

09/2015

MaSoCist 'agathe' and 'anselm' reference

Martin Strubel

June 26, 2016

Revision:

v0.2-opensource

Overview

The MaSoCist is a SoC project based on a simple CPU core and peripheral IP cores, running on different architectures and FPGA families (Lattice, Xilinx, Altera on request). A short feature overview:

- IP core instantiation and SoC configuration via XML description
- Automatic synchronization of HDL and software
- Wishbone support
- Optional Co-Simulation support
- CPU and IP cores mostly open source

All other modules that are not listed in the Custom modules section are section5 IP supplied for evaluation only.

The MaSoCist environment is featured by a few number of CPU cores as shown in Table 1.1. The SoC configuration families have specific names, like agathe, beatrix, etc. Typically, the configuration is independent from the CPU core, however, the address map of the CPU core must correspond to the configuration, therefore the core variants may have specific extensions.

Name	Description
ZPU small core (Zealot)	OpenSource ZPU small, non pipelined
ZPUng	Proprietary, pipelined ZPU variant, configurable

Table 1.1: Default SoC core CPU options

Optionally, more complex but significantly faster cores can be supplied on request. These are customized MIPS cores for specific applications and require more FPGA hardware resources. Basically, the MaSoCist environment can be customized to use any other third party core with Wishbone support at possible sacrifices with respect to In Circuit Emulation debugging. The MaSoCist supports integration of third party, Wishbone compatible IP cores, such as the popular I2C core from opencores.org. These will need a specific description in the I/O peripheral map description.

This overview does not cover the details of SoC architecture implementations using the devdesc XML device description. Please refer to the generic netpp/SoC manual (available as separate support package).

1.1 Target applications

The MaSoCist does not intend to replace any of the middle end solutions provided by most FPGA vendors. It tries to match a niche where resources are sparse and where IP core configuration and debugging as well as optimum code density is more important than speed. Apart from the PyPS core, the system is not intended to run any major operating systems, but rather simple, bare metal code running with a somewhat deterministic sequence.

Reference applications:

- Smart video I/O processor for HDMI I/O and RGB/YUV conversion

- JPEG encoder with direct Bayer to YUV ('Cottonpicken' engine)
- Simple multi channel LED PWM controller (LED matrix)
- netpp communication stack on FPGA for IoT applications
- Special Coprocessor DSP solutions

A few more application scenarios are listed below:

The core is too slow for some operations, for example realtime PWM control

Enhance CPU core by special engine 'code window'

You need a multitasking OS to serve several tasks

Instance another core and set up shared memory or inter-core FIFO for communication

You are tired of hardware and software mismatches

Let configuration tools do the versioning and synchronization **automatically**.

You require increased security or you would like to prevent cloning

Lock up your intellectual property behind obscurity or even encryption

1.2 Licensing

To avoid too much complicated language, the licensing scheme is simplified in this paragraph. There are several licenses and other parties' rights involved in the MaSoCist:

1. MaSoCist components license
2. ZPU Opensource license
3. Third party (typically FPGA vendor) licenses, such as for generated IP cores

The licensing terms may turn out complex when further developments fall under different licenses, or may even collide with licenses of a third party module. The original MaSoCist module license rules, apart from the ZPU/Zealot licensing, are summarized as follows:

- You do have a license to simulate, build, and use the synthesized image on the hardware whose board supply image distribution you have received
- The software tools and scripts received with the MaSoCist are subject to dual licensing. For the OpenSource version, you may use the tools for other projects, but you will have to publish modifications and disclose their source. If you wish to use the tools in a completely different project that can not be opensourced, you will have to acquire a separate non-Opensource license.
- If you have received the opensource eval distribution (to be recognized by the '-opensource-\$VERSION' suffix of the distribution directory, all modifications you make must end up in the official distribution again
- If you received a non-opensource distribution, you are not allowed to make it OpenSource without signed consent of section5.

The dual licensing scheme does no longer apply, once a third party module using a license similar to a GNU Public License is introduced.



You are responsible for the licensing terms with regards to your end product. section5 will not take responsibility or liability for license violations caused by other parties

1.3 How to read this documentation

This SoC documentation is set up in a modular way, where parts may be shared by different architectures and SoC configurations. Therefore you may find references to other SoC setups. However, the register map listed in this document applies specifically to the configuration you have obtained. Since the same configuration can run on different platforms, there are platform specific sections listed below.

There may be reference to configuration variables like CONFIG_foo. These are not explicitly documented in this documentation, rather, the kconfig tool mentioned in Section 3.1 contains built-in help functionality about each documentation variable.

