# FMC masterFIP

## design guide

M. Arruat | E.Gousiou | June 2017

# Contents

# 1. Introduction

This document describes the implementation of the WorldFIP master, masterFIP. according to the functional specifications described in [1]. The project is hosted in the Open Hardware Repository [2] and is composed of:

- Hardware: dedicated FMC mezzanine board.
- Gateware: HDL code for the configuration of the SPEC FPGA.
- Software:
    - Real-time Software: bare-metal C code running in the SPEC FPGA embedded CPUs.
    - Software Libraries: C libraries for the development of user-space applications.
    - Note that there are no dedicated drivers for this project; as it is described in Section 5, the generic Mock Turtle driver is used.

The masterFIP is for CERN applications a drop-in replacement of the Alstom master. It was designed mainly during 2015-2016 with the aim of replacing all the Alstom masters at CERN (around 500 units) in 2019. The masterFIP does not implement all the WorldFIP protocol features, only those needed by CERN applications. Consult the functional specifications [1] for a detailed description of the implemented features and slight declinations from the protocol for aperiodic messaging and aperiodic SMMPS traffic.

# 2. Hardware

The FMC-masterFIP board [3] is an interface card for the WorldFIP network in an LPC FMC form-factor. The Simple PCI Express Carrier, SPEC [4], is used as carrier for the FMC-masterFIP. It provides FPGA resources with a Xilinx Spartan6 (XC6SLX45T- 3FGG484C), 4-lane PCIe interface through the Gennum GN4124 chip, several clocking resources, volatile and non-volatile memory, an on board thermometer and unique-64-bit-identifier chip, the FMC Low-Pin-Count connector, programmable red and green LEDs, a JTAG connector and a FMC front panel.

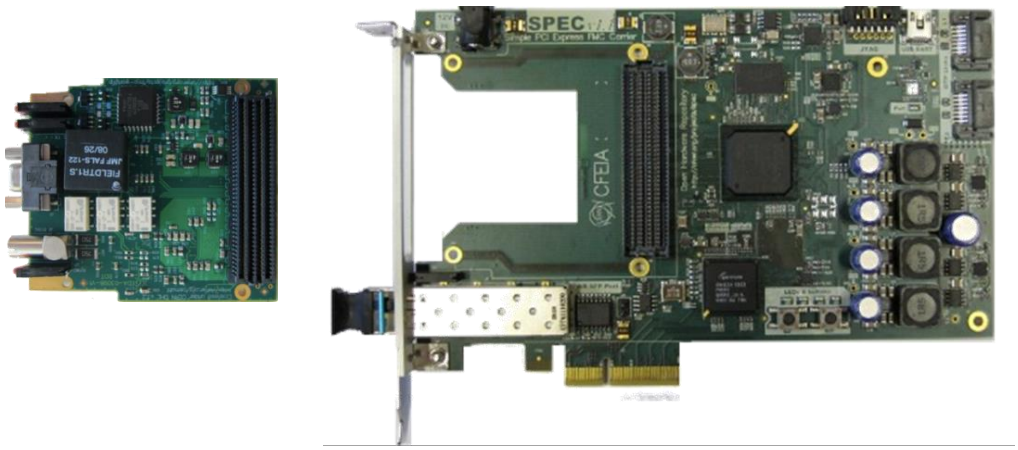


Figure 1: FMC masterFIP mezzanine and SPEC carrier boards

The main components of the FMC-masterFIP board are: the FielDrive bus driver [5] and FieldTR transformer [6], both developed and sold by the company Alstom. A Lemo-00 connector on the board is used for the reception of an input synchronization trigger pulse, usually from the timing board CTRI [7]. To comply with the FMC standard, the board has an EEPROM chip, loaded with IPMI FRU information, and a 1-wire thermometer-unique-id-chip.

The schematics of the board, drawn in Altium, are available in EDMS at [8]. As the following table shows, three versions of the board have been designed. A list of issues that were being identified on each version is available in [9]. Different board executions exist depending on the WorldFIP communication speed: 31.25kbps, 1Mbps, 2.5Mbps, and 5Mbps. A set of ten components differentiates the board versions.

| Version | Schematics | Comments |
|---|---|---|
| V1 | EDA-03098-V1-0, EDA-03098-V1-1, EDA-03098-V1-2, EDA-03098-V1-3 | First version |
| V2 | EDA-03098-V2-0, EDA-03098-V2-1, EDA-03098-V2-2, EDA-03098-V2-3 | Corrections on V1 |
| V3 | EDA-03098-V3-0, EDA-03098-V3-1, EDA-03098-V3-2, EDA-03098-V3-3 | Simplified, without ADC |

Table 1: FMC-masterFIP hardware versions

A dedicated Production Test Suite [10] has been developed for the FMC-masterFIP. It has been developed by the company Creotech, in collaboration with CERN. The FMC-masterFIP PTS is an extension of the original PTS which allows to perform functionality tests on FMC-masterFIP boards after manufacturing. It assures that the boards comply with a minimum set of quality rules, in terms of soldering, mounting and fabrication process of the Printed Circuit Boards (PCB). It also loads the FMC-masterFIP EEPROM with IPMI FRU information. It is important to note that the FMC masterFIP PTS covers only the functionality tests of the boards and does not cover any verification or validation tests of the design.

# 3. Gateware architecture

Following the evaluation of different implementation solutions [11], the technical choice for the design was the use of Mock Turtle [12]. Mock Turtle (also referred to as White Rabbit Node Core) is an HDL core of a generic distributed control system node, based on multiple deterministic CPU cores where the users can run any sort of hard real-time applications. The applications can be written in bare metal C, using standard GNU tool set, cross-compiled and loaded into the CPUs. The CPUs can communicate between each other through a dedicated Shared Memory (SM) and with the host through Host Message Queues (HMQ). Mock Turtle also offers Remote Message Queues (RMQ) for communication with a White Rabbit Network; RMQs are not used in this project.

Figure 2 illustrates the SPEC Spartan6 FPGA gateware architecture, spec_masterfip_mt.

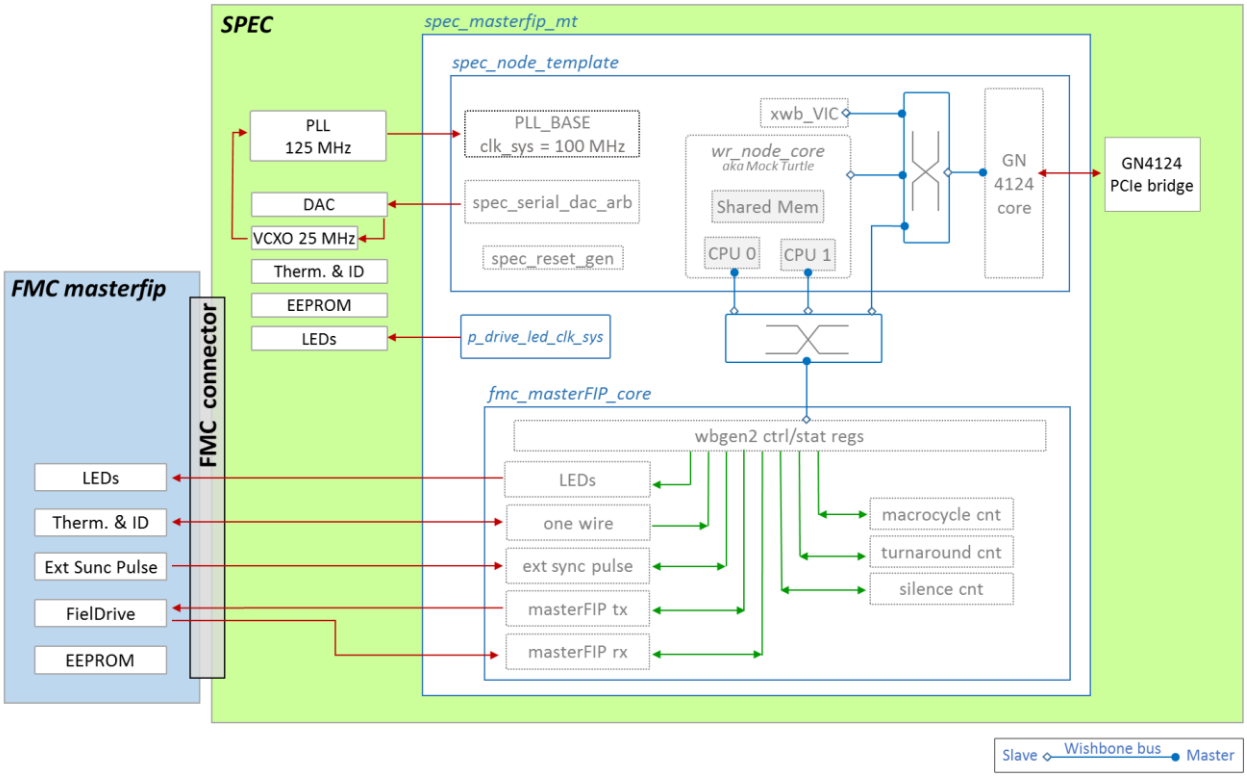


Figure 2: SPEC FMC masterfip gateware architecture

The top entity consists of two main blocks: the spec_node_template, which is the Mock Turtle top level for SPEC, and the application-specific fmc-masterFIP_core. The communication between the two modules is through WISHBONE.

