RISC-V on MAX10 User Guide

Version: **1.0**

Created on: Feb 14, 2022

Created by: Johannes Schwenk

CONTENTS:

CHAPTER ONE

ABOUT THIS MANUAL

1.1 Imprint

Adress:

ARIES Embedded GmbH

Schöngeisinger Str. 84

D-82256 Fürstenfedbruck

Germany

Phone:

+49 (0) 8141/36 367-0

Fax:

+49 (0) 8141/36 367-67

1.2 Disclaimer

ARIES Embedded does not guarantee that the information in this document is up-to-date, correct, complete or of good quality. Liability claims against ARIES Embedded, referring to material or non-material related damages caused, due to usage or non-usage of the information given in this document, or due to usage of erroneous or incomplete information, are exempted, as long as there is no proven intentional or negligent fault of ARIES Embedded. ARIES Embedded explicitly reserves the rights to change or add to the contents of this Preliminary User Guide or parts of it without notification.

1.3 Copyright

This document may not be copied, reproduced, translated, changed or distributed, completely or partially in any form without the written approval of ARIES Embedded GmbH.

1.4 Registered Trademarks

The contents of this document may be subject of intellectual property rights (including but not limited to copyright, trademark, or patent rights). Any such rights that are not expressly licensed or already owned by a third party are reserved by ARIES Embedded GmbH.

1.5 Care and Maintenance

- Keep the device dry. Precipitation, humidity, and all types of liquids or moisture can contain minerals that will corrode electronic circuits. If your device does get wet, allow it to dry completely.
- Do not use or store the device in dusty, dirty areas. Its moving parts and electronic components can be damaged.
- Do not store the device in hot areas. High temperatures can shorten the life of electronic devices, damage batteries, and warp or melt certain plastics.
- Do not store the device in cold areas. When the device returns to its normal temperature, moisture can form inside the device and damage electronic circuit boards.
- Do not attempt to open the device.
- Do not drop, knock, or shake the device. Rough handling can break internal circuit boards and fine mechanics.
- Do not use harsh chemicals, cleaning solvents, or strong detergents to clean the device.
- Do not paint the device. Paint can clog the moving parts and prevent proper operation.
- Unauthorized modifications or attachments could damage the device and may violate regulations governing radio devices.

1.6 Change Log

Revision	Date	Revised	Comment
1.0	14.02.2022	js	Initial creation

CHAPTER

TWO

INTRODUCTION

The reference designs demonstrate the implementation of an open-source RISC-V core running FreeRTOS and interfacing with different peripherals. This guide shows how to install the neccessacy requirements and how to run and modify the RISC-V examples on the MX10 and SpiderBoard SoMs. The examples are also suitable as starting point for developement.

2.1 Cores

The following open source cores are available as Intel Platform Designer (Qsys) Component:

- Serv

  The Serv core by Olof Kindgren is a bit-serial RV32I core. By only handling one bit at a time, the core trades performance for its small size. An additional memory-mapped interrupt controller is connected to the core to enable external, software and configurable timer interrupts similar to as described in the RISC-V specification. As such the Serv is fully capable of running FreeRTOS.

- PicoRV

  The PicoRV32 core by Claire Wolf implements the RV32I[M][C] architecture. The core connects to the avalon-interconnect via its native memory interface. The external PCPI, look-ahead and trace interface are not connected. The core implements its own native interrupt controller.

- VexRiscv

  The VexRiscv core by SpinalHDL is written in Scala, highly configurable and builds to Verilog via SpinalHDL. Five different variants were built and merged into one component to allow specification of the supported architecture. RV32I[M] and RV32IM[A[F]C] with data and instruction caches are available. Similar to the Serv a memory mapped interrupt controller is implemented.

2.1.1 RISC-V Core Benchmark

The following tests were conducted on the MX10-U (with 10M50DAF256I7G, Speedgrade 7) using Quartus 20.1 Lite.