For quick system specific reference, you may want to use the following short links:

- Peripheral map of this SoC: Table B.1
- Platform specific memory and pin maps: Appendix A

1.4 CPU cores

Several CPU cores have been adapted to the MaSoCist.

1.4.1 ZPU small core 'Zealot'

The CPU core used in this SoC is based on the non-pipelined ZPU small design by Øyvind Harboe. It is enhanced by a In Circuit Emulation module (proprietary in house standard "StdTAP") for native JTAG debugging on FPGA hardware. The HDL for this CPU is OpenSource. This CPU variant is a full 32 bit CPU, not allowing unaligned access, therefore uses 32 bit addressing on a 4 byte for I/O access only. It is also enhanced by a small WishBone wrapper to access externally supplied IP cores (I2C, UART). The Zealot core has limited possibilities for customization.

The ZPU interrupts are non-reentrant, i.e. only one interrupt service at a time is processed.

1.4.2 ZPUng core

The ZPUng ('ng' standing for 'next generation') core is a pipelined proprietary redesign of the original ZPU architecture in the Python HDL extension **MyHDL**. A short overview:

- Instruction compatible to the ZPU small
- Three stage pipeline with shortcut mechanisms for some instruction sequences
- Separate Stack and Data/Program memory
- Typical native instruction duration: 1 cycle (up to 3 cycles on I/O and ALU operations)
- Enhanced interrupt, exception and event vector capabilities (ZPUng v1)
- Extended StdTAP debugging features
- Slightly higher logic usage than ZPU small
- Easily configureable and extendable using custom microcode extensions (ZPUng v2)

The ZPUng code is primarily available in VHDL under a custom source license. Verilog output can be supported on request.

The ZPUng v1 minimum configuration typically synthesizes in less than 800 LUTs (including debug logic), depending on architecture.

The ZPUng v2 is still under development and offers higher speed, however it breaks compatibility to ZPUng v1 with respect to microcode emulation features. Also it can be configured to use a different opcode set. Custom operations can be implemented such that some processing is occurring in parallel, but still within a callable function, accessible to an application programmer. The main ZPU machine code is still compatible to the classic ZPU variants. Interrupt and debugging functionality on the ZPUng v2 are yet limited.

1.4.3 PyPS core

The PyPS core is based on a classic MIPS architecture design with a few specific optimizations with regards to Block RAM. It runs up to 70 MHz on Spartan3 platforms ('cordula' configuration) with a BRAM register bank. Higher speeds can be achieved with distributed register RAM.

- MIPS-I instruction set with microcode enhancements (DIV/MUL)
- Five-Stage pipeline with hazard resolving unit
- StdTAP debugging interface
- Limited MIPS16 ASE option (no exception support)
- Configurable to be either fully compatible to MIPS-I or fast
- Limited interrupt support (use DMA where possible)
- DMA coprocessor option (two simultaneous memory channels)

1.5 Memory map and register architecture

The SoC variants agathe and anselm use a 16 bit wide addressing space. Configurations starting with 'b' have an extended address range.

The address map is typically split into a memory range and memory mapped register (MMR) for peripheral configuration, plus there is a range specifically reserved for the stack.

The precise mapping is depending on the used core configuration. Please refer to Appendix A for the detailed address map of your CPU variant.

1.5.1 Addressing

When accessing a memory mapped register, the address base of the MMR range must be added within the physical addressing routines. This is also regarded as "MMR offset" below. Note that the MMR space is a fully linear address space, indirect addressing is not used.

For the C header generation, the address offset calculation for each register is taken care of in the code. It should just be noted that there are up to four offsets specified for the effective I/O address calculation of a register:

1. The **MMR offset**
2. The **register map offset**
3. An **index offset** when several units of one map are instanced (e.g. UART0, UART1, ...)
4. Finally, the **register address offset**

Detailed address generation details are covered in the generic netpp/SoC manual.

1.5.2 Register architecture

Typically, a register map is defined per peripheral unit. Per register map, a register table can be emitted, such as Table 4.2.

In this SoC design, registers have the following access properties:

RO Read only, writing to this register has no effect unless a WO register is specified for the same address

WO Write only, reading of this register returns undefined values

RW Read/write, instanced as a real hardware register inside the MMR map

These properties are shown in the table under the `Access` header. For the documentation, they are rendered as shown in Fig. 1.1. The `MixedAccess` register type is not used in this architecture. The single bits are typically marked with a code, when relevant, denoting the default value after power up, or:

- x** Undefined, do not probe this bit for old HW revisions without checking for the `HWVersion` register
- s** Sticky status bit, clears when you write a '1' at this position ('W1C')
- Reserved bits, do not use for own purpose or alter when RW



A soft reset does not necessarily reset to the default values. See next section for details.