# 3.1 Clocks and Resets

There is one clock domain of 100 MHz in the design. As the following figure shows the 100 MHz clock is generated in the spec_node_template from the 125 MHz clock of the SPEC PLL IC6. All the units of the design (e.g. CPU0, CPU1, fmc_masterfip_core, crossbars) are running on the the 100 MHz clock. The spec_node_template also provides a reset, coming from the PCIe host, synchronised to the 100 MHz clock.

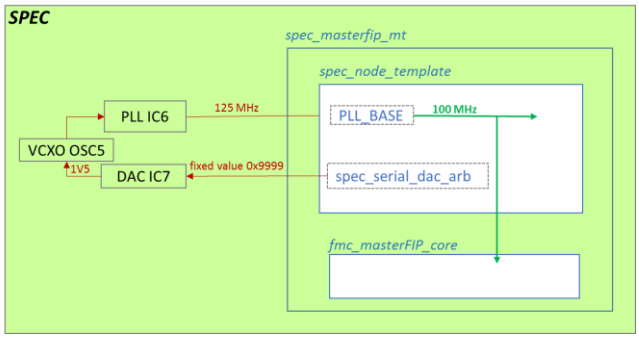


Figure 3: Clock generation

# 3.2 Mock Turtle

The spec_node_template is a highly configurable module designed to ease the development of Mock Turtle applications using the SPEC card as platform. It is basically a wrapper containing Mock Turtle as well as typical generic cores required to work with a SPEC carrier: GN4124 core, Vectored Interrupt Controller (VIC) and SPEC reset generator.

# Mock Turtle configuration

In this design Mock Turtle is configured as follows:

| Parameter | Value |
|---|---|
| CPUs | CPU0, CPU1: running at 100 MHz |
| White Rabbit Support | No |
| Remote Message Queues | 0 |
| CPU0 memory size | 98304 bytes |
| CPU1 memory size | 8192 bytes |
| Shared Memory size | 65536 bytes |
| Host Message Queues from MT -> host | 8 |
| Host Message Queues from host -> MT | 2 |

Table 2: Mock Turtle configuration

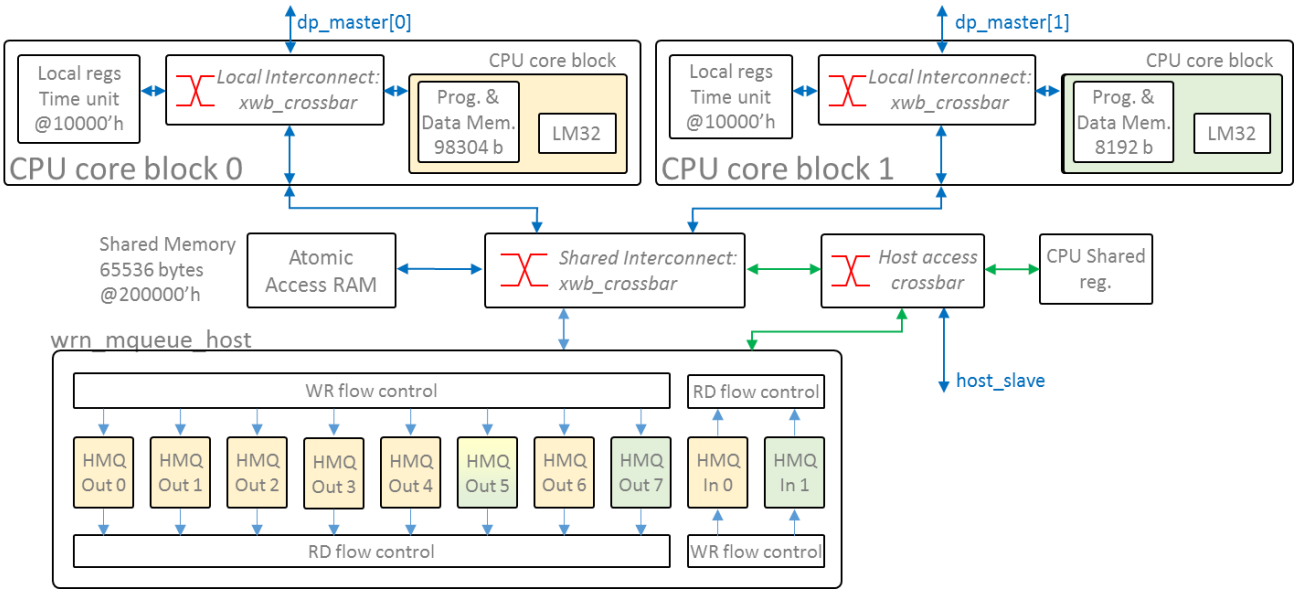


Figure 4: Mock Turtle main modules

| READ HMQ | Description |
|---|---|
| HMQ0 | Host -> CPU0 for<br>- commands for the bus configuration used only at startup: MSTRFIP_CMD_SET_HW_CFG, MSTRFIP_CMD_SET_VAR_LIST, MSTRFIP_CMD_SET_BA_CYCLE, MSTRFIP_CMD_START_BA |
| HMQ1 | Host -> CPU1 for<br>- payloads for produced WorldFIP frames, variables and messages. CPU1 then puts this data into the Shared Memory for CPU0 to access and put them in WorldFIP.<br>- requests for report data, requests for the scheduling of aperiodic traffic, presence/ identif/messages etc; CPU1 again passes these requests into the Shared Memory.<br>- stop/reset commands MSTRFIP_CMD_STOP_BA, MSTRFIP_CMD_RESET_BA that are passed to the Shared Memory and are checked by CPU0 at the end of each macrocycle window. |

Table 3: Read HMQs

| WRITE HMQ | Description |
|---|---|
| HMQ0 | CPU0 -> Host for the WorldFIP payloads from periodic consumed variables |
| HMQ1 | CPU0 -> Host for the WorldFIP payloads from aperiodic consumed variables<br>(only for the case of identif variable, scheduled as periodic variable, by radMon app) |
| HMQ2 | CPU0 -> Host forthe WorldFIP payloads from aperiodic consumed messages |
| HMQ3 | CPU0 -> Host for the WorldFIP payloads from periodic consumed diagnostic variables<br>(only for the case of the FIPdiag variable 0x067F) |
| HMQ4 | CPU0 -> Host for the WorldFIP payloads from aperiodic consumed diagnostic variables<br>(aperiodic presence and identification) |
| HMQ5 | CPU0 & CPU1 -> Host for debugging |
| HMQ6 | CPU0 -> Host for the responses of the commands of the host from READ HMQ0<br>(acknowledgement of the configuration; load, start, stop, reset macrocycle) |
| HMQ7 | CPU1 -> Host for the responses to the commands of the host from READ HMQ1<br>(e.g.: content of the report variable) |

Table 4: Write HMQs

CPU0 is the heart of the design; its purpose it to “play" in a deterministic way the WorldFIP macrocycle. For example, it initiates the delivery of a WorldFIP question frame, by providing the frame bytes to the fmc_masterfip_core, and then awaits for the reception of the response frame. It retrieves the consumed data from the fmc_masterfip_core, packs them in the corresponding HMQ (according to the frame type) and can notify the host through an IRQ.

The only interaction between the host and CPU0 is for the macrocycle configuration; upon the application configuration, which in principle takes place only once at startup, CPU0 is receiving through a dedicated HMQ (RD HMQ0) the macrocycle configuration (for example: the number and size of produced/ consumed variables, the lengths of periodic/aperiodic windows etc). At the end of every window (periodic/aperiodic window of a macrocycle) CPU0 is checking the Shared Memory for a RESET/ STOP request.

The following figure shows the states of the CPU0 state machine; during WorldFIP operation the state machine is at “running” state.