Quartus Compilation effort was set to **Performance (Aggressive)** with most options that increase maximum frequency turned on. The clock frequency of the FPGA system was 25 MHz. The benchmarks were compiled using GCC version 11.1.0 with the compiler flags `-O3`

2.1.1.1 Dhrystone

Core @ 25 MHz	Dhrystones/s	DMIPS	DMIPS/MHz	f_{max}	$DMIPS_{max}$
Serv (RV32I)	1262	0.718	0.028	135.8 MHz	3.938
PicoRV Small (RV32I)	13347	7.596	0.303	130.3 MHz	40.263
PicoRV (RV32IM)	14705	8.369	0.334	123.9 MHz	42.250
VexRiscv (RV32IM)	48449	27.574	1.102	86.8 MHz	97.824
VexRiscv+Cache (RV32IMAFC)	61950	35.258	1.410	77.3 MHz	112.626

2.1.1.2 CoreMark

Core @ 25 MHz	Iterations	CoreMark	CM/MHz	f_{max}	$CoreMark_{max}$
Serv (RV32I)	10	0.590	0.024	135.8 MHz	3.123
PicoRV32 Small (RV32I)	110	6.351	0.254	130.3 MHz	32.705
PicoRV32 (RV32IM)	200	16.577	0.663	123.9 MHz	81.774
VexRiscv (RV32IM)	600	50.123	2.005	86.8 MHz	172.298
VexRiscv+Cache (RV32IMAFC)	1100	61.811	2.472	77.3 MHz	189.076

2.1.2 FPGA Resource Usage

For the resource usage statistics the Quartus Compilation effort was set to **Area**. The data was taken from the fitter report.

Core	Logic Cells (Total)	Logic Cells (Core)	M9K / Bits (Core)
Serv (RV32I)	1126	383	1 / 1152
PicoRV Small (RV32I)	2929	2523	0
PicoRV (RV32IM)	3148	2766	2 / 2340
VexRiscv (RV32IM)	3243	2404	2 / 2048
VexRiscv+Cache (RV32IMAFC)	9151	7537	21 / 131520

Note: Results are specific to exact compilation of the FPGA design and firmware, results may not be reproducable.

CHAPTER

THREE

REQUIREMENTS

3.1 MAX10 SoMs

To run the RISC-V demos on a MAX10 board one of the following SoMs is required:

- SpiderSoM-S (10M08SAU169C8G)
- MX10-S8 (10M08DAF256C8G)
- MX10-U (10M50DAF256I7G)

To compile the firmware the RISC-V GCC toolchain is required.

To build the hardware design Intel Quartus Prime is required.

To programm the image on the board either a JTAG USB-Blaster (with Quartus Programmer) or OpenOCD is required.

As reference for this guide Linux Ubuntu 20.04 is used, other Linux distributions may work similarly. While the toolchain can be used on Windows only limited support is available.

3.2 Reference Designs

The first step is to to download the reference designs using git. Open a terminal window to clone the repository with the command:

```
git clone https://github.com/ARIES-Embedded/riscv-on-max10
```

3.3 RISC-V GCC

This guide installs the toolchain under `/opt/riscv`, this path is configurable. For other Linux distributions the toolchain can be installed similarly. For more information, please visit the official RISC-V GNU Compiler Toolchain repository.

Note: To use the RISC-V GCC toolchain on Windows the Windows Subsystem for Linux is recommended. The guide for Ubuntu then applies.

The first step is to download the prerequisites. Open a terminal window and enter the following command:

```
sudo apt-get install autoconf automake autotools-dev curl python3 libmpc-dev \
libmpfr-dev libgmp-dev gawk build-essential bison flex texinfo gperf libtool \
patchutils bc zlib1g-dev libexpat-dev
```

Navigate to a temporary directory and clone the toolchain from github:

```
git clone https://github.com/riscv/riscv-gnu-toolchain
cd riscv-gnu-toolchain
```

Configure the build for the available architectures and run make to start the build:

Note: This step may take a while.

```
./configure --prefix=/opt/riscv --with-multilib-generator="rv32i-ilp32--;"\
"rv32im-ilp32-rv32ima-;rv32imc-ilp32-rv32imac-;rv32imafc-ilp32f--"
sudo make
```

Add the build tools to the path by opening `~/.bashrc` (or equivalent) and add the line:

```
export PATH="$PATH:/opt/riscv/bin"
```

Finally reload the terminal with the following command:

```
source ~/.bashrc
```

Now the RISC-V tools are available on the terminal via `riscv64-unknown-elf-(*)`

3.4 Intel Quartus Prime

To synthesize the hardware design Intel Quartus Prime is required. The lite edition is available from Intel free of charge, please make sure that the MAX10 Device support is included.

