White Rabbit Node Core
Technical Specification (draft)

Tomasz Włostowski/CERN BE-CO-HT

May 2014

1 Introduction

This document describes the technical details of the White Rabbit Node Core (WRNC). The WRNC is an HDL core of a generic control system node using White Rabbit as the means of communication and synchronization. We cover here the technical aspects of the design necessary to start up the HDL and driver development.

The proposed design may be applied for a variety of current and future projects in the BE-CO-HT section at CERN:

- LHC Instability Trigger Distribution system (LIST) [1].
- OASIS Trigger Distribution system.
- White Rabbit D3S (RF Over WR) transmitter/receiver node.
- White Rabbit Timing Master for AD/ELENA.
- White Rabbit to GMT (General Machine Timing) converter.
- White Rabbit Timing Receiver (a successor of the current CERN timing receiver - the CTR).

2 The WR Node Concept

The main assumption in the WRNC is that most, if not all tasks of a control system node at CERN can be done in software, executed in a deterministic way by an embedded CPU. Although this usually takes more time than a custom-designed HDL would take to do the same job, in large (in terms of physical distances) control networks, the processing latency takes only a tiny fraction of the delays introduced by kilometers of cabling. Savings in processing latency are often inversely proportional to the resources put in the gateware and driver development.

A notable example of a CPU-based control system nodes are the current GMT and Beam Synchronous Timing (BST) masters, which both use deterministic custom-designed processors. The WRNC extends and generalizes this idea, standardizing the communication interfaces, CPU architecture and development tools.

Figure 1 shows a simplified block diagram of the WRNC. The key ingredients in the proposed design are:

1. **Up to 8 CPU cores**, which:
   - Execute any code the user wishes, loaded, started and stopped on request. User applications are written in *bare metal* C (using the standard GNU tool set). Assembly may be used if necessary.
   - Communicate between each other through a dedicated shared memory.

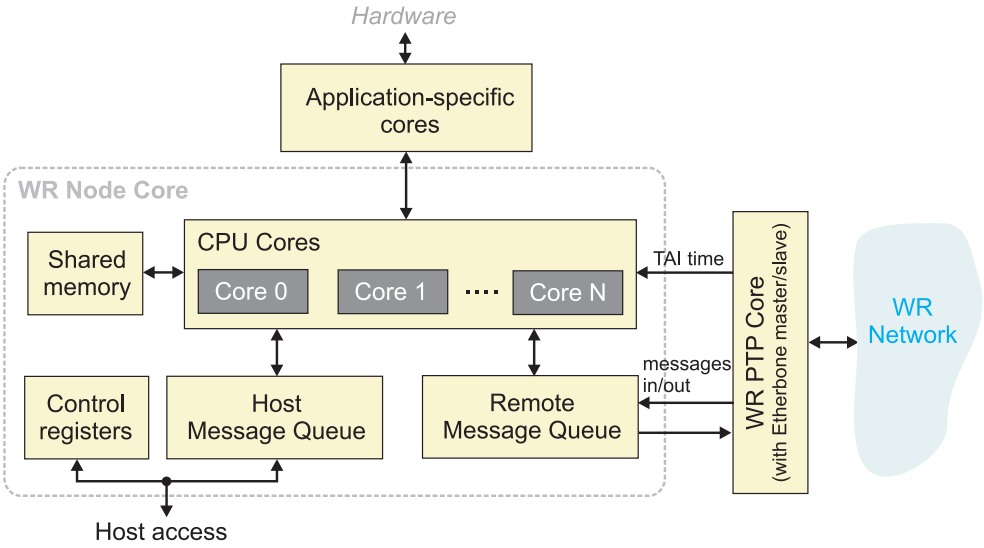

Figure 1: General concept of the WR Node Core.

- Have no interrupts to ensure deterministic execution timing.
- Keep code and data in a private memory for the same reason.

2. **Application-specific cores and hardware.**

   The cores are accessed by the CPUs through a Wishbone bus and may interface with external hardware of any sort. For example, a LIST node will interface with a TDC core and a Fine Delay core for time tagging/producing trigger pulses. Conversely, an RF distribution node will drive a DDS synthesizer and read data from a phase detector.

3. **International Atomic Time (TAI) provided by WR.**

   The CPUs and user cores in each node in the network will have direct access to TAI, with a granularity of 8 nanoseconds. The system clock of the CPUs may be synchronous to the WR reference frequency if necessary.

4. **Communication system**, incorporating two message queues, shared by the CPUs:

   - Remote Message Queue (RMQ), which exchanges messages with remote nodes in the WR network. The Etherbone protocol [2], developed by GSI, is used as the transport layer.
   - Host Message Queue (HMQ), exchanging messages between the CPUs and the host system. The HMQ is the primary means of communication between the node and the host software (e.g. FESA [7]). Direct access to shared CPU memory or other resources may be allowed, depending on the needs of the particular application.

3 The HDL Design

This section describes the technical details for the HDL design of the WRNC.