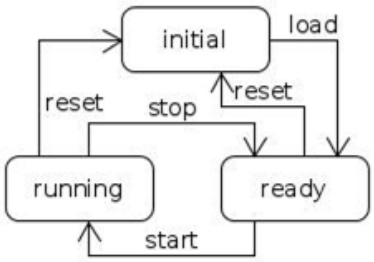


Figure 5: CPU0 state machine

CPU1 is mainly polling the host to retrieve new payload bytes for production. When new data is received from the host through a dedicated HMQ, CPU1 puts them into the Shared Memory for CPU0 to retrieve them and provide them to the FMC MASTERFIP CORE for serialization. CPU1 does not need access to the FMC MASTERFIP CORE; however, access is possible for debugging purposes. CPU1 is also polling the host for a masterFIP stop or reset command and passes this information to the Shared Memory to make it available to CPU0.

## GN4124

The GN4124 chip [13] is a four lane PCI Express Generation 1.1 bridge. In addition to the PHY, it contains the data link and transaction layers. The GN4124 chip is used to access the FPGA registers, but also to generate MSI interrupts and re-program the FPGA. The BAR 4 (Base Address Register in the PCIe memory space) allows access to the GN4124 internal registers. The BAR 0 is connected to the local bus allowing access to the FPGA. The GN4124 core is the interface to the Gennum 4124 PCIe bridge chip. Internally it provides two separate WISHBONE buses: the Control and Status Register (CSR) standard WISHBONE interface and the Direct Memory Access (DMA) pipelined WISHBONE interface. Mock Turtle does not make use of DMA.

## VIC

The vectored interrupt controller (VIC) gathers all the interrupts into one single interrupt request line. On the SPEC carrier, the VIC interrupt request output is connected to GPIO 8 of the GN4124 chip. Therefore, the GN4124 must be configured to generate an MSI when a rising edge is detected on GPIO 8. In this design an interrupt is generated each time a message is pushed in any of the HMQs.

## SPEC serial DAC arbiter

As there is no White Rabbit support in the design, the SPEC oscillator is disciplined through a fixed value on the DAC. This spec_serial_dac_arbiter sends the default fixed value “0x9999” to the DAC; this value centers the OSC5 VCXO at 1V5, which gives the best oscillator stability.

# 3.3 FMC masterFIP core

On one side (red arrows in the figure below) the fmc_masterfip_core is the interface to the FMC hardware i.e. FielDrive chip, external pulse LEMO, 1-wire DS18B20 chip, LEDs. On the other side (blue line in the figure below) it provides a wbgen2 WISHBONE where a set of control and status registers have been defined to interface with the MOCK TURTLE.

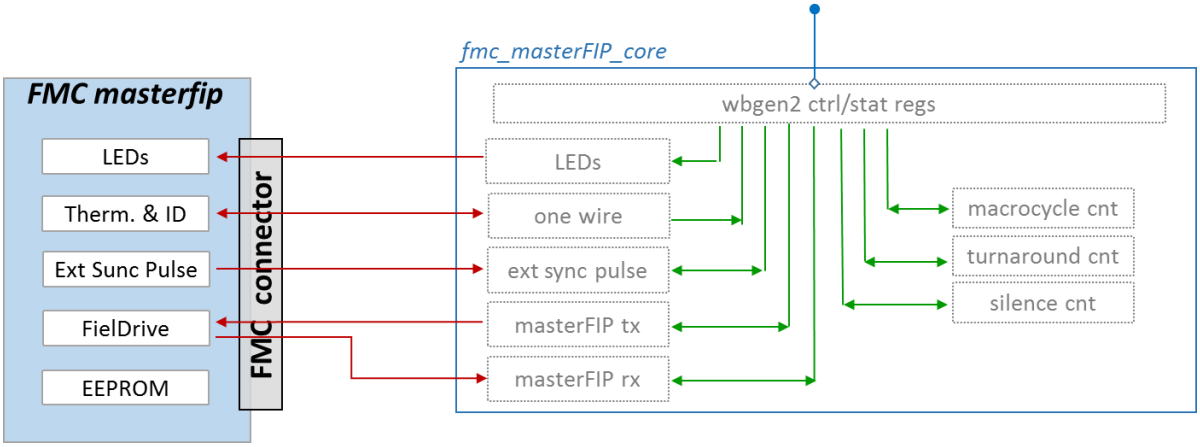


Figure 6: fmc_masterfip_core main units

The core ignores the notion of the WorldFIP frame type (ID_DAT/RT_DAT/..etc), or the macrocycle sequence and macrocycle timing; the sw running on the MOCK TURTLE CPUs is responsible for managing these aspects and for providing to this core all the payload bytes (coming from the host) that have to be serializedand or for enabling the deserializer and then providing to the host the deserialized bytes.

## Wbgen2 CSR registers

The masterfip_wbgen2_csr module has been generated through the wbgen2 application. It establishes the interface with the Mock Turtle core. This interface contains a set of control (from the MT) and status (from the fmc_masterfip_core) registers for each one of the units of Figure 6; it also contains the WorldFIP frame payload data for the TX and from the RX.

The wbgen2 html in [14] lists each register, its location and use in detail.

The control signals are usually defined as “monostables”, that is 1-clk-tick long pulses. The status signals are set by the fmc_masterfip_core and cleared by the Mock Turtle, usually through a dedicated reset.

Regarding the payload bytes’ exchange between the Mock Turtle and the fmc_masterfip_core, it was decided not to use a FIFO; instead a set of 67 32-bit-registers (= 268 bytes, which is the max WorldFIP frame size) is used for each of the TX and RX; like this the time for which the data need to remain static to be extracted is minimized (the copy of all the payload bytes requires one clock cycle), leading to a simpler design.

## MasterFIP TX

The masterfip_tx places a complete WorldFIP frame on the WorldFIP bus. It ignores the frame type (ID_DAT/RT_DAT/RP_MSG etc..), or the macrocycle sequence and macrocycle timing; the software running on Mock Turtle is responsible for managing all these aspects and for providing to the masterfip_tx the bytes to serialize, along with a start pulse. As explained in the section above, the communication between Mock Turtle and the masterfip_tx is handled through a set of control (from the MT) and status (from the masterfip_tx) signals/registers defined in the wbgen2 csr module.

The following figure shows the main functionality of the unit. The modules tx_osc and tx_serializer come unmodified from the nanoFIP design.

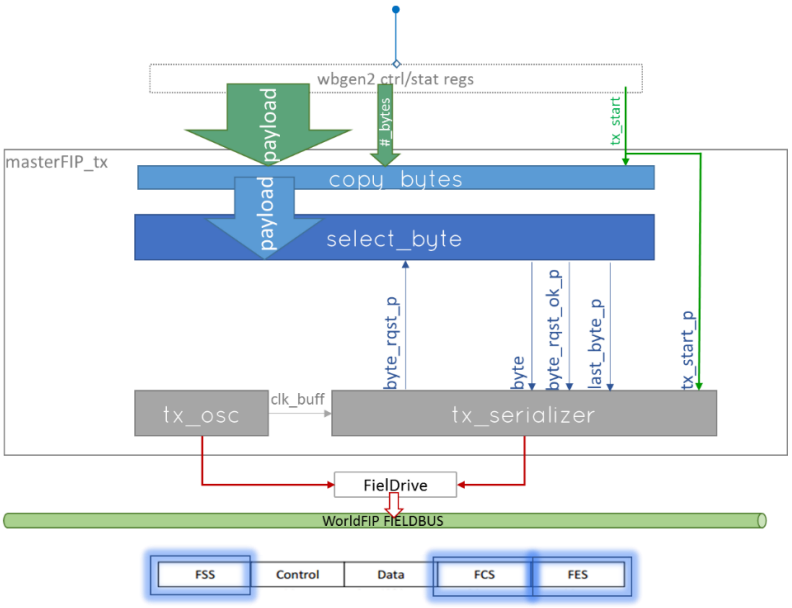


Figure 7: masterfip_tx main units

Upon a rising edge on the tx_ctrl_start pulse (which is defined as monostable in the wbgen2), the masterfip_tx:

- copies all the payload registers (tx_payld_ctrl, tx_payld_reg1..tx_payld_reg67) and the register that indicates the number of payload bytes to serialize (tx_ctrl_bytes_num)

- starts serializing a WorldFIP frame. Note that the FSS, CRC and FES fields are generated internally in the masterfip_tx unit. All the rest of the bytes are coming from Mock Turtle trough the wbgen2 payload registers.

- after the FES, rises the tx_stat_stop status bit to signal to the MT the end of a successful frame transmission.