If the bit is not marked but named, it is:

- Not initialized in a R/W register
- Volatile in a Read-Only register



A special case is, when a RO and a WO register definition map to the same register address. This is the standard solution for a 'W1C' (Write one to clear) scenario. In this case, the WO register is not explicitly documented, rather, the 's' notation from above will be used on the RO register.

Registers also have a specific size, in this design up to the maximum possible atomic indirect addressable data width of 32 bits (4 bytes). The size of the register is listed in the `Span` property of the register table.

1.5.3 Register initialization

One important topic is the reset value initialization of registers with specified defaults. There are two implementation variants:

1. Register is initialized at FPGA boot time (default)
2. Register is initialized using a hard/soft reset (See `CONFIG_GENSOC_RESET_DEFAULTS`)

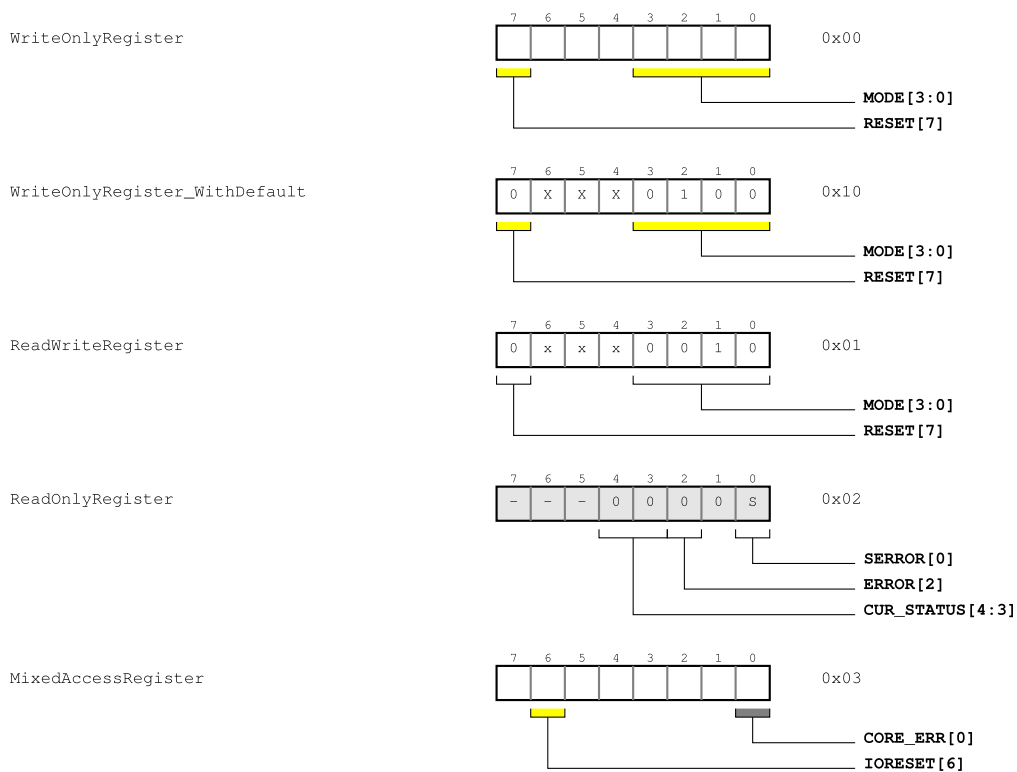


Figure 1.1: Register example legend

Depending on the FPGA architecture, the explicit reset pin initialization might be preferred. Registers that do not contain a default specification are not initialized, therefore their value can be undefined.

 Generally, it is considered good programming practise to only use defaults where necessary and initialize registers within the hardware driver routines.

Application examples

One example where defaults are necessary is the SIC_IMASK register. If it is undefined, spurious interrupts may trigger an IRQ while the event vector registers are yet uninitialized. The CPU will then jump into undefined code at startup, which can be tricky to debug. Likewise, it makes sense to initialize specific hardware enable bits for crucial peripherals to zero so that they do not consume power when the system resets, for example due to a watchdog event. In this case, it may be reasonable to enable `CONFIG_GENSOC_RESET_DEFAULTS`.

1.6 Register naming and namespaces

The gensoc translator utility generates source code from the device description XML following roughly this scheme:

- In C, Registers are prefixed by 'Reg_' and the register name. For example, the Magic register in the SysCtrl register map becomes `Reg_Magic` when the map prefix is not enabled (default).

- Likewise in VHDL, however, registers are prefixed by 'R_' and bit field definitions by 'B_'. Bit fields are always spelled in capitals.

Automatic conversion of XML to source is not covered in this documentation.