To make the RISC-V cores available for the Intel Platform Designer open Quartus and under the menu **Tools** select **Options**. There select **IP Settings > Ip Catalog Search Locations** and add the the following path to the Global IP search directory, substituting the path to the repository previously cloned:

<path to repository>/Cores/**/*

The MAX10 SoM is programmed via JTAG either through the intregrated PIC microcontroller using a Serial Vector Format (.svf) file or through a Quartus Prime compatible USB-Blaster connected to the JTAG Header on the Spider Baseboard.

3.5 OpenOCD

The PIC onboard programming solution is used in conjunction with OpenOCD. For OpenOCD, the libftdi driver with blaster support is required.

3.5.1 Linux

On Linux the `apt` version usually suffices:

```
sudo apt install openocd
```

Create a bash script to programm the FPGA more conveniently. In order to do so, create the file `~/.local/bin/mx10spider` (including parent directories, should they not exist) and add the following content:

```
#!/bin/sh

me=$(basename $0)

if [ -f "$1" ]; then
openocd -c "interface usb_blaster" -c "usb_blaster_lowlevel_driver ftdi" \
        -c "usb_blaster_vid_pid 0x04d8 0xefd0" -c "jtag newtap max10 tap
        -irlen 10 -expected-id 0x31810dd -expected-id 0x318a0dd \
        -expected-id 0x31820dd -expected-id 0x31830dd -expected-id 0x31840dd \
        -expected-id 0x318d0dd -expected-id 0x31850dd -expected-id 0x31010dd \
        -expected-id 0x310a0dd -expected-id 0x31020dd -expected-id 0x31030dd \
        -expected-id 0x31040dd -expected-id 0x310d0dd -expected-id 0x31050dd" \
        -c "init" -c "svf $1 progress" -c "shutdown"
elif [ "$1" = "" ]; then
echo "\tError: No file specified.\n\tUsage: $me <file.svf>"
elif [ "$1" = "-h" ] || [ "$1" = "--help" ]; then
echo "\tUtility script to start openocd and run an svf file.\n\tUsage: $me <file.svf>"
else
echo "\tFile not found: $1\n\tUsage: $me <file.svf>"
fi
```

Make sure the directory is included in the path variable.

Then the FPGA can be programmed via a Serial Vector Format File (.svf) using the following example command:

```
mx10spider example.svf
```

3.5.2 Windows

For Windows OpenOCD binaries are available on the GitHub repository. Extract the content of the archive to any directory (for example **C:/openocd**), then add the subdirectory **bin/** to the path environment variable. OpenOCD can be used by running a bash-emulation such as MINGW64 (for example shipped with Git for Windows). Create the file **mx10spider** in the subdirectory **bin/** of OpenOCD and insert the bash script for Linux from above. Then the FPGA can be programmed in the same way as on Linux.

3.6 Serial Console

The reference design implements UART connected via PIC-USB to the host PC. To use the serial communication to the FPGA a console emulator is required.

3.6.1 Linux

One can install **picocom** on Linux and add the user to the dialout-group using the following commands on the terminal:

```
sudo apt install picocom
sudo usermod -a -G dialout $USER
```

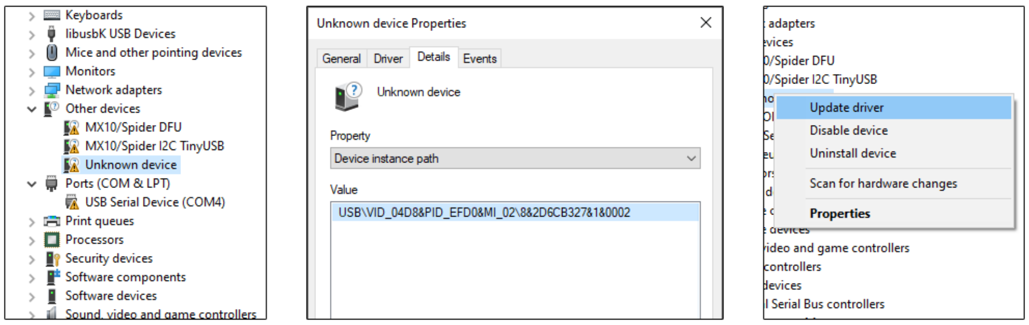
Note: Changes of the user's groups may require a relog or reboot.

