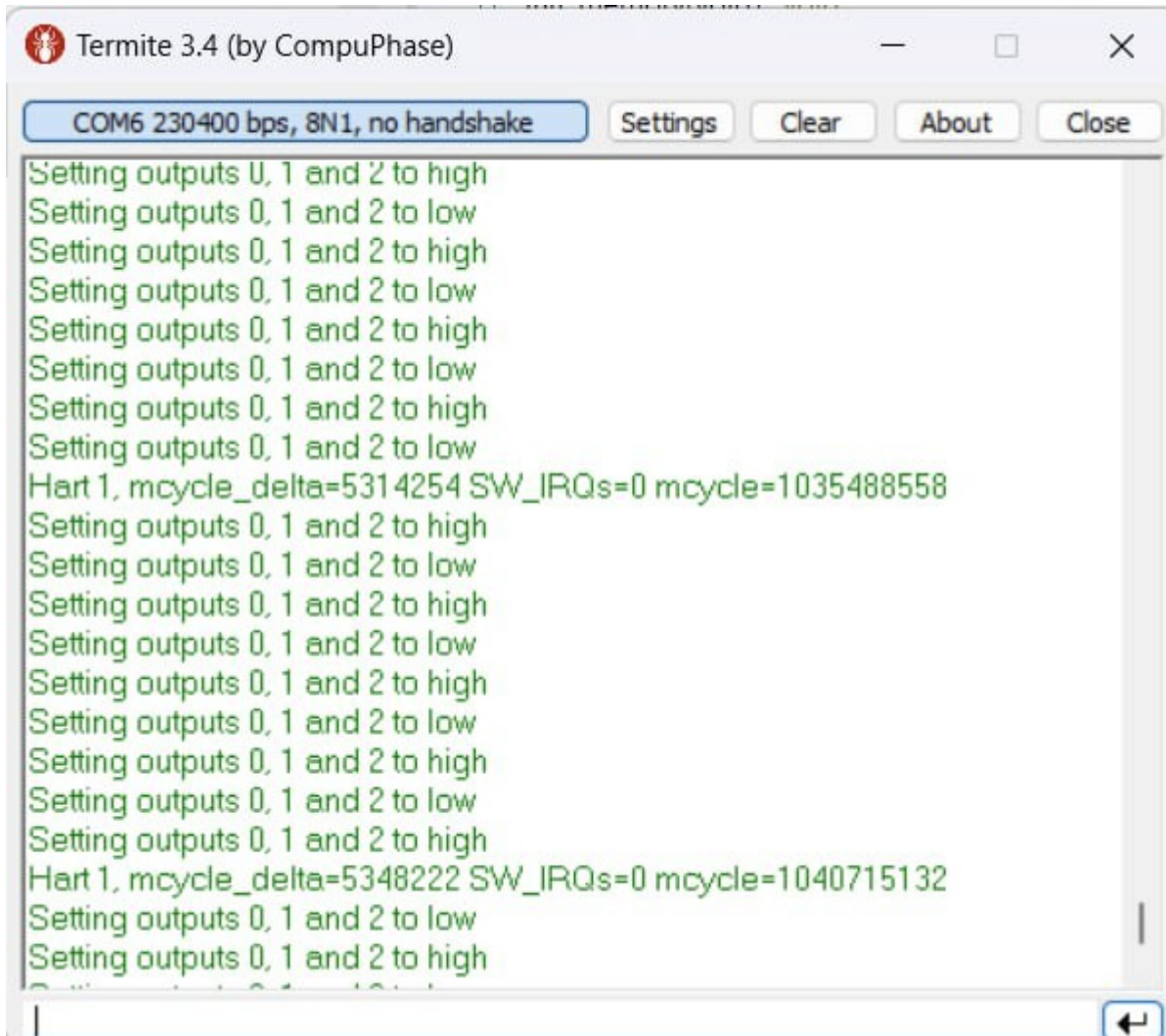
To do this we go to *Debug Configurations...*



Now we look for the *mpfs-blinky hw Debug*, and we launch it.



If we open the terminal we can see that the application runs correctly.