1.7 Firmware

The bare metal software running embedded on the SoC is considered to be the 'firmware', although this term might be used for the HDL as well. The firmware is basically the main program, converted into ROM such that it is starting immediately after the FPGA is powered up.

The firmware is built during the general synthesis 'make' procedure. Prior to building, the target must be configured. This is elaborated in Section 3.1.

1.7.1 ROM generation

The Makefile in the firmware directory creates a VHDL ROM instantiation for synthesis such that the program code is run at power up time just like when started from GDB (see Section 3.3.3). The Makefile evaluates a `SIMULATION=yes` option from the command line to optionally build for simulation. If the `SIMULATION` flag is set for the firmware that is built into the synthesized ROM, the system will not start up correctly and likely hang.



Make sure to fully rebuild the firmware (**make clean all**) before running synthesis to be sure everything is in sync.

More details about the ROM generation in Section 3.2.3.

Quickstart

2.1 Prerequisites

You need a development environment. If you have signed up for a board supply package agreement, you may have received a virtual machine image (Linux container or VirtualBox image file). In this case you do not need to worry about the prerequisites, because everything has been installed "ready to go". If you have obtained the tar file distribution, the whole package may not be complete. In this case you have to make sure that the following tools are installed on your system in order to run the simulation or generate code. The tools are typically found as a package for Debian and other systems with their corresponding name.

1. Linux kernel config 'kconfig'
2. GNU full native toolchain with make, gcc, cpp, etc.
3. Target toolchain, zpu-elf-gcc, mips-elf-gcc, etc.
4. python 2.x, IntelHex module
5. xsltproc

To compile and run the simulation, the following packages are required:

1. GHDL v0.30 or greater, GHDLex v0.051 or greater
2. netpp (/usr/lib/libslave.so)
3. gtkwave

2.2 Quick board selection

The supported board supply packages, 'out of the box':

- Papilio One platform
- MachXO2-7000 Breakout board

To configure for one of these boards, change your working directory to the MaSoCist top level dir, run

```
make papilio_config
```

or

```
make breakout_config
```

Then you are ready to rebuild the source code and HDL files.

For available custom or third party configurations, see vendor/\${VENDOR}/defconfig_* or check your custom SoC documentation.

2.3 Simulation

The entire SoC is simulated by GHDL with a few extensions to support virtual interfaces. Not all boards can be simulated by default, see below.

To build the simulation, GHDL and the GHDLex library must be present. Then you execute

```
make sim
```

in the MaSoCist directory. Possibly you will have to call **make -C sim clean** after a reconfiguration.

2.3.1 Running the simulation

Once the simulation was built correctly, the resulting `tb_<platform>` will be created in the `sim/` directory. Some configurations are based on a virtual UART instance, see `sim/virtualuart.vhdl`. This is started using the `init-pty.sh` script. Then you can connect to the running simulation using a terminal program, such as `minicom`:

```
minicom -o -D /tmp/virtualcom
```



When running inside the linux container (LXC), no manual setup using `init-pty.sh` is required. The virtual COM port is set up by the LXC.

Note that this UART is a full UART emulation, so the baud rate defined by the divider value passed to `virtualuart.vhdl` must match the UART peripherals baud rate, unlike the virtual console (`CONFIG_VIRTUAL_CONSOLE`), which does not depend on a baud rate. Finally, the simulation is run by starting the `tb_<platform>` executable from the command line. For interactive waveform tracing using GTKwave, there is a script `run.sh`, taking the platform name as argument, e.g.

```
./run.sh papilio
```

2.3.2 Advanced interactive simulation

Enter the `sim/` directory and type "make run". You should see the GTKwave window popping up, showing a slowly progressing wave display due to the enabled throttle. To disable the throttle, run the following command inside the virtual machine (or on your host)

netpp localhost TapThrottle 0

If you run `netpp` from your host to access the Virtual Machine or LXC (Linux Container), you need to determine the IP address of its virtual bridge interface (or what you have configured your VM with). For example, `ifconfig` on Linux may display:

```
vboxnet0 Link encap:Ethernet HWaddr 0a:00:27:00:00:00
          inet addr:192.168.56.1 Bcast:192.168.56.255 Mask:255.255.255.0
```

Then you simply access the simulation via

netpp 192.168.56.1 TapThrottle 0



This option only applies when the simulation is configured with a virtual TAP, like `CONFIG_VTAP`

2.3.3 Vendor specific simulation issues

Simulation of some boards may require libraries that are not included in the MaSoCist, because they are vendor specific. There may be several solutions:

- Obtain necessary files from your local FPGA tool installation and create a GHDL library. Use the `-P` option to GHDL to specify the search path to the GHDL config file.
- Try the `CONFIG_EMULATE_PLATFORM_IP` option
- Use a virtual board config that is fully vendor IP independent