3.1 Top-level

The top level block diagram is shown in Figure 2. The WRNC talks to the external world through several interfaces:

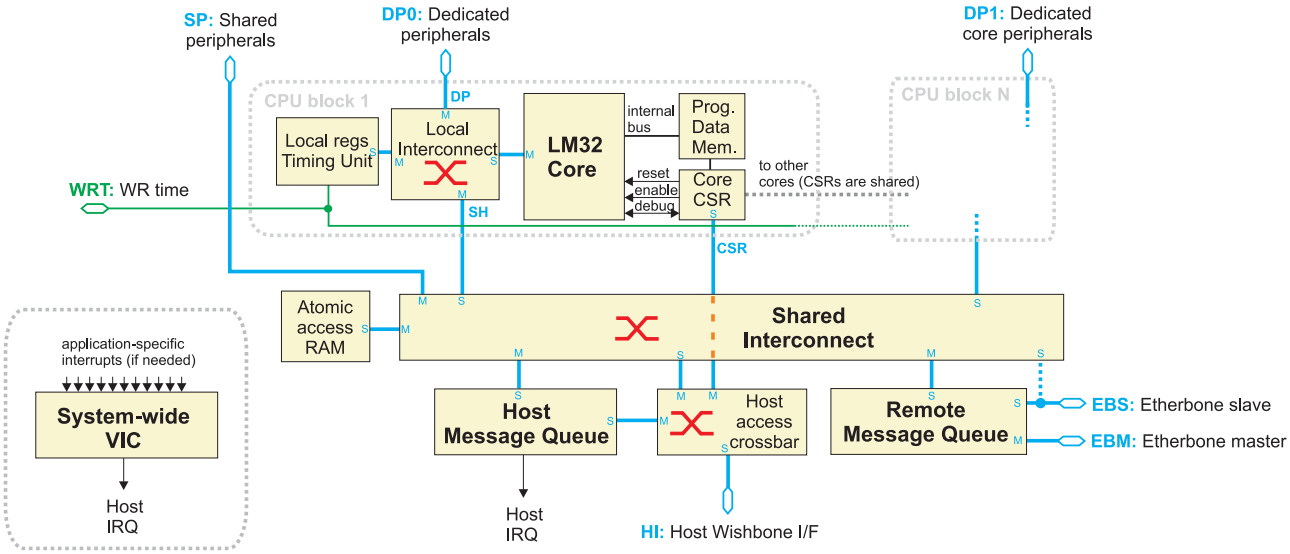

Figure 2: Structure and internal connections of the WRNC.

- `DPx`: A number (`N_CPU`) of Wishbone masters, connecting the respective CPU cores with their private peripherals. For example, the LIST node design will have a CPU core responsible for reading out timestamps from a TDC Core connected to its `DP` port.
- `SP`: shared Wishbone master, allowing all CPU cores to access certain shared peripherals. For example, an event table for the future GMT master could be stored in a large DDR memory connected through the `SP` port.
- `HI`: shared Wishbone slave, used by the host to configure the node, access shared peripherals and Host Message Queues.
- `EBM`: dedicated Wishbone master for communicating with the Etherbone master core (assembling outgoing Etherbone frames), hardwired to the RMQ.
- `EBS`: dedicated Wishbone slave for receiving writes from Etherbone to the RMQ. Optionally, the `EBS` can be used for raw Etherbone access to the entire Shared Interconnect.

3.2 CPU Core Blocks

Each node contains from one to eight (`N_CPU`) CPU Core Blocks (CPU CB). Each CPU core has its private code/data memory, a Local Registers block, a Timing Unit and a private interconnect. The CPU CB exposes the following interfaces:

- a Wishbone master `DP`, connected to the `DPx` port of the top level entity.
- a Wishbone master `SH`, for accessing the Shared Interconnect.
- a WR Core Timing Interface, for retrieving accurate WR time from the WR PTP Core [3].
- the Wishbone slave `CSR`, which allows the host to access the CPU Control Registers (CPU CSR) shared between all CPU Blocks in the node, for the purpose of firmware loading, CPU execution control and debugging.

3.2.1 CPU Implementation

The CPU core we have chosen is the Lattice Mico 32, a pipelined RISC CPU targeted specifically at FPGAs. The main reasons justifying the choice are:

- Full GCC tool set available, with a debugger.
- Very easy to tailor to one's needs (caches/additional instructions/memory configuration).
- Works on all major FPGA platforms.
- Our experience (HT's and GSI's designs already use the LM32).

The LM32 implementation for the WRNC uses the following configuration:

- **Per-core program/data memory**.

  The code and data for the program is stored in a built-in dual port RAM, separate for each CPU core. Port 1 is reserved for the instruction pipeline, port 2 - for the data pipeline. There are no caches in the way to preserve memory access determinism. The memory size is user-configurable, up to 1 MB per CPU. The CPUs cannot execute code from outside this memory. Initialization of the CPU P/D memory is done through the CPU CSR.

- **JTAG/Debug port:** allows real-time debugging, exposed to the host through the CPU CSR.
- **Instruction set:** baseline LM32 + fast multiply, barrel shifter, division, fast unconditional branches, sign extension.
- **Other features (or lack thereof):** no interrupts, no bus error handling.

All communication between the CPU core and the outside world is done through a single Wishbone master port, connected to the Local Interconnect.

3.2.2 CPU Local Interconnect

The Local Interconnect (CPU LI) is a non-blocking pipelined Wishbone crossbar with 1 slave and 3 master ports. The CPU uses it to communicate with:

- the Local Registers,
- the Dedicated Periperhal (`DPx`).
- the Message Queues, Shared Memory and Shared Peripherals through the Shared Interconnect.