## MasterFIP RX

The masterfip_rx retrieves a WorldFIP frame from the WorldFIP bus. The following figure shows the main functionality of the unit. The modules rx_osc, rx_deserializer and rx_deglitcher come

unmodified from the nanoFIP project. Similar to the masterfip_tx, the masterfip_rx has no intelligence regarding the macrocycle sequence; it is controlled and monitored by the Mock Turtle through the wbgen2 control and status signals. As long as it is not under reset, the masterfip_rx is probing the WorldFIP bus trying to identify the FSS sequence of a WorldFIP frame. It signals the FSS detection to the processor through the status bit fss_detect and continues deserializing the rest of the frame. It stores the first byte after the FSS to the rx_payld_ctrl register and the rest of the bytes to the registers rx_payld_reg1..rx_payld_reg67.

Upon the detection of a FES the masterfip_rx checks the CRC of the frame and enables the status bit rx_stat_frame_ok or rx_stat_crc_err accordingly. Upon the rx_stat_frame_ok, the status register rx_stat_bytes_num indicates the number of bytes in the frame (this indicates the number of rx_payld_regs and the number of bytes inside the last rx_payld_reg to be retrieved by the processor).

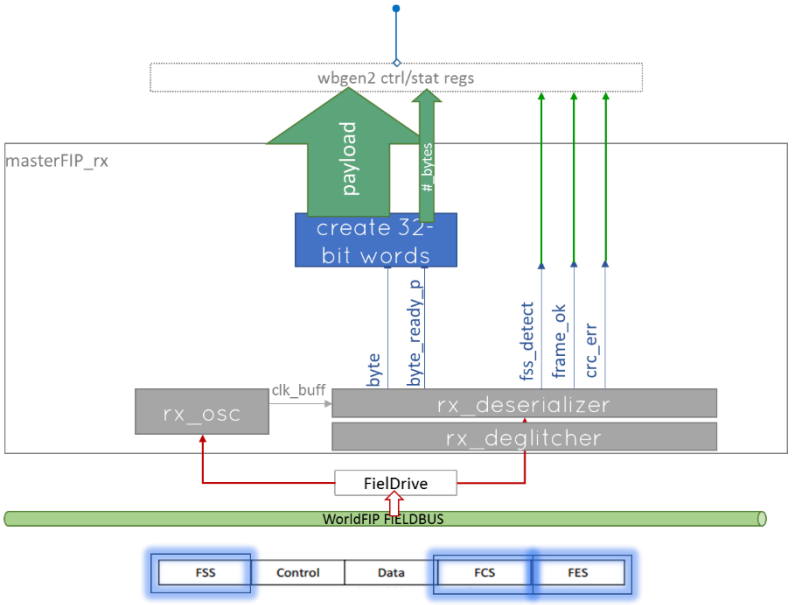


Figure 8: masterfip_rx main units

The processor should copy the rx_payld_regs upon a frame_ok; the regs keep their values until a rx_rst or until the detection of a new frame_ok. This time, in the worst case (bit rate 2.5 Mbps) can be calculated as: (Min Turnaround time of a node = 4 us) + (RP_FIN duration = 19.2 us) = 23.2 us.

Note that the deserializer is under reset when the serializer is working, that is when the fd_txena is enabled; this happens in the gateware.

## Ext Sync Pulse

The modules related to the ext_sync_pulse, synchronize, deglitch and count the number of rising-edge pulses that arrive to the ext_sync input of the board; they provide the result to a dedicated wbgen2 CSR register.

Note that the FMC-masterFIP hardware allows the LEMO connector to be used as input through the transceiver IC1; on the gateware however the transceiver is configured constantly with the direction as input.

## Macrocycle cnt

The modules related to the macrocycle:

- count the time of a macrocycle using the 100 MHz clock
- count the number of macrocycles since startup/a reset.

Dedicated registers in the wbgen2 CSR provide the counters values to Mock Turtle. Note that the duration/length of the macrocycle comes from the processor through another dedicated register in the wbgen2 CSR that should be set once in the application startup.

## Silence/ Turnaround cnt

The modules related to the turnaround and silence time, count the respective time using the 100 MHz clock. Dedicated regs in the wbgen2 CSR provide the counters values to Mock Turtle. As in the case of the macrocycle length, the turnaround and silence time length is provided through other dedicated regs in the wbgen2 CSR that should be set once in the application startup.

## One-Wire

The 1-wire unit is reading the unique ID and temperature on the mezzanine. Differently than in other designs that implement sw-bit-banging, here the communication is hard-coded in VHDL, so as to simplify the drivers.

## 3.4 LEDs

There are 6 LEDs on the FMC-masterFIP mezzanine. They are all driven by the Mock Turtle real-time software; they are updated at the end of each macrocycle.

| FMC-masterFIP LED | Function |
|---|---|
| Green TX active | ON when at the end of a macrocycle there has been no transmission failure |
| Red TX error | ON when at the end of a macrocycle transmission errors have been detected |
| Green RX active | ON when at the end of a macrocycle there has been no reception failure |
| Red RX error | ON when at the end of a macrocycle transmission errors have been detected |
| Green EXT SYNC | ON when an application expects EXT PULSE and it is receiving it successfully. OFF when an application is not using the EXT_SYNC. |
| Red EXT SYNC error | ON when an application expects EXT PULSE which it is not arriving. |

Table 5: FMC-masterFIP front panel LEDs

There are 2 LEDs on the SPEC front panel; they are driven by the gateware.

| SPEC LED | Function |
|---|---|
| green | Blinking using the 100 MHz clock |
| red | ON upon a PCIe reset |

Table 6: SPEC front panel LEDs

## 3.5 Design Register Map

The following list shows the register map of the different parts, from the PCIe host point of view:

- 0x00000000 (size: 4 bytes) : SDB signature
- 0x00002000 (size: 64 bytes) : VIC
- 0x00010000 (size: 644 bytes) : Host access to the fmc_masterfip_core
- 0x00020000 (size: 128 kB) : Mock Turtle
    - 0x00020000 : HMQ Global Control Registers
    - 0x00024000 : HMQ incoming slots (Host->CPUs)
    - 0x00028000 : HMQ outgoing slots (CPUs->Host)
    - 0x0002c000 : CPU Control/Status Registers
    - 0x00030000 : Shared Memory (64 kB)

The following list shows the register map of the different parts, from the CPUs point of view:

- 0x00010000 : Host access to the fmc_masterfip_core
- 0x00020000 : Mock Turtle

The fmc_masterfip_core map is available at the wbgen2 html file in [15].

## 3.6 Xilinx XC6SLX45T resources

| XC6SLX45T | | | |
|---|---|---|---|
| Occupied slices | 5255 | 6822 | 77% |
| Slice LUTs | 15586 | 27288 | 57% |
| Slice Registers | 12622 | 54576 | 23% |
| MUXCYs | 1452 | 13644 | 10% |
| IOBs | 96 | 296 | 32% |
| RAMB16BWER | 88 | 116 | 75% |
| RAMB8BWER | 22 | 232 | 9% |
| BUFIO2 | 1 | 32 | 3% |
| BUFIO2FB | 1 | 32 | 3% |
| BUFG/BUFGMUX | 3 | 16 | 31% |
| ILOGIC2/ISERDES2 | 29 | 376 | 7% |
| IODELAY2/IODRP2/IODRP2_MCB | 2 | 376 | 1% |
| OLOGIC2/OSERDES2 | 21 | 376 | 5% |
| BUFPLL | 1 | 8 | 12% |
| BUFPLL_MCB | 0 | 4 | 12% |
| DSP48A1 | 6 | 58 | 10% |
| PLL_ADV | 2 | 4 | 50% |

Table 7: spec_masterfip_mt design resources

# 4. Source Code

There are different repositories for the project's components.

## 4.1 Gateware code

Here is the procedure to build the FPGA binary image from the hdl source.

1. Clone git repo:
   $ git clone git://ohwr.org/cern-fip/masterfip/masterfip-gw.git

2. Checkout 'eva_dev' branch (TODO: marge to master branch):
   $ cd masterfip-gw
   $ git checkout –t origin/eva_dev

3. Submodules initialization and update (not recursive!):
   $ git submodule init
   $ git submodule update

4. To synthesize, open the 'syn/spec/spec_masterfip_mt.xise' with Xilinx ISE®:
   $ cd syn/spec
   $ ise spec_masterfip_mt.xise

5. To simulate, run
   $ cd syn/spec
   $ ise spec_masterfip_mt.xise

## Gateware Source Code Organization

*masterfip-gw /rtl:* specific hdl sources of the fmc-masterfip core.