For example, when you receive an error message like this:

```
../hdl/plat/breakout_top.vhdl:16:9: cannot find resource library "machxo2"
```

You have to create a file `lattice/machxo2-obj93.cf` somewhere using the rule:

```
MACHXO2_VHDL = $(wildcard $(LATTICE_SIM)/machxo2/src/*.vhd)
lattice/machxo2-obj93.cf: $(MACHXO2_VHDL)
    [ -e lattice ] || mkdir lattice
    ghdl -i --workdir=lattice --work=machxo2 $(MACHXO2_VHDL)
```

where `LATTICE_SIM` is the directory of your simulation VHDL files, like

```
/usr/local/diamond/3.1_x64/cae_library/simulation/vhdl/
```

Then set the `LIBGHDL` variable in `vendor/default/local_config.mk` to the directory where you created `lattice/machxo2-obj93.cf`, like:

```
LIBGHDL = $(HOME)/src/vhdl/lib/ghdl
```



For full 'model in the loop' simulation, it will be necessary to acquire an additional, non-free Simulation package by section5

2.4 Prepare synthesis

If you have configured something using the `menuconfig` utility, run either "make syn" from the top level or "make" inside the `syn/` directory to update your files. Then run your synthesis tool and open up the corresponding project file in `syn/<FPGA_VENDOR>/<PLATFORM>`, for example:

breakout

```
syn/lattice/breakout/breakout-opensource.ldf
```

papilio

```
syn/xilinx/papilio/zpu/zpu-opensource.xise
```

If the project files are not present for your platform, see below on how to quickly import the files required for synthesis.

2.4.1 Porting to new platform

When synthesizing for a new FPGA platform, you will have to create a new project first. The `MaSoCist` environment helps you with importing all the necessary files by creating TCL scripts for the supported synthesis tools (Xilinx ISE 13.4, Lattice Diamond 3.2 at this time).

1. Create the new project in `syn/<fpga_vendor>/<platform_name>/<project_name>`
2. To import the project files, open a TCL shell inside your IDE and run `source ../../proj_<platform>.tcl`
3. Run Synthesis to check if all files referenced are imported



FPGA platform specific IP cores may have to be manually imported into the project

2.5 Configure and build

For usage of the kconfig environment, see SoC specific documentation.

Basically, you execute the kconfig menu by running

make menuconfig

from the top level MaSoCist directory.

Before synthesis or simulation, it is required to rebuild the corresponding system files:

make syn



In the evaluation version (no gensoc included), the hardware peripherals can not be configured. You can only choose among supplied board configurations in `CONFIG_SOCDESC`.

2.6 Target download

2.6.1 Papilio

First, you have to set the `XILINX_ISE_DIR` variable in order to assemble the full firmware image for the SPI flash.

- Define `XILINX_ISE_DIR` in `vendor/opensource/local_config.mk`
- Export `XILINX_ISE_DIR` in `.bashrc` or alternative shell startup file

Example:

```
XILINX_ISE_DIR = /media/sandbox/Xilinx/13.4/ISE_DS
```

The Xilinx tools allow to recompile the software without the full ROM synthetization. See `syn/xilinx/papilio/Makefile` rules:

\$(PLATFORM)_fw.bit:

Build rule to merge program data with existing firmware bit file

download:

Downloads bare SRAM image into target

flash:

Merges extended memory (cacheable) space into overlay SPI flash image and programs it into the target

For download to the target, the **papilio-prog** application is required. See Papilio homepage for details.

The full firmware download is run by **make flash**. If no SPI code cache and program overlay is used, you can run **make download** instead. Otherwise, you have to make sure that all addresses are in sync.

2.6.2 MachXO2 Breakout

The MachXO2 variant requires a full synthetization of the generated HDL when the boot rom was altered. For download to the target, please use the Lattice Diamond Programmer.

Project files for target download exist:

breakout.xcf

Download into SRAM (volatile)

flash.xcf

Flash permanently onto the target

2.6.3 Other boards

All other boards such as proprietary / custom development can be programmed using an ICEbearPlus adapter, if the JTAG pins are accessible. Some boards may have embedded USB JTAG controllers. See Table 2.1 for details.

Board name	Programming	JTAG adapter
HDR60	Diamond Programmer	Integrated USB JTAG
gözcü	xc3sprog	ICEbearPlus
EFM01	Proprietary Cesium tool	ICEbearPlus (debug only)
Digilent Spartan-3 Board	xc3sprog	ICEbearPlus
denver2/3	xc3sprog	Embedded ICEbear

Table 2.1: Programming method

The Impact Cables server for ICEbear is no longer supported.