.

As the LI is exclusive per each CPU and has only one slave port, there are no concurrency issues when accessing the CPU's private peripherals. Duration of every access to the Local Register and DP port depends only on the Wishbone acknowledge delay of the connected peripheral.

Table 1: Local Interconnect address map.

Range	Purpose
`0x00000 - 0xfffff`	CPU Program/Data memory
`0x100000 - 0x1000ff`	CPU Local Registers
`0x40000000 - 0x7fffffff`	Shared Interconnect
`0x80000000 - 0xffffffff`	Dedicated Peripheral port

3.2.3 CPU Local Registers and Timing Unit

The Local Registers (CPU LR) are the private registers of each core, providing non-arbitrated access to the most often used features of the node. The functionalities for the CPU LR are:

- Direct polling of the HMQ and RMQ. Since the CPUs have no interrupts, they spend most of their idle time polling the HMQ/RMQ for newly arrived messages. If the CPUs were reading the MQ status through the Shared Interconnect, it would quickly overload (because of many WB masters accessing a single WB slave). Placing the queue slot status bits inside the LR block lets the CPUs poll as frequently as they want without disturbing the SI.
- Providing information on the WR link and WR time status.
- Reading current WR time (TAI seconds and 8 ns cycles).

Table 2: CPU LR and TU registers.

Name	Longer name	Purpose
`POLL`	Polling Register	16 bits for slot status of the HMQ and 16 bits for the slot status of the RMQ.
`STAT`	Status Register	1 bit for WR link status, 1 bit for WR time status.
`TAI_CYCLES`	TAI Time 8ns Cycles	8 ns ticks since the beginning of the current TAI second. Reading this register latches the current TAI second in the `TAI_SECONDS` register.
`TAI_SECONDS`	TAI Time Seconds	TAI seconds, updated on read of `TAI_CYCLES`. Reading the seconds alone must be therefore preceded by a dummy read of `TAI_CYCLES`.
`TU_CTLx`	TU Control	See Table 3.
`TU_CNTRx`	TU Counter	Internal period counter, counting 8 ns ticks.
`TU_FLAGx`	TU Flag	Set to 1 on a TU event. Clear-on-read (together with `TU_HITSx`).
`TU_HITSx`	TU Hit counter	Increased by 1 on a TU event. Clear-on-read (together with `TU_FLAGx`).
`TU_CYCLESx`	TU Cycles	Cycles part of the trigger time.
`TU_SECONDSx`	TU Seconds	Seconds part of the trigger time.
`TU_REPEATx`	TU Repeat Count	If the event is periodic, write the count here.

The Timing Unit (CPU TU) lets the CPUs measure time and generate periodic events/delays. It operates by:

1. Waiting for a trigger condition: an offset relative to the current time, the PPS or a value of TAI.
2. Setting a clear-on-read flag in a register (since we don't have interrupts) when triggered.
3. Keeping a hit counter, counting up on each trigger event and reset on read of itself or the flag register.
4. Repeating point 2 a programmable number of times with a programmable period.

The following conditions should be sufficient to generate accurate machine cycles split into basic periods (1.2 s, a feature commonly used in the CERN GMT Master). The TU may have multiple independent channels, depending on the implementation.

The CPU LR and TU registers are organized as a single block with the structure described in Table 2.

Note: the Timing Unit functionality and registers are subject to change. The information above is an initial proposal for next-gen CERN timing, it's not relevant for the LIST node.

Table 3: `TU_CTLx` register fields.

Name	Size	Purpose
`BUSY`	bit	0: when the TU is not doing anything at the moment, 1: TU busy.
`ARM`	bit	1: to arm the TU in given `MODE`, 0: no action.
`MODE`	3 bits	Counter mode: absolute/relative/PPS.
`CONT`	bit	1: continuous operation, 0: repeat count defined in `TU_REPEAT`.
`STOP`	bit	1: immediately stops the TU channel. 0: no action.

3.3 Shared Control Registers

The Shared Control Registers (CSR) expose the following functionality for the host:

- CPU execution control: enable/reset one or more CPUs.
- CPU debugging: raw access to the JTAG host in each of the CPUs.
- Program/Data memory access for firmware download.
- Auto-enumeration/detection info for the drivers/library (TBD).

Table 4: The CSR registers.

Name	Longer name	Purpose
APP_ID	Application ID Register	User application ID for the software library to identify the node configuration
RESET	CPU Reset Register	Each bit corresponds to the reset line of a CPU Core. 1 holds the CPU in reset.
ENABLE	CPU Enable Register	Each bit corresponds to the enable line of a CPU Core. 1 pauses code execution.
CORE_COUNT	CPU Core count	Number of CPU cores in this node.
CORE_SEL	CPU Core select.	Selects the CPU core whose memory/JTAG port will be accessible through the UADDR, UDATA and DEBUG registers.
UADDR	CPU Address Register.	Provides indirect access to the memory of CPU selected in the CORE_SEL register. Write the destination address before storing/reading data through the UDATA register.
UDATA	CPU Data Register.	Provides indirect access to the CPU memory. Reads/writes the memory cell pointed by UADDR.
DEBUG	Debug Register.	Raw LM32 JTAG access. See Table 5.

Table 5: The DEBUG register layout.