*masterfip-gw/rtl/wbgen:* wbgen2 descriptor file describing all the registers used by the fmc-masterfip-core. It contains also the documentation of those registers in html format, the C header file with useful register offsets and mask macros and a '.sh' script allowing to automatically generate the wishbone slave hdl sources accessing those registers.

masterfip-gw/ip_cores: essential submodules

- gn4124-core: repo: git://ohwr.org/hdl-core-lib/gn4124-core.gi
  commit : e3a0bf97e125020c83bff6e40199a717e7fda738 (proposed_master, 31.05.16)
- general-cores: repo: git://ohwr.org/hdl-core-lib/general-cores.git
  commit:1c2dd12b1bceeab3b32b41c3522931c658ad15a7 (tom-proposed-master-feb28, 28.02.17)
- nanofip: repo: git://ohwr.org/cern-fip/nanofip/nanofip-gateware.git
  commit: 752512a82a05ce5ac4c69ad19f68921762bdd512 (master, 24.03.15)
- wr-node-core: repo: git://ohwr.org/white-rabbit/wr-node-core.git
  commit: 96a78592b4d20140bf13662476605ab7e96d7710 (proposed_master, 10.02.17)

Despite the fact that there is no White Rabbit support for the project, the wr-node-core requires certain packages that are defined in the wr-cores and etherbone-cores; therefore, these submodules are also loaded.

- wr-cores: repo: git://ohwr.org/hdl-core-lib/wr-cores.git
  commit: d0d4d09d5f0355dfc6c078171bc6856e580a7496 (tom-wr-node, 19.12.2016)
- etherbone-cores: repo: url = git://ohwr.org/hdl-core-lib/etherbone-core.git
  commit : c1e676dc9d35028910c50431d70328e522396c89 (master, 06.02.15)

*masterfip-gw/sim/spec:* location of the carrier related simulation files.

*masterfip-gw/syn/spec:* Xilinx ISE project for synthesis

*masterfip-gw/top/spec:* hdl of the top entity for the specified carrier; also contains the ISE constraints and sbd synthesis_descriptor files.

# 4.2 Software code

The software code is housed in gitlab.

1. Clone git repo:
   $ git clone https://gitlab.cern.ch/cohtdrivers/masterfip

2. Submodules initialization and update:
   $ git submodule init
   $ git submodule update

## Software Source Code Organization

***masterfip/rt/ba:*** real time software running on CPU0

***masterfip/rt/cmd:*** real time software running on CPU1

***masterfip/rt/common:*** shared memory allocation functions; common definitions for CPU0 and CPU1

**masterfip/rt/common/hw:** wbgen2-generated header file with registers definitions

**masterfip/include**: shared lib, user space application common definitions

**masterfip/mockturtle:** Mock Turtle as a submodule

**masterfip/lib:** library for building applications with masterFIP

**masterfip/tools/testbed**: example application

# 5. Software stack

As described in Section 3 Mock Turtle is a framework for hard real-time applications. Developing a Mock Turtle application consists in providing a specific library and real time applications running on the Mock Turtle processing cores. The picture below depicts the current software stack for the masterFIP Mock Turtle application.

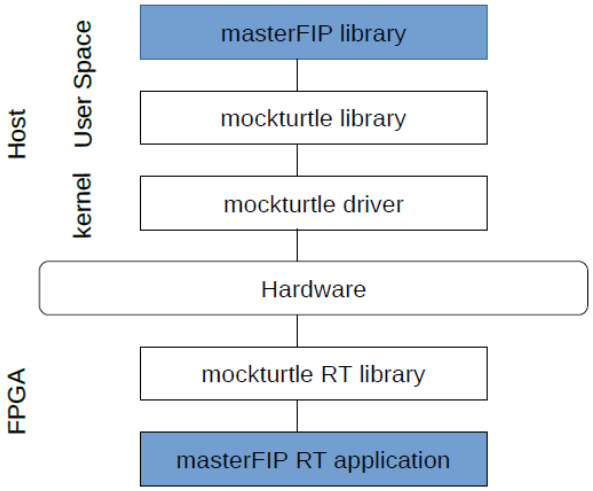


Figure 9: masterFIP software stack

# 6. Library API

## I. *Open/close device*

**int mstrfip_init()**
It initializes the mstrfip library. It must be called before doing anything else.
@return 0 on success, otherwise-1 and errno is appropriately set

**void mstrfip_exit()**
It releases the resources allocated by mstrfip_init(). It must be called when you stop to use this library.

**struct mstrfip_dev *mstrfip_open_by_fmc(uint32_t device_id)**
Open a mstrfip device node using PCIe slot ID
@param[in] **device_id**
@return It returns an anonymous mstrfip_dev structure on success. On error, NULL is returned, and errno is set appropriately.

**struct mstrfip_dev *mstrfip_open_by_lun(uint32_t lun)**
Open a mstrfip device node using logical number unit
@param[in] **lun** an integer argument to select the device or negative number to take the first one found.
@return It returns an anonymous mstrfip_dev structure on success. On error, NULL is returned, and errno is set appropriately.

**void mstrfip_close(struct mstrfip_dev *dev)**
It closes a mstrfip device opened with one of the following function: mstrfip_open_by_lun(), mstrfip_open_by_fmc().
@param[in] **dev** device token

## II. MasterFIP configuration

**enum mstrfip_bitrate:**

**MSTRFIP_BITRATE_31**, /**< 31.25 kb/s */
**MSTRFIP_BITRATE_1000**, /**< 1 Mb/s */
**MSTRFIP_BITRATE_2500**, /**< 2.5 Mb/s */

**struct mstrfip_hw_cfg** :

**int enable_ext_trig**; An external trigger is used to drive the macro cyle
**int enable_ext_trig_term**; A 50ohms termination is added to the external trigger
**int enable_int_trig**; Internal trigger is used to drive the macro cycle.
Set in conjunction with enable_ext_trig, it supplies to the absence of external trig and the system runs in free run mode and resynchronize when the external pulse is back.
**uint32_t turn_around_ustime;** set a specific turn around time for the master.
The FIP protocol specifies that this value should be inside a range according to the speed of the bus:
31.25KB/s tr_min:424us tr_max:440us
1MB/s tr_min:10us tr_max:33us
2.5MB/s tr_min:13us tr_max:40us
Set to 0, the maximum value of the range is taken as turn around time for the master.

**struct mstrfip_sw_cfg :**

**int irq_thread_prio;** in the range 1 to 99 as defined by Linux RT schedulers.
**int diag_thread_prio;** in the range 1 to 99 as defined by Linux RT schedulers.
**uint32_t event_ustimeout;** masterfip library waits on event programmed by the application,
and calls application's function registered with the event. The wait stops when the time out has elapsed and application's error handler is called. By default the time out is set to 1.1*macrocycle.
**void (* mstrfip_error_handler)(struct mstrfip_dev *dev, enum mstrfip_error_list error);**
Application error handler called by the library in case of run time error.

**int mstrfip_rtapp_reset(struct mstrfip_dev *dev)**
Restarts both real-time applications.
@param[in] **dev** device token
@return 0 on success,-1 on error and errno is set appropriately

**int mstrfip_hw_speed_get(struct mstrfip_dev *dev, enum mstrfip_bitrate *bitrate)**
Read from the HW the bus speed.
@param[in] **dev** device token
@param[in, out] **bitrate** returned speed as one member of **enum mstrfip_bitrate**

**int mstrfip_hw_response_time_get(struct mstrfip_dev *dev, int agent_count, int agent_addr_list[], uint32_t response_nstime_list[])**
This process runs only once at startup and is used to collect the response (turnaround) time of each nodes in the bus.
Returns the response time in nanosecond for all the requested agents identified by their address on the bus.
@param[in] **dev** device token
@param[in] **agent_count** number of agents, defining the size of agent_addr_list and response_nstime_list.
@param[in] **agent_addr_list** list of addresses for the requested agents.
@param[in, out] **response_nstime** allocated by the caller and filled in with response time in nano second for requested agents. A value of 0ns means that the communication with the agent has timed out.
@return 0 on success,-1 on error and errno is set appropriately

**int mstrfip_hw_cfg_set(struct mstrfip_dev *dev, struct mstrfip_hw_cfg *cfg)**
Set hardware configuration
@param[in] **dev** device token
@param[in] **cfg** desired hardware configuration.
@return 0 on success,-1 on error and errno is set appropriately

**int mstrfip_sw_cfg_set(struct mstrfip_dev *dev, struct mstrfip_sw_cfg *cfg)**
Set software configuration.

@param[in] **dev** device token
@param[in] **cfg** desired software configuration.
@return 0 on success,-1 on error and errno is set appropriately

## *III. Macrocycle configuration*

**struct mstrfip_macrocycle *mstrfip_macrocycle_create(struct mstrfip_dev *dev)**
Creates a macrocycle object associated to the given masterfip device
@param[in] **dev** device token
@return It returns an anonymous macrocycle structure on success. On error, NULL is returned, and errno is set appropriately.