Software

3.1 Configure and build

The configure system is borrowed from the Linux kernel project and uses a `.config` file for the target configuration. The simulation build system fully relies on Gnu Make. Synthesis is partially (for specific platforms) supported by Makefiles as well. The Makefile structure of the SoC project is somewhat modular and described in more detail below.

3.1.1 Kconfig

The target configuration is done like in the Linux project, however, the `kconfig` frontend package needs to be installed. Then, the target can be configured by running the command **make menuconfig** from the top level `MaSoCist` directory.

When the `kconfig` package is set up correctly, a dialog should display as shown in Fig. 3.1. The configuration is saved in the file `.config` (again in the top level `MaSoCist` directory). This is a simple text file with typical configuration statements like `CONFIG_FOO=y`. To verify a configuration variable, the text file can be opened directly. It is also possible to manually edit the file and change variables, however care needs to be taken about the syntax and possible configuration variable dependencies. `kconfig` is able to fix most mistakes, so when the configuration turns out to be invalid, simply run `menuconfig` again.

Default configurations

The vendor specific directories (`vendor/$VENDOR`) contain default configurations for one or more board supply packages, recognized by the prefix `defconfig_`. To use this configuration, which is basically copying them over the local `.config`, type "**make <board>_config**" where `<board>` is among the supplied `defconfig_<board>` files.

To store a valid, changed configuration, type **make defconfig**. The make rule script will check for changes using the `diff` tool and ask you if you want to overwrite the previous configuration. When denying the overwrite, it will call an external `diff` tool (default: `meld`) to interactively display the changes and let you edit them.

Note that the `defconfig` rule will choose the default board name to generate a `defconfig_board` file in your default vendor directory. If you have different variants for one board, use the `CONFIG_PLAT_EXTENSION` variable to name your specific configuration variant.

3.1.2 Makefiles

Every subdirectory that contains configureable modules typically contains a Makefile which is included from a superior Makefile. Plus, several auxiliary make files with the `*.mk` suffix are included from the top level build Makefiles.

The list of supporting Makefiles:

config.mk

Contains system environment specific settings. Edit this file to configure your toolchain paths.

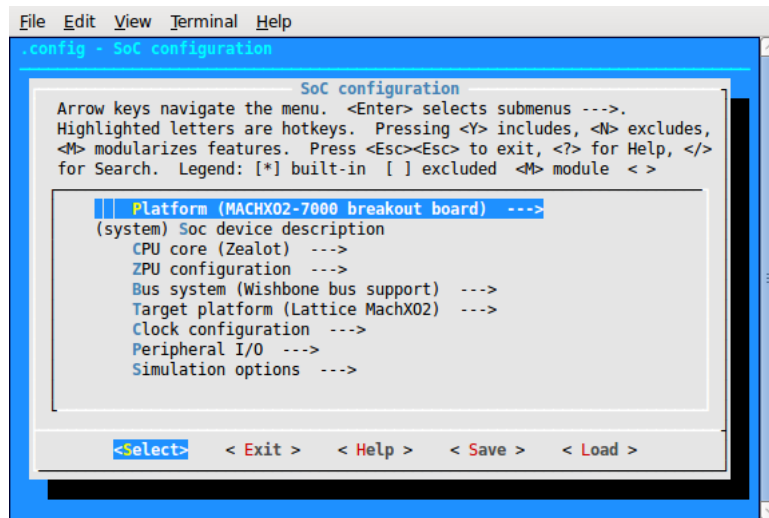


Figure 3.1: Typical menuconfig dialog

platform.mk

Contains platform specific configuration settings

gensoc.mk

Gensoc specific build rules

generate.mk

All rules for VHDL file generation

hdl/vhdlconfig.mk

Generates a VHDL configuration package from the Kconfig configuration

hdl/core/zpu.mk

ZPU core specific build rules

sim/libsim.mk

Simulation library specific build rules

syn/lattice/diamond_proj.mk

Makefile to create TCL helper scripts to import files to a Diamond project

syn/xilinx/ise_proj.mk

Makefile to create TCL helper scripts to import files to a ISE project

Some build processes depend on external tools whose location must be customized to the local machine settings. These settings are typically made in `vendor/$VENDOR/local_config.mk`. For example, to configure the diff tool to be the 'meld' application, you'll find a line like

```
DIFF = meld
```

Typical make rules

A make rule defines how things should be built and are executed by "make <rule>". They should be only executed from the top level or inside the `sim/` or `syn/` directory. All other Makefiles, especially those in lower directory hierarchies, require environment settings.

sim

Builds the top level simulation entity

syn

For supported platforms, the synthesis is run and a bit file is typically generated. If automated build is not supported, a TCL script is generated to ease the import into an existing project.

run

The simulation is run for a certain time. See `sim/Makefile` for the exact parameters and output files.