Name	Size	Purpose
DATA	8 bits	LM32 Debug Unit data (read/write)
ADDR	3 bits	LM32 Debug Unit address (read/write)
JTCK	bit	Writing 1 generates a JTAG TCK clock pulse.
JTRST	bit	Writing 1 resets the JTAG TAP.
UPDATE	bit	Writing 1 updates the JTAG address/data with the values in ADDR and DATA fields.

3.4 Shared Memory

The Shared Memory (SMEM) is a small block of RAM accessible by all CPU Cores, the Host, and Etherbone (optional). The SMEM's purpose is data/event exchange between multiple CPU cores, therefore it comes with hardware support for atomic operations:

- Increase/decrease (used by semaphores and queue pointers).
- Bit set/clear/flip (used by mutexes, flags and events).

The size of SMEM is fixed to 8 kB (2048 32-bit words, no byte access), with mirroring into multiple address ranges, each responsible for a single atomic operation. The address layout is described in Table 6.

Table 6: Shared Memory layout.

Address	Purpose
`0x0000 - 0x1fff`	Direct read/write access.
`0x2000 - 0x3fff`	Atomic set area. On write, the corresponding memory cell is ORed with the written value.
`0x4000 - 0x5fff`	Atomic clear area. On write, the corresponding memory cell is ANDed with the complement of written value.
`0x6000 - 0x7fff`	Atomic flip area. On write, the corresponding memory cell is XORed with the written value.
`0x8000 - 0x9fff`	Atomic add area. On write, the corresponding memory cell is increased by the written value. Negative values are supported.

3.5 Message Queues

The Message Queues (MQs) are the heart of the WRNC's communication system, providing a simple and robust way of sending/receiving messages. The structure of the MQs is depicted in Figures 3 and 4.

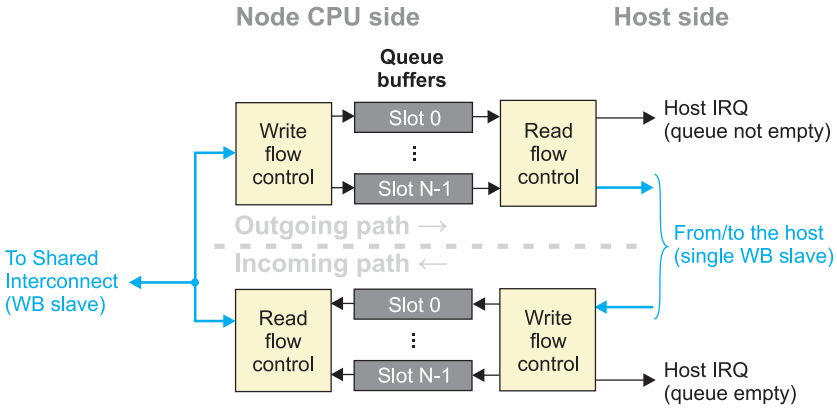

Figure 3: Block diagram of the Host Message Queue.

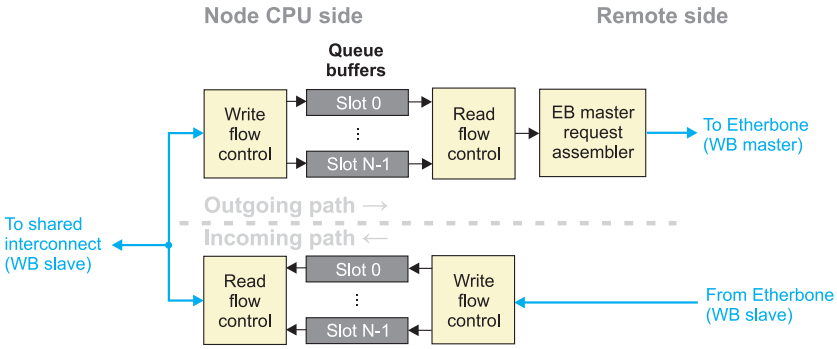

Figure 4: Block diagram of the Remote Message Queue.

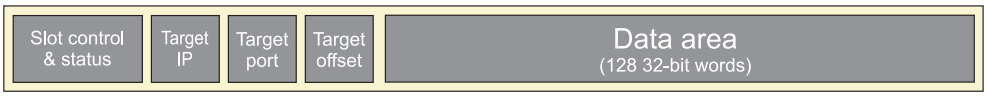

Figure 5: Layout of an MQ slot.

The principles of MQs operation are listed below:

- **MQs are organized as multi-word FIFOs**: The transmitter writes a number of words to an MQ outgoing slot and marks it as ready to send. The receiver side gets an indication that its MQ incoming slot is not empty, reads out its contents and indicates that it has processed the message.
- **MQ have a number of incoming and outgoing slots**. The direction is relative to the node CPU (i.e. the node CPU receives messages from an incoming slot and sends them through an outgoing slot). Multiple slots allow handling independent message streams from independent sources. Each in/out slot consists of control/status register(s) and a payload area, allowing to transfer up to 128 32-bit words (see Figure 5). If the node has more than one CPU core, the CPUs must have assigned individual slots

or an SMEM-based mechanism for ensuring atomic send/receive operations must be implemented. The number of slots and the size and width of each slot are configurable by a generic parameter.