The UART on the FPGA uses a fixed baudrate of 115200, connect to the serial port with the following command. **ttyACM0** refers to the default device name, it may be different per user.

```
picocom -b 115200 /dev/ttyACM0
```

3.6.2 Windows

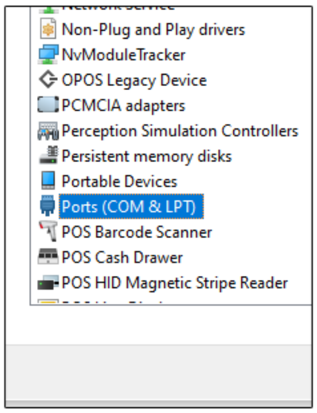
On Windows the serial driver for the interface is required to be manually installed. Open the Windows Device Manager and under **Other devices** select **Unknown device**. Verify that it is the correct device by looking at its properties, **Device instance path** should read something similar to `USB\VID_04D8&PID_EFD0&MI_02\...`

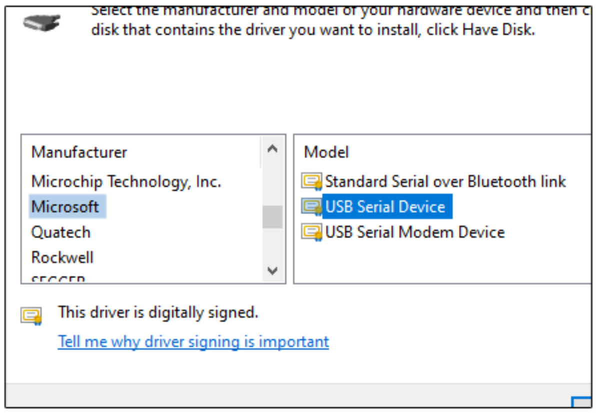


Property: Device instance path

Value: USB\VID_04D8&PID_EFD0&MI_02\8&2D6CB327&1&0002

Right-click on the **Unknown device** and select **Update driver**. In the following dialog select **Browse my computer for drivers**, then **Let me pick from a list of available drivers on my computer**. In the list select **Ports (COM & LPT)**, on the next page select Manufacturer **Microsoft** and Model **USB Serial Device**, finally on the message box select **Yes** to install the driver.





Select the manufacturer and model of your hardware device and then c
disk that contains the driver you want to install, click Have Disk.

Manufacturer	Model
Microchip Technology, Inc.	Standard Serial over Bluetooth link
Microsoft	USB Serial Device
Quatech	USB Serial Modem Device
Rockwell	

This driver is digitally signed.
Tell me why driver signing is important

Now the serial port is available as a Windows **COM** device and can be used with tools such as PuTTY or TeraTerm.

CHAPTER

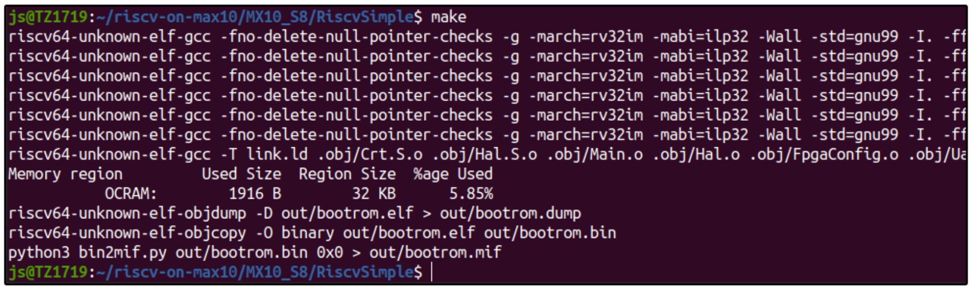
FOUR

PROGRAMMING THE DEMOS

Note: The directory **Prebuild** contains a precompiled firmware image aswell as a prebuild FPGA image. These can be used to skip the corresponding steps in this chapter.