The screenshot shows a terminal window titled "Termite 3.4 (by CompuPhase)". The window has a menu bar with "Settings", "Clear", "About", and "Close". The status bar at the top indicates "COM6 230400 bps, 8N1, no handshake". The terminal displays the following text in green:

```
Setting outputs 0, 1 and 2 to high
Setting outputs 0, 1 and 2 to low
Setting outputs 0, 1 and 2 to high
Setting outputs 0, 1 and 2 to low
Setting outputs 0, 1 and 2 to high
Setting outputs 0, 1 and 2 to low
Setting outputs 0, 1 and 2 to high
Setting outputs 0, 1 and 2 to low
Hart 1, mcycle_delta=5314254 SW_IRQs=0 mcycle=1035488558
Setting outputs 0, 1 and 2 to high
Setting outputs 0, 1 and 2 to low
Setting outputs 0, 1 and 2 to high
Setting outputs 0, 1 and 2 to low
Setting outputs 0, 1 and 2 to high
Setting outputs 0, 1 and 2 to low
Setting outputs 0, 1 and 2 to high
Setting outputs 0, 1 and 2 to low
Hart 1, mcycle_delta=5348222 SW_IRQs=0 mcycle=1040715132
Setting outputs 0, 1 and 2 to low
Setting outputs 0, 1 and 2 to high
```

If we analyze the code, we can see that there is a UART and some GPIOs in the example. The GPIOs do not work (*I have not managed to get a PolarFire to be able to run, because it does leave something with the GPIOs assigned from the PolarFire*).



```
safe_MSS_UART0_polled_tx_string("Hello World from e51 (hart 0).\r\n");

while (1)
{
    // Stay in the infinite loop, never return from main

    const uint64_t delay_loop_max = 10000;
    volatile uint64_t delay_loop_sum = 0;

    for (uint64_t i = 0; i < delay_loop_max; i++) {
        delay_loop_sum = delay_loop_sum + i;
    }

    safe_MSS_UART0_polled_tx_string("Setting outputs 0, 1 and 2 to high\r\n");

    MSS_GPIO_set_output(GPIO1_LO, MSS_GPIO_0, 1);
    MSS_GPIO_set_output(GPIO1_LO, MSS_GPIO_1, 1);
    MSS_GPIO_set_output(GPIO2_LO, MSS_GPIO_9, 1);

    for (uint64_t i = 0; i < delay_loop_max; i++) {
        delay_loop_sum = delay_loop_sum + i;
    }

    safe_MSS_UART0_polled_tx_string("Setting outputs 0, 1 and 2 to low\r\n");
    MSS_GPIO_set_output(GPIO1_LO, MSS_GPIO_0, 0);
    MSS_GPIO_set_output(GPIO1_LO, MSS_GPIO_1, 0);
    MSS_GPIO_set_output(GPIO2_LO, MSS_GPIO_9, 0);
}
```

The UART operation looks easy, but an intermediate UART appears. That comes from here.



```
void u54_1_application(void)
{
    time_benchmark_t mcycle      = { .start = 0, .end = 0, .delta = 0};
    volatile uint64_t loop_count_h1 = 0;
    const uint64_t num_loops      = 100000;
    uint64_t hartid               = read_csr(mhartid);
    char uart_buf[100];

    while (1)
    {
        mcycle.start = readmcycle();

        for (uint64_t i = 0; i < num_loops; i++) {
            dummy_h1 = i;
        }

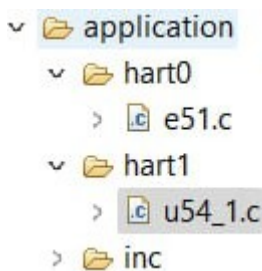
        sprintf(uart_buf, "Hart %ld, mcycle_delta=%ld SW_IRQs=%ld mcycle=%ld\r\n",
                hartid, mcycle.delta, count_sw_ints_h1, readmcycle());

        safe_MSS_UART0_polled_tx_string(uart_buf);

        hartid      = read_csr(mhartid);
        mcycle.end   = readmcycle();
        mcycle.delta = mcycle.end - mcycle.start;

        loop_count_h1++;
    }
}
```

The last thing we have seen is the execution on a second core, so the application can only be executed with two cores, e51 and e54\_1, (if you try to remove one of the two cores, which seems easy but is not even close, you run the risk of everything stopping working, and the board going «dumb», and you having to quit everything and return).



**NOTE:** if you look at the UART baud rate, you can see that it has been set to 115200 baud, but the Terminate is set to 230400, this is because this SoC, like the SmartFusion2, has a design problem, which theoretically is solved at the driver level, but which Microchip has not wanted to solve. This problem is that the core frequency has to be a specific one that Microchip never says, so if the frequency that the core needs is 50MHz, but you feed it at 100MHz, the board doubles its frequency at the peripheral level making a 115200 UART go to 230400 (this happens at all UART baud rates). This is supposed to be solved at the driver level, but it isn't, so it's important to keep this in mind.

## FINAL NOTE

Working with a PolarFire SoC may look like a SmartFusion2, but it's not. PolarFire SoCs at the time of writing this are extremely difficult to handle. So if you know how to handle a SmartFusion2, stick with it.

<https://soceame.wordpress.com/>