**int mstrfip_macrocycle_delete (struct mstrfip_dev *dev, struct mstrfip_macrocycle *mcycle)**
Delete a macrocycle object and its associated resources (in the FPGA).
@param[in] **dev** device token
@param[in] **mcycle** pointer to the macrocycle to delete.
@return 0 on success,-1 on error and errno is set appropriately

**int mstrfip_macrocycle_reset( struct mstrfip_dev *dev, struct mstrfip_macrocycle *mcycle)**
Reset a macrocycle object by deleting all its associated resources (in the FPGA).
@param[in] **dev** device token
@param[in] **mcycle** pointer to the macrocycle to reset
@return 0 on success,-1 on error and errno is set appropriately

**int mstrfip_macrocycle_isvalid (struct mstrfip_dev *dev, struct mstrfip_macrocycle *mcycle)**
Check if the given macrocycle has a valid configuration in order to be loaded.
@param[in] **dev** device token
@param[in] **mcycle** pointer to the macrocycle to check
@return 1 if the macrocycle is valid and 0 if it is invalid.

**enum mstrfip_data_type**
**MSTRFIP_PER_VAR,** periodic variable
**MSTRFIP_APER_VAR,** aperiodic variable
**MSTRFIP_IDENT_VAR,** identification aperiodic variable
**MSTRFIP_APER_MSG,** aperiodic message
**MSTRFIP_APER_MSG_ACK,** aperiodic acknowledged message

**enum mstrfip_data_status**
**MSTRFIP_DATA_OK,** valid FIP data
**MSTRFIP_DATA_FRAME_ERROR,** frame fault. @see frame_error field for details
**MSTRFIP_DATA_PAYLOAD_ERROR,** payload fault. @see payload_error for details
**MSTRFIP_DATA_NOT_RECEIVED,** no new data received since last reading.

**enum mstrfip_frame_errors**
**MSTRFIP_FRAME_OK** no frame error
**MSTRFIP_FRAME_TMO** time out. No reply frame from agent until silence time expired
**MSTRFIP_FRAME_ERR** CRC frame error
**MSTRFIP_FRAME_BAD_CTRL** bad control byte. Agent replied but the control byte is incorrect
**MSTRFIP_FRAME_BAD_PDU** bad PDU byte. Agent replied but the PDU byte is incorrect (N/A for messages)
**MSTRFIP_FRAME_BAD_BSZ** wrong number of bytes. The payload doesn't have the right size

**enum mstrfip_payload_errors**
**MSTRFIP_FRAME_PAYLOAD_OK** payload is OK
**MSTRFIP_FRAME_PAYLOAD_NOT_REFRESH** not a fresh payload sent by the agent (MPS != 0x5)
**MSTRFIP_FRAME_PAYLOAD_NOT_SIGNIFICANT** the payload content is not significant (MPS != 0x5)
**MSTRFIP_FRAME_PAYLOAD_NOT_PROMPT** not implemented for the time being

**struct mstrfip_data**

The same structure is used for ID_DAT or RP_DAT* frames, produced or consumed

{

enum mstrfip_data_type type; type of FIP data object

uint16_t id; identification encoded in 2bytes: |Byte1: var number (for msg it´s=0) | Byte0: agent address|.

uint8_t* buffer; data buffer containing the payload value

uint32_t bsz; size of the payload. For variables the payload size is constant but for messages, size is dynamic.

enum mstrfip_data_status status; global status. @see enum mstrfip_data_status

enum mstrfip_frame_errors frame_error; detailled frame error. @see enum mstrfip_frame_errors

uint32_t payload_error; detailed payload errors.

payload_errors should be decoded using enum mstrfip_payload_errors (bits are not exclusives)

void *priv; private object linked to the FIP data for internal purpose of the library.

}

struct mstrfip_data_cfg:

int id; variable id made of 2 bytes: |Byte1: var number|Byte0: agent address|

enum mstrfip_data_flags flags;

MSTRFIP_DATA_FLAGS_PROD flag for a produced variable/message

MSTRFIP_DATA_FLAGS_CONS flag for a consumed variable/message

MSTRFIP_DATA_FLAGS_ACK flag for a message being acknowledged.

int max_bsz; max size in bytes of the payload. FIP protocol defines a limit of 263 bytes.

void (* mstrfip_data_handler)(struct mstrfip_dev *dev, struct mstrfip_data *data, struct mstrfip_irq *irq);

User callback called when this FIP data is scheduled on the bus.

struct mstrfip_data *mstrfip_var_create(struct mstrfip_macrocycle * mcycle, struct mstrfip_data_cfg *cfg)

Create a periodic variable object for the given macrocycle, according to the given configuration.

@param[in] mcycle pointer to the macrocycle

@param[in] cfg periodic variable configuration (@see struct mstrfip_data_cfg)

@return a pointer to the new object on success, NULL on error and errno is set appropriately.

struct mstrfip_data *mstrfip_msg_create(struct mstrfip_macrocycle * mcycle, struct mstrfip_data_cfg *cfg)

Create an aperiodic message object for the given macrocycle, according to the given configuration.

@param[in] mcycle pointer to the macrocycle

@param[in] cfg aperiodic message configuration (@see struct mstrfip_data_cfg)

@return a pointer to the new object on success, NULL on error and errno is set appropriately.

struct mstrfip_data *mstrfip_ident_var_create(struct mstrfip_macrocycle * mcycle, uint32_t agent_address)

Create an aperiodic identification variable object for the given macrocycle, according to the given configuration.

@param[in] mcycle pointer to the macrocycle

@param[in] agent_address address of the agent for which the identification variable is created

@return a pointer to the new object on success, NULL on error and errno is set appropriately.

struct mstrfip_per_var_wind_cfg

struct mstrfip_data **varlist; list of pointers to periodic var objects, to be scheduled in this periodic window.

unsigned int var_count; number of pointers to periodic variables in varlist.

int mstrfip_per_var_wind_append (struct mstrfip_macrocycle * mcycle, struct mstrfip_per_var_wind_cfg *cfg)

Append a periodic variable window in the macro cycle using the given configuration.

@param[in] mcycle pointer to the given macrocyle

@param[in] cfg periodic window configuration (@see struct mstrfip_per_var_wind_cfg)

@return 0 on success,-1 on error and errno is set appropriately

struct mstrfip_aper_var_wind_cfg {

uint32_t end_ustime; time (in us), relative to the macro cycle start, at which the aperiodic window ends.

unsigned int enable_diag; Enabling diagnostic implies an extra traffic on the bus

- Periodic traffic: a periodic window scheduling the two variables of the fip diag (0x067F and 0x057F) is inserted in the macro-cycle to ensure a good survey of the electrical quality of the bus. This periodic window is always inserted in the beginning of the aperiodic window.

- Aperiodic traffic: In the aperiodic window, after having served application's requests (like schedule identification variables), if some time remains, masterFIP can schedule presence variables to keep updated the list of presence, and identification of the FIP diag.

**struct mstrfip_data **ident_varlist;** List of pointers to identification variables to be scheduled during the aperiodic window if there is enough time.

**int ident_var_count;** number of identification variables

**void (* mstrfip_ident_var_handler) (struct mstrfip_dev *dev, struct mstrfip_irq *irq);**
callback registered by the application, to be notified when the requests to schedule identification var has been completed.(see mstrfip_ident_request() to know how to request identification traffic)

**int mstrfip_aper_var_wind_append(struct mstrfip_macrocycle * mcycle, struct mstrfip_aper_var_wind_cfg *cfg)**
Append an aperiodic variable window in the macro cycle using the given configuration.
@param[in] **mcycle** pointer to the given macrocyle
@param[in] **cfg** aperiodic variable window configuration (@see **struct mstrfip_aper_var_wind_cfg** )
@return 0 on success,-1 on error and errno is set appropriately

**struct mstrfip_aper_msg_wind_cfg {**

**uint32_t end_ustime;** time (in us), relative to the macro cycle start, at which the aperiodic window ends.

**unsigned int prod_msg_fifo_size;** Maximum number of pending aperiodic messages request sent by the application, which can be accumulated till they are processed in the coming aperiodic message windows.

**unsigned int cons_msg_fifo_size;** Maximum number of pending aperiodic message request sent by agents during a periodic window which can be accumulated till they are processed by the master in the coming aperiodic message windows.

**void (* mstrfip_cons_msg_handler)(struct mstrfip_dev *dev, struct mstrfip_data *data, struct mstrfip_irq *irq);** Global callback for consumed message as opposed to callback registered per consumed message (see struct mstrfip_data_cfg). By registering this callback, the application is notified at the end of an aperiodic msg window, in order to get all aperiodic messages received during this aperiodic msg window. In case application has registered callback per consumed message in addition to this global callback, only the global will be considered. If none of them is registered (neither global, neither individual), an error is returned when application adds an aperiodic msg window in the macro cycle.