4.1 Compiling the Firmware

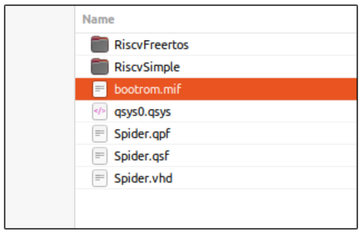
Navigate to the project directory corresponding to the module (Spider_S, MX10_S8, MX10_U). The RISC-V firmware is available as a simple, standalone version (RiscvSimple) or using FreeRTOS (RiscvFreeRTOS). Both versions demonstrate the same functionality, a binary counter on PMod J3 and loopback on UART. Open a terminal and navigate to either firmware and call **make**

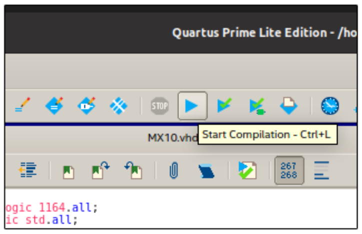


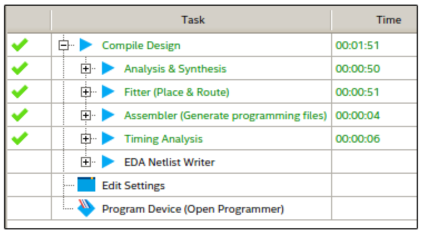
```
js@TZ1719:~/riscv-on-max10/MX10_S8/RiscvSimple$ make
riscv64-unknown-elf-gcc -fno-delete-null-pointer-checks -g -march=rv32im -mabi=ilp32 -Wall -std=gnu99 -I. -ff
riscv64-unknown-elf-gcc -fno-delete-null-pointer-checks -g -march=rv32im -mabi=ilp32 -Wall -std=gnu99 -I. -ff
riscv64-unknown-elf-gcc -fno-delete-null-pointer-checks -g -march=rv32im -mabi=ilp32 -Wall -std=gnu99 -I. -ff
riscv64-unknown-elf-gcc -fno-delete-null-pointer-checks -g -march=rv32im -mabi=ilp32 -Wall -std=gnu99 -I. -ff
riscv64-unknown-elf-gcc -fno-delete-null-pointer-checks -g -march=rv32im -mabi=ilp32 -Wall -std=gnu99 -I. -ff
riscv64-unknown-elf-gcc -fno-delete-null-pointer-checks -g -march=rv32im -mabi=ilp32 -Wall -std=gnu99 -I. -ff
riscv64-unknown-elf-gcc -T link.ld .obj/Crt.S.o .obj/Hal.S.o .obj/Main.o .obj/Hal.o .obj/FpgaConfig.o .obj/Ua
Memory region         Used Size  Region Size  %age Used
           OCRAM:        1916 B        32 KB      5.85%
riscv64-unknown-elf-objdump -D out/bootrom.elf > out/bootrom.dump
riscv64-unknown-elf-objcopy -O binary out/bootrom.elf out/bootrom.bin
python3 bin2mif.py out/bootrom.bin 0x0 > out/bootrom.mif
js@TZ1719:~/riscv-on-max10/MX10_S8/RiscvSimple$ 
```

4.2 Building the Hardware Design

Copy the file **bootrom.mif** from either the previous step or from the precompiled files to the Quartus Project directory corresponding to the module, then start Intel Quartus Prime and open the project. Press **Start Compilation** and Quartus will build the FPGA image and generate the programming files in the subdirectory **output_files/**







	Task	Time
✓	Compile Design	00:01:51
✓	Analysis & Synthesis	00:00:50
✓	Fitter (Place & Route)	00:00:51
✓	Assembler (Generate programming files)	00:00:04
✓	Timing Analysis	00:00:06
	EDA Netlist Writer	
	Edit Settings	
	Program Device (Open Programmer)	

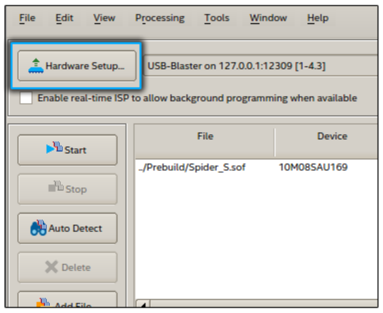
4.3 Programming the FPGA

The MAX10 FPGA supports programming the SRAM cells integrated into the FPGA fabric, this configuration is lost whenever the FPGA is powered off and as such is useful for testing and debugging. The MAX10 FPGA also includes internal non-volatile FLASH to store a configuration image, which on powerup will be loaded into the SRAM. This is usually used for deployment.

The FPGA image can be programmed either with Quartus Programmer using the output file ***.svf** to target the SRAM or ***.pof** to target the FLASH or with OpenOCD using the output file ***.svf** to target the SRAM or ***_pof.svf** to target the FLASH.