3.1.3 C preprocessing

The C preprocessor `cpp` is used for conditional code generation according to the `CONFIG` variables generated by the `kconfig` tool. This is also used for some VHDL files. The source files are given the `.chdl` extension. If you edit the generated VHDL files, keep in mind the changes will not be permanent, as they will be overwritten by the next build procedure. A warning in the file header will typically remind you again of this detail.

3.1.4 Large programs - function overlay

If the `CONFIG_SCACHE_INSN` option is present in the board supply package, larger programs than the actual block ram size allows can be created. Depending on the board configuration, functions may be placed into the extended program area by default, or a special attribute might be required to decorate the function. See also Section 4.1.2.

Which of these applies, is easily determined by the first letter of the board supply package name. If it begins with 'a', then you will need to use the decorator attribute. If it begins with 'c' or later, the linker script will put all non-decorated program code into extended memory. For details, please look at the generated map files or linker script of your board supply package. Also, the SoC specific section might describe implementation specific details.



Up to the ZPUng v1 architecture, program code in extended memory can not be single step debugged using ICE. When single stepping inside exception code, the debugger may freeze.

Linker scripting

Where a program section is placed, is determined by the linker and a memory map specified in the linker script that is specific to each board supply package and memory configuration. By default, code ends up in the `.text` segment. If the special attribute define `EXTERN_PROG` is used, code is placed in the `.ext.text` section. The linker script snippet below defines the placement of functions from this segment in the `xprog` section. Likewise, read-only data that is no program code is allocated into the `xdata` segment.

```

/* external cached data area: */
.ext.text      :
{
    _ext_program_memory = .;
    KEEP (*.ext.text)
} > xprog

. = 0x10000;
.ext.rodata    :
{
    _ext_data_memory = .;
    /* Put bitmap files into ext ROM */
    bm*.o(.rodata)
    (*.ext.rodata)
} > xdata

```

The sections are typically defined in a board layout specific separate linker script that is included or in the header of the active linker script. The section memory layout must specifically match the memory map of the SoC, as shown in the SoC specific memory map Section A.2. If the cache is not enabled at all, the `l1cache` area can be used for normal program/data storage. The snippet below shows an example MEMORY specification.

```

MEMORY
{
    l1ram(rwx): ORIGIN = 0x0000, LENGTH = 0x2000
    l1cache(rwx): ORIGIN = 0x2000, LENGTH = 0x2000
    xprog(r): ORIGIN = 0x08000, LENGTH = 0x8000
    xdata(r): ORIGIN = 0x20000, LENGTH = 0x20000
}

```

Program layout and timing issues

Note that loading from SPI flash takes some time. It is therefore recommended, to make sure that frequently used code is not spread across different pages, unless code in `l1ram` is called.



Currently, functions can not cross a page boundary. Make sure your functions are completely inside one cache page!

To examine the precise memory layout of the program, a map file is typically output during compilation. More advanced linker scripts may assert that no code inside a function can overlap the page boundaries.

Also important to know is that interrupt routines must not be in extended program memory, since the exception handlers are NON-reentrant. Even if reentrancy would be implemented, it would massively slow down execution due to frequent external memory access.

3.2 Firmware

The software running on the FPGA is named 'firmware' in this reference. It is cross compiled using the target toolchain within the general HDL build process.

3.2.1 Shell

The standard board supply packages are featured by a simple UART shell and a few test functions. Users wishing to test their hardware on a low level can do so easily by extending the `exec_cmd()` function of their board supply test code (naming scheme `test_${PLATFORM}.c`)

3.2.2 Register access

The firmware source code is written in C using a GNU cross compiler toolchain. Direct register access is supported through a few macros by using generated register definitions (`soc_register.h`). Programmers who wish to write their own drivers are encouraged to include the mid level auxiliary header `driver.h` and access registers as follows:

```
#include "driver.h"

...
// Set GPIO port 0 bits (14..15) to out:
MMR(Reg_GPIO_GPIO_OUT) = 0xc000;
```

Note that some CPU cores do not care about the effective register width, plus big endian and little endian (default) aspects become relevant. On the default MaSoCist SoC distribution, all registers are accessed using the `MMR()` macro. Other SoC variants using the PyPS core may use macros of the sort `MMR16()`, etc.

When accessing a register of width < 32 bits, the upper bits are undefined and can be garbage.



The most effective code is produced on the Zealot when accessing the bus using a 32 bit wide access. If you assign the result of an `MMR` query to a variable with higher width than the actual register you were reading, you have to mask out the undefined MSBs before comparison!