**void (* mstrfip_prod_msg_handler) (struct mstrfip_dev *dev, struct mstrfip_data *data, struct mstrfip_irq *irq);** Global callback for produced message as opposed to calback registered per produced message (see struct mstrfip_data_cfg). By registering this callback, the application is notified at the end of an aperiodic msg window, in order to get a status of aperiodic messages sent by the master during this aperiodic msg window. In case application has registered callback per produced message in addition to this global callback, only the global will be considered. Unlike for consumed message, if none of them is registered (neither global, neither individual), it's not an error and simply means that the application is not interested to be notified about produced message.

**int mstrfip_aper_msg_wind_append(struct mstrfip_macrocycle *mcycle, struct mstrfip_aper_msg_wind_cfg *cfg)**
Append an aperiodic message window in the macro cycle using the given configuration.
@param[in] **mcycle** pointer to the given macrocyle
@param[in] **cfg** aperiodic message window configuration (@see **struct mstrfip_aper_msg_wind_cfg**)
@return 0 on success,-1 on error and errno is set appropriately

**int mstrfip_wait_wind_append (struct mstrfip_macrocycle *, uint32_t silent_wait, uint32_t us_cycle_end);**
Append a wait window in the macro cycle. Note a macrocycle configuration should always end with a wait-window. This marker is used to determine the macrocycle length
@param[in] **mcycle** pointer to the given macrocyle
@param[in] **silent_wait** set to 1 the wait window is silent otherwise stuffing frames are scheduled.
@param[in] **us_cycle_end** time (in us), relative to the macro cycle start, at which the wait window ends.
@return 0 on success,-1 on error and errno is set appropriately

**int mstrfip_ba_load(struct mstrfip_dev *dev, struct mstrfip_macrocycle *mcycle);**
Loads the given macrocycle associated with this materFIP device (@see **mstrfip_macrocycle_create()**)
@param[in] **dev** device token

@param[in] **mcycle** pointer to the macrocyle to be loaded.
@return 0 on success,-1 on error and errno is set appropriately

**int mstrfip_ba_start(struct mstrfip_dev *dev);**

Starts the loaded macrocycle for the given masterFIP device.
@param[in] **dev** device token
@return 0 on success,-1 on error and errno is set appropriately

**int mstrfip_ba_stop(struct mstrfip_dev *dev);**

Stops the running macrocycle for the given masterFIP device. Then it can be restarted using **mstrfip_ba_start().**
@param[in] **dev** device token
@return 0 on success,-1 on error and errno is set appropriately

**int mstrfip_ba_reset (struct mstrfip_dev *dev);**

Resets the running macrocycle for the given masterFIP device. This will require to reload a macrocycle and to start it by using respectively **mstrfip_ba_load()** and **mstrfip_ba_start()**
@param[in] **dev** device token
@return 0 on success,-1 on error and errno is set appropriately

## IV. MasterFIP runtime:

**void mstrfip_var_update (struct mstrfip_dev *dev, struct mstrfip_data *var);**

Update the content of the given periodic variable. This call should be done from a callback registered with the macrocycle, in order to be synchronized with the macrocycle execution. Otherwise chances are that you get an error. This function doesn't return any error, but periodic variable contains errors, if any. (@see **struct mstrfip_data**)

@param[in] **dev** device token

@param[in] **var** pointer to the periodic var to be updated.

**int mstrfip_varlist_update (struct mstrfip_dev *dev, struct mstrfip_data **varlist, int var_count);**

Update the content of the given list of periodic variables. For more details @see **mstrfip_var_update()**

@param[in] **dev** device token

@param[in] **varlist** list of pointers to periodic var to be updated.

@param[in] **var_count** varlist element count.

@return the number of periodic vars updated

**void mstrfip_msg_update (struct mstrfip_dev *dev, struct mstrfip_data *msg);**

Update the content of the given aperiodic message. For more details @see **mstrfip_var_update().**

@param[in] **dev** device token

@param[in] **msg** pointer to the aperiodic msg to be updated.

**void mstrfip_ident_update (struct mstrfip_dev *dev, struct mstrfip_data *ident_var);**

Update the content of the given identification variable. For more details @see **mstrfip_var_update().**

@param[in] **dev** device token

@param[in] **ident_var** pointer to the identification variable to be updated.

**int mstrfip_identlist_update (struct mstrfip_dev *dev, struct mstrfip_data **identlist, int ident_count);**

Update the content of the given list of identification variables. For more details @see **mstrfip_var_update()**

@param[in] **dev** device token

@param[in] **identlist** list of pointers to identification variables to be updated.

@param[in] ident**_count** identlist element count.

@return the number of identification variables updated

**int mstrfip_var_write (struct mstrfip_dev *dev, struct mstrfip_data *var);**

Send the new payload of the given periodic variable to the masterFIP device. This new payload is kept by the masterfip, waiting for the moment where the macrocycle manager sends it on the bus.

@param[in] **dev** device token

@param[in] **var** pointer to the periodic variable containing the new payload to write.

@return 0 on success,-1 on error and errno is set appropriately

**int mstrfip_msg_write (struct mstrfip_dev *dev, struct mstrfip_data *msg);**

Send the new payload of the given aperiodic message to the masterFIP device. This new payload is kept by the masterfip, waiting for the moment where the macrocycle manager sends it on the bus.

@param[in] **dev** device token

@param[in] **msg** pointer to the aperiodic message containing the new payload to write.

@return 0 on success,-1 on error and errno is set appropriately

**int mstrfip_ident_request (struct mstrfip_dev *dev, struct mstrfip_data **identlist, int ident_count);**

Send a request to the masterFIP device for scheduling a batch of identification variables. This request is kept by the masterfip, waiting for the moment where the macrocycle manager schedules an aperiodic window.

@param[in] **dev** device token

@param[in] **identlist** list of pointers to identification variables to be scheduled.

@param[in] ident**_count** identlist element count.

@return 0 on success,-1 on error and errno is set appropriately

**struct mstrfip_report:**

**uint32_t tx_ok;** number of **successful** TX during a macrocycle

**uint32_t tx_err;** number of failed TX since the system start.

**uint32_t rx_ok;** number of successful RX during a macrocycle

**uint32_t rx_err;** number of failed RX since the system start

**uint32_t fd_tx_err;** number of fieldrive TX error since the system start

**uint32_t fd_tx_err_hwtime;** time when the last fieldrive TX error occurred.
**uint32_t fd_cd;** number of fieldrive carrier detect since the system start.
**uint32_t fd_tx_watchdog;** number of fieldrive watchdog since the system start.
**uint32_t fd_tx_watchdog_hwtime;** time when the last fieldrive watchdog occurred.
**uint32_t rx_tmo;** number of timed-out since the system start. Time-out on presence frame are not counted.
**uint32_t cycles;** number of executed macrocycles since the system start.
**uint32_t ext_sync_pulse_count;** number of external trigger seen by RT sw. since the system start.
**uint32_t ext_sync_pulse_missed_count;** number of external trigger missed by RT sw. since the system start.
**uint32_t int_sync_pulse_count;** number of internal trigger since the system start.
**enum mstrfip_ba_fsm ba_state;** current bus arbiter state
**enum mstrfip_cycle_fsm cycle_state;** current macrocycle state.
**uint32_t temp;** current temperature measurment of the masterfip device.

**int mstrfip_report_get (struct mstrfip_dev *dev, struct mstrfip_report *report);**

Return run time statistics for the given masterfip device. This report includes TX and RX error counters since system start, in addition to other useful informations helping to diagnose the running sytsem. (@see **struct mstrfip_report)**
@param[in] **dev** device token
@return 0 on success,-1 on error and errno is set appropriately

**struct mstrfip_diag_shm* mstrfip_diag_get(struct mstrfip_dev *dev);**

Should not be used by application. This function allows existing F**IP** diagnostic tool to see the new masterFIP as the old one. Not documented on purpose.

# 7. Example application

The following code snippets show the main steps required to write a simple masterFIP application using the API described in Section 6.