- **Each slot can buffer multiple messages**. The depth of the slots is configurable in the VHDL entity.
- **MQs ensure integrity** of the messages: if the message is not received completely (i.e. the Etherbone slave core reported an error), it is not received at all. Future systems using the WRNC will use an external Forward Error Correction block between the Etherbone master/slave and the WRPC to make sure no messages are dropped even in case of an error.
- **There is no flow control**: if a MQ slot becomes full, the incoming messages are dropped. Users may implement flow control in software if needed, although in all WRNC applications currently foreseen, any buffer contention is pathological.

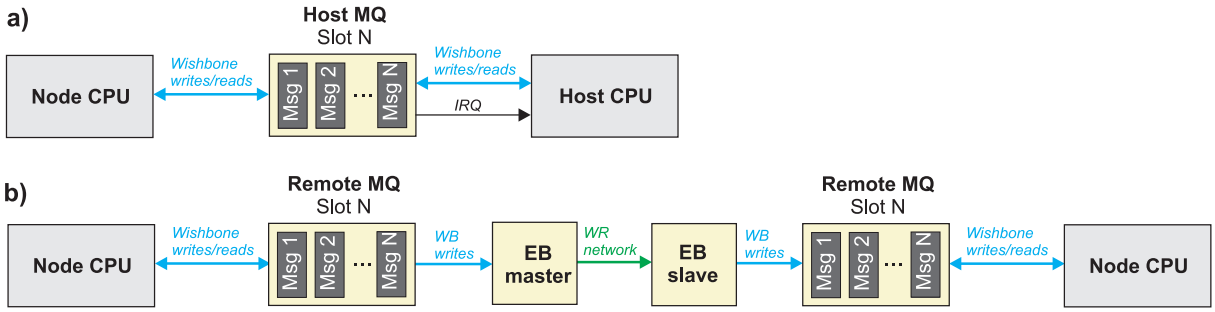

Figure 6: Transmission of a message using HMQs (a) and RMQs (b).

MQs exist in two variants:

- The HMQ is nothing more than a fancy FIFO between the WRN CPU Cores and the host processor. Its purpose is sending commands and exchanging data with the CPU CBs. On the host side, the empty/full status of the queue is indicated by interrupts (see Figure 6a). The HMQ has two Wishbone slave ports, exposing identical register blocks to the CPU CBs and the host. A CPU writes a message to an outgoing slot in the CPU CB-side registers and the host reads it from the outgoing slot at the same address in the Host-side registers (and *vice-versa* in reverse direction).
- The RMQ casts every transmitted message into an Etherbone request, containing a sequence of write operations to MQ slot registers that will make the message appear in a given incoming slot of the RMQ in the selected remote node (see Figure 6b). Message transmission using the RMQ requires setting up the IP configuration in the `TARGET_` registers and initialization of the Etherbone core. The RMQ has a Wishbone slave port (for communication with the CPU CBs), a Wishbone Master port producing requests for the external Etherbone master core and a Wishbone slave port for accepting remote Etherbone traffic.

Both modules share a large part of the VHDL design and same register layout (in case of the HMQ, target IP/port/offset registers are not used).

Table 7: Message Queue address map.

Range	Purpose
0x0000 - 0x0fff	MQ Global Control/Status
0x4000 - 0x43ff	Incoming Slot 0 (and so on)
0x7c00 - 0x7fff	Incoming Slot 15
0x8000 - 0x83ff	Outgoing Slot 0 (and so on)
0xbc00 - 0xbfff	Outgoing Slot 15

Table 8: Message Queue Global Control/Status.

Name	Longer name	Purpose
SLOT_COUNT	Slot Count	Number of available incoming/outgoing slots.
SLOT_STATUS	Slot Status	Each bit corresponds to one MQ slot and indicates whether it's not full (outgoing slots) and not empty (incoming slots).
SLOT_IRQ_MASK	Slot Interrupt Mask	Writing 1 to a particular bit enables host interrupt generation when the corresponding bit in SLOT_STATUS is set. HMQ only.
SLOT_IRQ_COALESCE	IRQ coalescing control	Messange count threshold and interrput timeout. Used by the driver to optimize buffer readout, exact layout TBD.

Table 9: COMMAND register layout.

Name	Size	Purpose
CLAIM	bit	Claims an incoming slot for sending a message. Issuing CLAIM on a full slot erases the oldest message making space for the new one.
READY	bit	Marks the message in the slot as ready to send. When issuing a READY command, write the message size to the SIZE field.
DISCARD	bit	Discards the current message. Issued after processing every incoming message.
PURGE	bit	Discards all messages in the slot.
SIZE	8 bits	Size of the message to be sent (number of data words).

Table 10: Message Queue STATUS register layout.

Name	Size	Purpose
FULL	bit	1: the slot is full.
EMPTY	bit	1: the slot is empty.
COUNT	8 bits	number of messages in the slot.
SIZE	8 bits	number of data words in the message in the data area.

Table 11: Message Queue slot registers.

Name	Longer name	Purpose
COMMAND	Command register	Input commands for this message queue slots (see Table 9).
STATUS	Slot Status Register	Slot status (see Table 10.
TARGET_IP	Target IP Address	IP address for the Etherbone message (RMQ only)
TARGET_PORT	Target UDP Port	UDP port for the Etherbone message (RMQ only)
TARGET_OFFSET	Target RMQ Address	Wishbone address of the target MQ slot (RMQ only)
DATA[0..127]	Payload	128 32-bit data words - payload to be sent/received.