4.3.1 Quartus Programmer

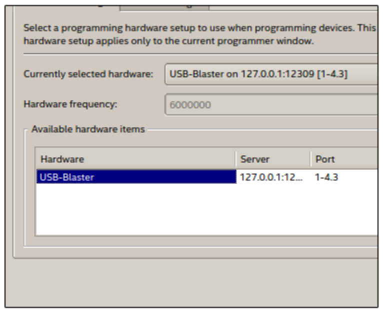
Open the Quartus Programmer and under **Hardware Setup...** select the USB-Blaster. Then select the either the **.sof** or **.pof** file in the project subdirectory **output_files/** or the file corresponding to the module in the directory with the prebuild files and press **Start**.



USB-Blaster on 127.0.0.1:12309 [1-4.3]

Enable real-time ISP to allow background programming when available

File	Device
./Prebuild/Spider_S.sof	10M08SAU169



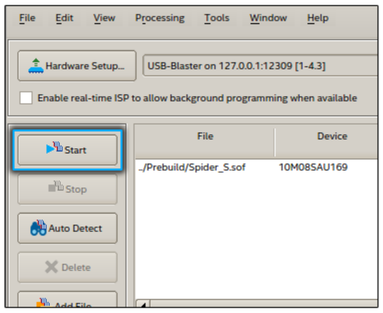
Select a programming hardware setup to use when programming devices. This hardware setup applies only to the current programmer window.

Currently selected hardware: USB-Blaster on 127.0.0.1:12309 [1-4.3]

Hardware frequency: 6000000

Available hardware items

Hardware	Server	Port
USB-Blaster	127.0.0.1:12...	1-4.3



USB-Blaster on 127.0.0.1:12309 [1-4.3]

Enable real-time ISP to allow background programming when available

File	Device
./Prebuild/Spider_S.sof	10M08SAU169

4.3.2 OpenOCD

Open a terminal and navigate to the project subdirectory **output_files/** or the directory with the prebuild files. Run one of the following commands (substitute **<image>** with the name of the file corresponding to the module):

```
# To programm the SRAM
mx10spider <image>.svf

# To programm the FLASH
mx10spider <image>_pof.svf
```

CHAPTER

FIVE

REFERENCE DESIGN

For each of the supported MAX10 SoMs a reference design is included.

The reference design implements:

- **RISC-V 32-bit Core**
  - PicoRV32 RV32IM (SpiderSoM-S)
  - VexRiscv RV32IM (MX10-S)
  - VexRiscv+Cache RV32IMAFC (MX10-U)
- On-Chip Memory (32KB, 64KB MX10-U) initialized with the firmware
- UART via PIC-USB to the host PC
- GPIO Counter on PMod J2

The MX10-U design also implements:

- DDR3 Controller connected to 512 MB RAM.

5.1 FPGA Design

The top-level file for the FPGA is depending on the project **Spider.vhd** or **MX10.vhd**. The top-level file provides the port declaration to interface with the physical pins of the FPGA, it also declares and instantiates the Qsys component. A process sensitive on the system clock (25 MHz) uses a counter to blink the LED on the module once per second. The RISC-V (Qsys) system provides a binary counter on PMod J2 and loopback for the UART interface.

5.1.1 Intel Platform Designer (Qsys)

The Intel Platform Designer implements the RISC-V system. The CPU core and peripheral devices are instatiated, configured and communicate via the Avalon Interconnect. Each device occupies a memory range in the address space, the interconnect will automatically resolve the access signals.

5.1.1.1 SERV

The Serv Core implements the RV32I instruction set, an integrated memory mapped interrupt controller provides handling for external, software and timer interrupts. The interrupt controller also provides a general purpose time register. The following configuration parameters are avilable:

Parameter	**Description**
Reset Vector	Address loaded into the program counter when the core starts.
Interrupts	Number of interrupts avilable in the interrupt controller. (Range 1 - 32)
Timer Width	Number of bits implemented for the timer counter. (Range 33 - 64)

5.1.1.2 PicoRV32

The PicoRV32 core can be configured as RV32E, RV32I, RV32IC, RV32IM, or RV32IMC core and implements a native custom interrupt controller. The following configuration parameters are avilable.