- Open device:

```
// mandatory init
res = mstrfip_init();
// Open the device by providing its lun
masterfip_dev = mstrfip_open_by_lun(lun);
```

- Reset mock turtle application

Before programming any FIP traffic, it is recommended to reset first the real time application running on mock turtle software cores.

```
// reset mock turtle software
mstrfip_rtapp_reset(masterfip_dev)
```

- Check the FIP bus speed:

FIP bus supports 3 different speeds, requiring a specific masterFIP board for each speed. Using this function an application can check if it is the right board driving the FIP bus.

```
enum mstrfip_bitrate bus_speed;
// Verify if the FIP bus speed is the expected one
res = mstrfip_hw_speed_get(masterfip_dev, &bus_speed);
```

- Set hardware config

In this example the application uses an external trigger connected to the external input of the masterFIP board, without termination. In addition, instead of using the default turnaround time for the master, the application sets the turnaround time at 20 us.

```
struct mstrfip_hw_cfg hw_cfg = {0};
/* set HW config */
hw_cfg.enable_ext_trig = 1;
hw_cfg.enable_ext_trig_term = 0
hw_cfg.turn_around_ustime = 20;
res = mstrfip_hw_cfg_set(masterfip_dev, &hw_cfg) ;
```

- Set software config

Basically here, the application registers its error handler and keeps the other parameters configured with their default values.

```
struct mstrfip_sw_cfg sw_cfg = {0};
/* set SW config */
sw_cfg.mstrfip_error_handler = my_mstrfip_error_handler;
res = mstrfip_sw_cfg_set(masterfip_dev, &sw_cfg);
```

- Create Macrocycle object

In order to configure a FIP macrocycle, the application first creates a macrocycle object. For applications needing to switch on the fly between different macrocycles, the application must instantiate macrocycle objects as many as necessary. Note that there is no warrantee in the amount of data loss while switching between macrocycles.

```
struct mstrfip_macrocycle *mcycle;
mcycle = mstrfip_macrocycle_create(masterfip_dev);
```

- Create FIP data objects

The FIP data object called “mstrfip-data” is the software representation of any FIP frame; it can be a periodic variable, an aperiodic variable or an aperiodic message. The library provides a specific factory method for each specific FIP frame because each one requires specific configuration parameters. In this example the application creates two produced periodic variables and two consumed periodic variables.

- Produced variables loop: the application registers a handler on the last produced periodic var, in order to be notified when the masterFIP schedules it on the bus.

```
struct mstrfip_data_cfg var_cfg = {0};

int var_id = 0x2; /* produced variable id */

/* build prod varlist */

for (int agt_idx = 0; agt_idx <2; agt_idx++) {

        var_cfg.max_bsz = 64;

        var_cfg.id = (0xFF00 & (var_id << 8)) | (agt_idx + 1);

        var_cfg.flags = MSTRFIP_DATA_FLAGS_PROD;

        /* Set callback on the last produced periodic var */

        var_cfg.mstrfip_data_handler = (agt_idx == 2 - 1) ? my_prod_var_handler : NULL;

        prod_varlist[agt_idx] = mstrfip_var_create(mcycle, &var_cfg);}
```

- Consumed variables loop: similar approach for consumed variables. Thanks to the handler registered on the last periodic consumed var, the application will be notified at the end of the of the periodic consumed var traffic in order to collect all the acquisition data.

```
struct mstrfip_data_cfg var_cfg = {0};

int var_id = 0x3; /*consumed varaiable id */

/* build consumed varlist *

for (int agt_idx = 0; agt_idx <2; agt_idx++) {

        var_cfg.max_bsz = 100;

        var_cfg.id = (0xFF00 & (var_id << 8)) | (agt_idx + 1);

        var_cfg.flags = MSTRFIP_DATA_FLAGS_CONS;

        var_cfg.mstrfip_data_handler = (agt_idx == 2 - 1) ? my_cons_var_handler : NULL;

        cons_varlist[agt_idx] = mstrfip_var_create(mcycle, &var_cfg);

}
```

- **Configure macro cycle:**

Up to now, only FIP data objects have been instantiated. Now it's time to order them and to orchestrate them into a time unit called macrocycle. In order to schedule the different FIP data, which, as a reminder, can be periodic variables, aperiodic variables and aperiodic messages, the FIP protocol defines three respective time-windows: periodic_window, aperiodic_var_window and aperiodic_message_window. Programing a macrocycle consists in appending, according to the application needs, those time windows. In this simple example, the application needs to schedule only periodic variables and the macrocycle will contain two periodic windows, followed by an aperiodic window for diagnostics.

```
struct mstrfip_per_var_wind_cfg pwind_cfg = {0};
struct mstrfip_aper_var_wind_cfg apwind_cfg = {0};

/* append a periodic window for produced variables */
pwind_cfg.varlist = prod_varlist; /* periodic var list */
pwind_cfg.var_count = 2; /* periodic var count */
res = mstrfip_per_var_wind_append(mcycle, &pwind_cfg);

/* append a periodic window for consumed varaiables */
pwind_cfg.varlist = cons_varlist; /*periodic var list */
pwind_cfg.var_count = 2; /* periodic var count */
res = mstrfip_per_var_wind_append(mcycle, &pwind_cfg) ;

/* append aperiodic window to enable diagnostic */
apwind_cfg.end_ustime = mcycle_uslength * 0.9; /* ends at 90% of macrocycle */
apwind_cfg.enable_diag = 1; /* enable diagnostic */
res = mstrfip_aper_var_wind_append(mcycle, &apwind_cfg)
```

The macrocycle configuration should always end by a wait_window. If not, the macrocycle configuration is rejected. Within this window there is no bus activity and the masterFIP is only awaiting for a trigger for a new macrocycle, through an ext_sync_pulse or an internal counter. Here the application appends a silent wait window.

```
res = mstrfip_wait_wind_append(mcycle, 1, mcycle_uslength);
```

- **Macrocycle load and start:**

Once a macro cycle is programmed and validated (see masterFIP lib API), it is ready to be loaded and

started. Once started, the masterFIP device schedules FIP frames, and the registered application handlers are called according to the macrocycle programmation.

```
/* load macro cycle */
res = mstrfip_ba_load(masterfip_dev, mcycle);
/* start macro ccyle */
res = mstrfip_ba_start(masterfip_dev);
```

- **Application handlers:**

To finish, the next snippets show the typical code one can write in the application's callback. Our application has registered a callback on the last produced variable to prepare the new payload to be sent to the agent in the next macro cycle, and another callback on the last consumed variable to get the replies from the agents.

- produced var handler:

```
void  my_prod_var_handler(struct mstrfip_dev *dev, struct mstrfip_data *pvar, struct mstrfip_irq *irq)
{
        struct mstrfip_data * per_var;

        for (int agt_idx = 0; agt_idx < 2; ++agt_idx) {
                per_var = prod_varlist[agt_idx];
                // application specific in charge of computing the new payload
                compute_new_payload(pvar);
                res = mstrfip_var_write(ftb->dev, pvar); /* send new payload */
}
```

- consumed var handler:

```
void my_cons_var_handler(struct mstrfip_dev *dev, struct mstrfip_data *pvar, Struct mstrfip_irq *irq)
{       for (int agt_idx = 0; agt_idx < 2; ++agt_idx) {
                per_var = prod_varlist[agt_idx];
                mstrfip_var_update(masterfip_dev, per_var);
                switch (pvar->status) {
                case MSTRFIP_DATA_OK:
                        // data is ok. Consume it
                        break;
                case MSTRFIP_DATA_PAYLOAD_ERROR:
                        // payload fault: not freshed or not significant
                        process_error();
                        break;
                case MSTRFIP_DATA_NOT_RECEIVED:
                        // no data received since the last reading
                        process_error();
                        break;
                case MSTRFIP_DATA_FRAME_ERROR:
                        // HW error : time out, CRC_err, bad_nbytes,...
                        process_error();
                        break;
                }
        }
}
```

# 8. References


[1] masterFIP specification: https://edms.cern.ch/document/1457263/2

[2]WorldFIP OHWR project: http://www.ohwr.org/projects/cern-fip/wiki/WorldFIP

[3] MasterFIP mezzanine board OHWR project: http://www.ohwr.org/projects/fmc-worldfip/wiki

[4] SPEC OHWR project: http://www.ohwr.org/projects/spec/wiki

[5] FieldDrive information: http://www.ohwr.org/projects/cern-fip/wiki/FielDrive

[6] FieldTR information: http://www.ohwr.org/projects/cern-fip/wiki/FieldTR

[7] CTRI timing board: https://wikis.cern.ch/display/HT/CTRI+-+PCI+timing+receiver

[8] EDMS: EDA-03098

[9] Hardware logged issues: http://www.ohwr.org/projects/fmc-masterfip/issues

[10] masterFIP PTS: http://www.ohwr.org/projects/masterfip-tst/wiki

[11] Gateware technical note: http://www.ohwr.org/documents/407

[12] Mock Turtle: http://www.ohwr.org/attachments/2903/node-tech.pdf

[13] GN4124: http://www.ohwr.org/projects/gn4124-core/wiki

[14] fmc_masterFIP_core reg map: http://www.ohwr.org/projects/masterfip-gw/repository/revisions/eva_dev/changes/rtl/wbgen/masterfip_wbgen2_csr.html