3.5.1 MQ Operations

1. Transmitting a message through the HMQ (from the host or from the WRN CPU):
   - Check if the required slot is available through `STATUS.FULL` bit.
   - If full, keep polling the full bit or start the transmission anyway by writing a `CLAIM` command to the `COMMAND` register. Claiming a full slot will discard the oldest message from the queue.
   - Write the payload.
   - Mark the message as ready to send by issuing a `READY` command and indicate its size in the `SIZE` field of the `COMMAND` register.
   - The message TX status can be read from `STATUS.EMPTY` bit.
   - Sending `CLAIM` before `READY` will clear the currently assembled message.
2. Transmitting a message to another node through the RMQ (from the WRN CPU):
   - Configure the slot's target IP, port and RMQ address. This can be done once during initialization of the node.
   - Follow the same procedure as in point 1.
3. Receiving a message on the WRN CPU:
   - Poll the CPU CR `POLL` register bit corresponding to the RMQ/HMQ slot we want to receive from, until it becomes set.
   - Process the received message.
   - Discard it by writing `DISCARD` command to the slot control register.
4. Receiving a message on the host CPU:
   - Configure the interrupts.
   - An interrupt corresponding to the chosen slot arrives: process message and discard it through `DISCARD` command.

3.6 Shared Interconnect

The Shared Interconnect (SI) ensures communication between the CPU Cores, Message Queues, Shared Memory, Shared Peripherals and the host system. The SI is a full matrix, non-blocking Wishbone crossbar switch with the following arbitration policy:

- If multiple masters (e.g. two CPUs) talk to different slaves (e.g. CPU 0 to the the Shared Memory and CPU 2 to the HMQ), host accesses are executed concurrently.
- If multiple masters attempt to concurrently access the same slave (e.g. two CPUs trying to send a message to the host at the very same moment), the access is arbitrated in a round-robin fashion.

Table 12: Shared Interconnect address map.

Range	Purpose
`0x00000 - 0x0ffff`	Shared memory
`0x10000 - 0x1ffff`	HMQ slots (node CPU side, up to 15)
`0x20000 - 0x2ffff`	RMQ slots (up to 15)
`0x100000 - 0x1fffff`	Shared Peripherals port (1 MB)
`0x40000000 - 0x7fffffff`	Shared Peripherals port (2 GB)

Note: the SP port is mapped twice to consume less addresses in applications that have limited address space (e.g. VME).

3.7 Host Access Crossbar

The Host Access Crossbar (HAC) allows the host to access the CPU CSR and the host-side registers of the HMQ without disturbing the traffic going through the CPU LIs or the SI. Host access to the SI address space is also possible (round-robin arbitrated).

Table 13: Host Access Crossbar address map.

Range	Purpose
`0x00000 - 0x0ffff`	HMQ slots (host CPU side, up to 32)
`0x10000 - 0x1ffff`	CPU CSR
`0x200000 - 0x3fffff`	Shared Interconnect (small)
`0x80000000 - 0xffffffff`	Shared Interconnect (full)

3.8 Interrupt support

The WRN core itself only provides interrupts for the HMQs. If other interrupt sources are needed (for example, from the shared/dedicated peripherals), they may be connected to the common Vectored Interrupt Controller. The node driver API will provide appropriate functions for generic user space interrupt handling.

4 Software Interface

The WR Node software consists of a Linux device driver and a generic user space library. User applications are built on top of the library, there is no need for custom drivers (except for the cases which may require special interaction with the cores connected to the WRNC).

4.1 Device driver

The driver is based on the `fmc-bus` framework, enabling the WRNC to run on any FMC carrier supported by HT's driver kit. WRN cores are automatically enumerated through Software-Defined Bus (SDB) [6]. The driver exposes a `sysfs` directory for each WRN core for enumeration, initialization and debugging purposes.

Bulk communication between the driver and the user space library is implemented through a dedicated character device (one per each node) and `ioctl()`s.

4.2 Node API

This section provides a mock-up of the WRN library C API. The API may be modified during the development of the driver.

4.3 Initialization and node enumeration

```
int wrn_lib_init();
void wrn_lib_exit();
```

Initialization and cleanup of the WRN library.

```
int wrn_get_node_count();
```

Returns the number of WR Node Cores found in the system.

```
struct wrn_dev* wrn_open_by_lun( int lun );
struct wrn_dev* wrn_open_by_fmc ( int device_id );
struct wrn_dev* wrn_open( const char *device );
```

Opens the WRN with a given Logical Unit (1st case), attached to a given FMC device (2nd case) or through a direct character device path (3rd case).

```
uint32_t wrn_get_app_id( struct wrn_dev *device );
```

Returns the value of the `APP_ID` register of the given WRN device. May be used to automatically match the gateware with appropriate software built on top of the WRN library.

```
void wrn_close ( struct wrn_dev *dev );
```

Closes the connection with a WRN. Note that closing doesn't cause the node core to stop working.