Parameter	**Description**
Enable Counters	Enables support for `RDCYCLE[H]`, `RDTIME[H]` and `RDINSTRET[H]` instructions.
Enable Counters (64bit)	Enables support for `RDCYCLEH`, `RDTIMEH` and `RDINSTRETH` instructions.
Enable Registers x16 to x31	Enables support for registers `x16` to `x31`. When disabled the core uses the RV32E instruction set.
Dual Port Registers	Increases performance for register access, but may increase the size of the core.
Two Stage Shift	Speeds up the shift operation, but increases the size of core slightly.
Barrel Shifter	Implements the shift operation by using a barrel shift, which is faster, but further increases the size of the core.
Two Cycle Compare	Relaxes the longest data path and improves timing, but adds an additional clock cycle for branch instructions.
Two Cycle ALU	Improves timing, but adds an additional clock cycle for instructions that use the ALU.
Compressed ISA	Enables support for the compressed (C) instruction set.
Catch Address Misalign	Enables circuitry for catching misaligned memory accesses.
Catch Illegal Instruction	Enables circuitry for catching illegal instructions.
Enable MUL	Enables support for the `MUL[H[SU|U]]` instructions.
Enable Fast MUL	Increases performance for multiplication, but increases the size of the core.
Enable DIV	Enables support for the `DIV[U]/REM[U]` instructions.

Parameter	Description
Enable Interrupts	Enables the internal interrupt controller.
Masked IRQ	A `1` bit in this bitmask permanently disables the corresponding interrupt.
Latched IRQ	A `1` bit in this bitmask latches the interrupt signal (edge-triggered) instead of operating on level sensitive interrupts.
Reset Vector	Address loaded into the program counter when the core starts.
Interrupt Vector	Address loaded into the program counter when an interrupt or execution error occurs.

5.1.1.3 VexRiscv

The VexRiscv core can be configured as RV32I, RV32IM without caches or as RV32IM, RV32IMAC, RV32IMAFC with 4KB instruction and data caches.

Parameter	Description
Reset Vector	Address loaded into the program counter when the core starts.
Exception Vector	Address loaded into the program counter when an exception (interrupt or trap) occurs.
IO Region Begin	First (inclusive) address of the uncached region; volatile memory such as registers of external modules are required to be in this region. Does not have an effect if no caches are used.
IO Region End	Last (inclusive) address of the uncached region. Does not have an effect if no caches are used.
Core Configuration	Specifies the implemented instruction set and caches.

5.2 C-Firmware

The C firmware by default will output a binary counter to the GPIO and loopback every character received on Uart. The internal counter (or in case of the Serv the counter of the interrupt controller) will be read to increment or decrement the binary counter value every 32th of a second. The timer will also be configured to provide an interrupt every 2 seconds. The corresponding interrupt handler changes the direction of the binary counter. An additional interrupt handler triggered on Uart receive will loop back the characters received.

5.2.1 Common Files

File	Description
Hal.c, Hal.h, Hal.S	**Hardware Abstraction Layer**, provides interface to the core hardware such as timers and interrupts
Crt.S	**C Runtime**, start-up code that initializes the core and invokes `main`
FpgaConfig.h, Fpga-Config.c	Configuration file that describes the Qsys RISC-V system
Main.c	Entry point for the C firmware
Makefile	Build file for `make`
RiscvDef.h	Definitions for RISC-V constants
Uart.c, Uart.h	Software description and driver file for Uart
bin2mif.py	Python script, converts the binary output to a memory initalization file
link.ld	Linker script, instructs the linker on how to assemble the binary.

5.2.2 FreeRTOS

For FreeRTOS the RISC-V specific files have been moved to the subdirectory **FreeRTOS/portable**. Timer and software interrupts are dedicated to FreeRTOS to provide context switching. To modify the reference to the current task, two additional functions (`void StoreStackPointerInCurrentTCB(uintptr_t stack)` and `uintptr_t LoadStackPointerFromCurrentTCB()`) were added in task.h and tasks.c.

5.3 Modifying the Examples

The reference designs provide an ideal starting point for the development of a RISC-V enabled FPGA project. The following sections show how to modify the FPGA design and the C firmware.