4.4 CPU Control

```
int wrn_cpu_count( struct wrn_dev* );
int wrn_cpu_stop ( struct wrn_dev*, uint32_t mask );
int wrn_cpu_start ( sturct wrn_dev *, uint32_t mask );
int wrn_cpu_reset ( sturct wrn_dev *, uint32_t mask );
int wrn_cpu_load_application ( sturct wrn_dev *, int cpu, const void *code, size_t code_size );
```

Obvious functions for enumerating/starting/stopping/firmware loading for the CPUs.

```
int wrn_cpu_jtag_command ( sturct wrn_dev *, int cpu, int command, void *buffer, size_t buf_size );
```

Executes a raw JTAG command on a given CPU. Foreseen for implementation of a GDB remote debugging host. The command set and arguments are TBD.

4.5 Sending and Receiving Messages

```
int wrn_open_slot ( sturct wrn_dev *, int slot, int flags );
void wrn_close_slot ( sturct wrn_dev *, int slot );
```

Opens/closes a given HMQ slot and returns its file descriptor. The `flags` parameter is used to pass the "mode" in which the slot will be opened:

- `WRN_SLOT_INCOMING` opens an incoming slot (through which we will be sending messages to the CPUs)
- `WRN_SLOT_OUTGOING` opens an outgoing slot (through which we will be receiving messages from the CPUs)
- `WRN_SLOT_EXCLUSIVE` opens an incoming slot without sharing its traffic to other processes which have the same slot open.

The slot can be opened with both `WRN_SLOT_OUTGOING` and `WRN_SLOT_INCOMING` flags set. In such case, the returned file descriptor is bidirectional.

```
// set of filtering rules for wrn_bind() (reverse polish notation)
// - compare with a value (against a mask)
// - and/or/negate top of the stack

struct wrn_message_filter {

#define FILTER_OR 0
#define FILTER_AND 1
#define FILTER_COMPARE 2
#define FILTER_NOT 3

 struct rule {
   int op;           // operation
   int word_offset;  // which word to compare
   uint32_t mask;
   uint32_t value;
 } rules[];

 int n_rules;
};

int wrn_bind ( sturct wrn_dev *, int fd, struct wrn_message_filter *flt );
```

Binds an incoming slot to receive only messages matching a given pattern. The match rule is a set of comparison and Boolean operations, written for simplicity in Reverse Polish Notation.

```
int wrn_wait ( sturct wrn_dev *, int fd, int timeout_us );
```

Waits for a message to appear in the given fd, failing if the timeout has expired (-1 to wait forever). The combination of `wrn_bind()` and `wrn_wait()` can be used to build powerful even filtering and polling system in an user space application.

```
int wrn_recv ( sturct wrn_dev *, int fd, void *buffer, size_t buf_size, int timeout_us );
int wrn_send ( sturct wrn_dev *, int fd, void *buffer, size_t buf_size, int timeout_us );
```

Ordinary send/receive functions. Since the communication is file descriptor based, one can also use standard POSIX functions (`read()`, `select()`, etc.) to communicate with the WRN.

4.6 User space interrupts

```
int wrn_enable_irq ( sturct wrn_dev *, uint32_t irq_id, int enable );
```

Enables a given VIC interrupt (application-specific).

```
int wrn_wait_irq ( sturct wrn_dev *, uint32_t irq_id, int timeout_us );
```

Puts the calling process to sleep until a particular interrupt has arrived.

4.7 Accessing Shared Memory and other peripherals

```
#define SMEM_ATOMIC_ADD   0
#define SMEM_ATOMIC_SUB   1
#define SMEM_ATOMIC_BSET  2
#define SMEM_ATOMIC_BCLR  3
#define SMEM_ATOMIC_BFLIP 4

int wrn_smem_read ( sturct wrn_dev *, uint32_t addr, uint32_t *data);
int wrn_smem_write ( sturct wrn_dev *, uint32_t addr, uint32_t data);
int wrn_smem_op ( sturct wrn_dev *, int operation, uint32_t addr, uint32_t data);
```

API for atomic operations on the Shared Memory.

```
int wrn_raw_write ( sturct wrn_dev *, uint32_t addr, uint32_t data);
int wrn_raw_read ( sturct wrn_dev *, uint32_t addr, uint32_t *data);
```

Direct access to the FMC carrier's address space. Foreseen for debugging purposes. Use with caution.

5 Application Examples

5.1 LIST Node

The LIST Node HDL block diagram is shown in Figure 7. The two small crossbars between the WRNC and the FD/TDC Cores let the FD/TDC drivers access the mezzanines for initialization/management tasks. When the node CPUs are running the host never accesses these crossbars.

LIST Node configuration:

- CPU core 0: DP0 connected to Fine Delay FMC core (trigger output - pulse generator).
- CPU core 1: DP1 connected to TDC FMC core (trigger input - TDC).
- RMQ slot 0 (incoming only): received trigger messages (assigned to CPU 0).
- RMQ slot 1 (outgoing only): transmitted trigger messages (assigned to CPU 1).
- HMQ slot 0 (in/out): control commands for CPU 0.
- HMQ slot 1 (in/out): control commands for CPU 1.

- HMQ slot 2 (out only): logging channel for CPU 0 (received messages, timestamps of received, generated and missed pulses).
- HMQ slot 3 (out only): logging channel for CPU 1 (transmitted messages and timestamps of sent pulses).
- SMEM is not used.
- No additional interrupts.
- TU TAI counter used for network latency calculation and measuring task execution times.

5.2 Timing master

The general idea *(to be discussed)*:

- 2 CPU cores.
- CPU 0, executing the timing master tasks, with context switched, non-preemptive multitasking (the LM32's processing power is very likely sufficient for directly porting the existing Timing Master assembly tasks to C without any complex rework and true parallelism).
- CPU 1, responsible for less time-critical activities (loading event tables, etc.) and implementing the real-time functionality currently running on the MEN A20 CPU.
- TU configured to generate 1 ms UTC-synchronized ticks that drive the task scheduler.
- DDR memory connected to SP port, storing event tables and outgoing message log buffer.
- RMQ slot 0 (outgoing): timing traffic.
- HMQ slot 0 (in/out): general control.
- HMQ slot 1 (out): log of all generated timing messages.
- SMEM used for implementing the external conditions registers: triggering an external condition is done by atomic set/clear operations executed directly through Etherbone (bypassing the RMQ).

5.3 Timing receiver

The design aims to be a WR-based successor of the CTR. The current CTR design:

- Receives events and UTC-synchronous time base over an RS485-like link. The events are then searched against a big lookup table, which contains the configuration for the CTR's output counters that is executed if a matching event has been received.
- Logs received events and counter history in a large memory.
- Forwards *telegrams* (messages informing the system in advance about the machine status) to the software.

Given the granularity of current CERN timing system (1 ms time slot, up to 7 events per slot), these features can be as well implemented on a deterministic CPU. Below is a sample configuration:

- 1 CPU core, with DP0 connected to a CTR-like counter block. The CPU receives timing events through the RMQ slot 0, decodes them and looks up matching conditions in a locally-stored hash table.
- No need for a special PLL anymore, since we get the UTC/TAI timebase straight from WR.

- DDR memory connected to SP: event/counter log buffer.
- RMQ slot 0 (incoming): timing traffic.
- HMQ slot 0 (in/out): general control.
- HMQ slot 1 (out): timing events.
- HMQ slot 2 (out): telegram readout.
- HMQ slot 3 (out): timestamps of output pulses.
- dedicated interrupt lines wired to the CTR counter block (counter hit indication).

Note: bidirectional WR link operation to be discussed.

5.4 Distributed DDS Transmitter/Receiver

The D3S system [5] encodes an arbitrary RF frequency into UDP packets and reproduces it in any number of receiver nodes. The very brief WRN application for this project goes as follows:

- 1 CPU core, with DP0 connected to the DDS/phase detector core. The CPU runs the actual phase locking and data compression algorithm on baseband signal samples. Interpolation/decimation is done in dedicated HDL (part of the DDS core).
- RMQ slot 0: incoming (receiver)/outgoing (transmitter) RF data.
- HMQ slot 0: general purpose control.

References

[1] *LIST: LHC Instability Trigger Distribution System*, CERN, 2014, `https://wikis.cern.ch/display/HT/LHC+Instability+Trigger+Distribution`

[2] *Etherbone Full specifications*, GSI, 2013, `http://www.ohwr.org/documents/208`.

[3] *White Rabbit PTP Core*, CERN, 2013, `http://www.ohwr.org/documents/308`

[4] *`fmc-bus`: an FMC abstraction layer for the Linux kernel*, CERN, 2013, `http://www.ohwr.org/projects/fmc-bus`

[5] *Distributed Direct Digital Synthesis over White Rabbit (D3S)*, CERN, 2014, `http://www.ohwr.org/projects/wr-d3s`

[6] *Software Defined Bus specification*, CERN, 2013, `http://www.ohwr.org/documents/256`

[7] *Frontend Software Architecture framework (FESA)*, CERN, 2014, `http://project-fesa.web.cern.ch/project-fesa/`

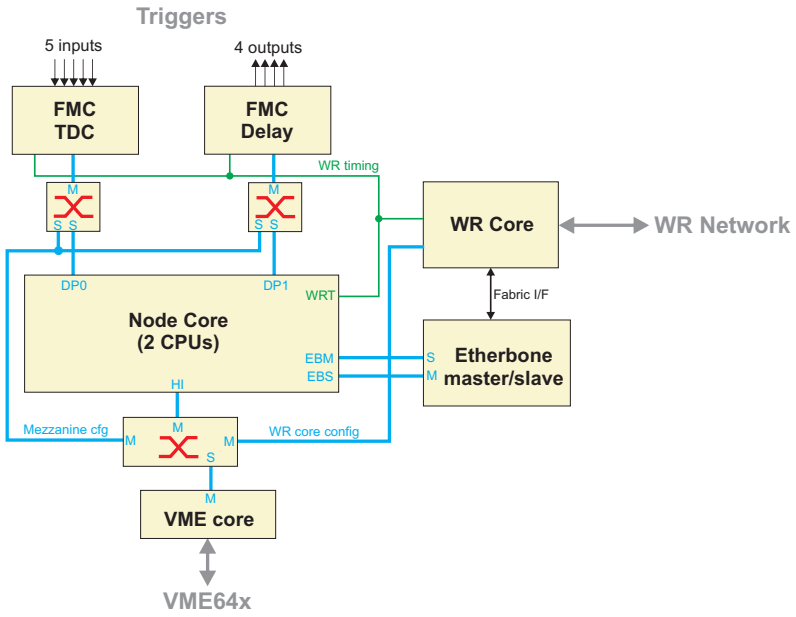

Figure 7: Conceptual HDL block diagram of a LIST node.