5.3.1 Adding a lightweight printf-library

An open source implementation of a printf library is available at https://github.com/eyalroz/printf. Copy the files **printf.c** and **printf.h** from the **src** subdirectory to the RISC-V firmware directory. Then create a new file called **printf_impl.c** with the following code inside:

```c
#include "printf.h"
#include "FpgaConfig.h"

void _putchar(char character) {
        if (character == '\n'){
                UartPut(g_Uart, '\r');
        }
        UartPut(g_Uart, character);
}
```

The library can now be used in any file by including the header:

```c
#include "printf.h"
```

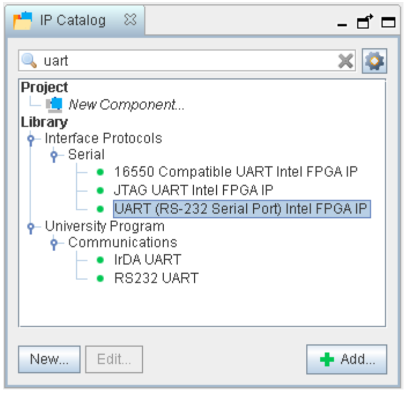
Replace the beginning of the **main**-function with the following code snippet for demonstrative purpose:

```c
int main() {

// Greetings
printf_("\n\n* * * Printf Demo - %s * * *\n", DBUILD_DATE);

// Set GPIO to output.
g_Pio->direction = 0xffffffff;
```

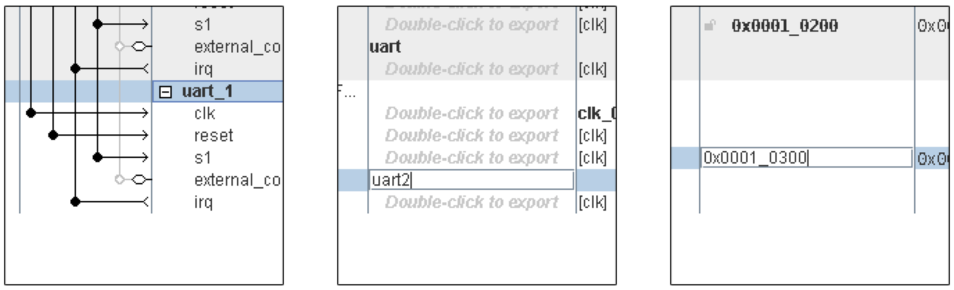
5.3.2 Adding a second Uart to Qsys

Note: This section uses the Spider_S project, there may be small differences between projects that have to be transfered.

Open the Qsys system in Intel Platform Designer. Search in the IP Catalog for "uart", select the `UART (RS-232 Serial Port) Intel FPGA IP` and press **Add**.



The default settings are suitable for the Uart core. Connect the signals to the interconnect, for the **SERV** and the **VexRiscv** core make sure to connect the `s1` signal to the `data bus`. Select an unoccupied memory range for the Memory Mapped, this example sets the base address to 0x00010300. Export the Conduit signal to the top-level file.



The new Uart also has to be added in the top-level file, in order to do so select the menu **Generate** in the Platform Designer and then **Show Instantiation Template**. In the dropdown menu HDL Language select **VHDL**. The two new signals are `uart2_rxd` and `uart2_txd`. Open the top-level file (**MX10.vhd** or **Spider.vhd**) in Quartus and modify the component declaration:

```vhdl
component qsys0 is
        port (
                clk_clk         : in    std_logic;
                reset_reset_n   : in    std_logic;
                gpio_export     : inout std_logic_vector(31 downto 0) := (others => 'X');
                uart_rxd        : in    std_logic;
                uart_txd        : out   std_logic;
                uart2_rxd       : in    std_logic;
                uart2_txd       : out   std_logic
        );
end component qsys0;
```

Modify the component instantiation, this example routes the Uart signals to PMod:

```
u0 : component qsys0
        port map (
                clk_clk        => clk25,
                reset_reset_n  => resetn,
                gpio_export    => gpio,
                uart_rxd       => uart_rx,
                uart_txd       => uart_tx,
                uart2_rxd      => pmod_j3(1),
                uart2_txd      => pmod_j3(0)
        );
```

To use the Uart from the RISC-V firmware open the file **FpgaConfig.h**

Add the memory address and interrupt number as specified in Intel Platform Designer:

```
#define MEMADDR_UART2  ((uintptr_t)(0x00010300))
#define IRQ_UART2      1
```

Declare the Uart struct:

```
extern Uart* g_Uart2;
```

Open the file **FpgaConfig.c** and provide the definition for the struct:

```
Uart* g_Uart2 = (Uart*)(MEMADDR_UART2);
```

Now the second Uart is available to be used in the firmware:

```
int main() {

// Greetings
UartWrite(g_Uart, "\n\n* * Example Demo - "DBUILD_DATE" * *\n");
UartWrite(g_Uart2, "Hello World\n");
```