3.2.3 ROM file generation

The ROM builder utility `utils/buildrom.py` is a Python script converting from an ELF format into a set of VHDL files that can be used to generate a preinitialized DRAM emulating a Boot ROM. It currently supports ZPU and MIPS architectures. It relies on an open source `elf.py` module, also included in the `utils` directory. The generation rules are found in `generate.mk`. `buildrom.py` relies on the external 'intelhex' python module. You may have to obtain this module separately.

The Makefile will build two files from your program code:

rom_generated.vhd

The built in boot loader code, used to initialize the block RAM

flashdata.bin

The overlay program and data for the SPI flash user partition

The synthesis tool will typically build a `.bit` file from your design. An extra script will append the binary image `flashdata.bin` to this bit file which will be used to program the flash.

There are also options to use in-system upgrade functionality for a user program. See SoC specific section Appendix A whether these options are present.

3.3 gdb - the GNU debugger

The following sections describe how to access the SoC ZPU core through the GNU debugger for non-intrusive debugging (to the most possible extent on this architecture).

3.3.1 uniproxy - the SoC debug agent

The uniproxy debug server (former gdbproxy/rproxy) is a small debug agent to allow remote debugging of a target featured by StdTAP/JTAG. It listens on a TCP port (standard: 2000) for incoming connections from a GNU debugger. It is making use of the uniemu TAP library that supports several processor architectures through a uniform test access port implementation (StdTAP IP core). It also supports various JTAG TAPs of different FPGA manufacturers. It is typically started from a linux console as follows:

gdbproxy zpu

and responds with the following output upon successful detection of the SoC:

```
Remote proxy for GDB, v0.9.302, Copyright (C) 1999 Quality Quorum Inc.  
Blackfin target adaption (C) 2005-2012 and  
StdTap generic interface [ZPU, MIPS, plm32, FLiX]  
      (c) 2013 by www.section5.ch
```

```
notice: Found FTDI unbranded adapter device  
notice: Setting clock to 6 MHz (wait cycle: 0)  
IR size: 8  
Trying Lattice ECP3 35EA, got id c2048080  
Found Lattice ECP3 35EA  
Got user id (cmd 17): f77db57b, reverse: deadbeef  
notice: Detected 1 device(s)  
notice: Selecting CPU 0  
notice: gdbproxy: waiting on TCP port 2000
```

Now, connection to the target SoC can be made.



From the v1.0 package release on, the proxy is called 'uniproxy' to avoid conflicts with other gdbproxy distributions

If the device is not found, you might have to use a different port using the `--port` option.

Virtual gdbproxy

As part of a Co-Simulation extension to the SoC project, a special proxy version is supplied to connect to the running simulation via a virtual JTAG adapter. This allows to live-debug the simulated CPU and SoC over gdb.

3.3.2 Connecting to the Target

Start GDB using the command `zpu-elf-gdb` and provide the firmware executable of the SoC as argument, for example:

```
zpu-elf-gdb main
```

Then make a target connection using

```
(gdb) target remote :2000
```

Once connected, gdb shows the current position the program was interrupted in, for example:

```
0x0000030b in _neqbranch ()  
(gdb)
```

The program can then be resumed using the **continue** command and interrupted again by using **Ctrl-C**.

3.3.3 Program download

To load the program onto the target at runtime, it is important to know that the SoC core is a stack machine and requires a valid stack pointer address in order to operate correctly by In Circuit Emulation. Therefore the core must be reset using **monitor reset** before download. It is no longer required to explicitly set the stack pointer.

```
define init
    monitor reset
    load
end
```

To load the current main program, only type **init**:

```
Loading section .fixed_vectors, size 0x400 lma 0x0
Loading section .init, size 0x7 lma 0x400
Loading section .text, size 0x16e8 lma 0x407
...
Start address 0x0, load size 8857
Transfer rate: 11809 bits/sec, 98 bytes/write.
```

For all other gdb commands, please type **help** or use the various gdb documentation resources from the internet.



monitor reset may not fully reset all logic, depending on your peripheral configuration. If peripheral logic prevents the CPU from booting correctly, like for example interrupts, you may have implement further reset scripts to restore default register values. See Section 1.5.3 for specific reset initialization options.

When downloading a large program using function overlays the jump addresses may have changed. Therefore, functions that call code from non-updated ROM will likely crash or produce unexpected results. When testing programs via RAM downloads, make sure your ROM is up to date or not called (CONFIG_SCACHE_INSN disabled). Another possibility is to avoid relative jump addresses by using the `--mno-callpcrel` flag during compilation (CFLAGS variable in the Makefile).

If the startup procedure of your program code is not depending on any global variables that can be altered, you can use **monitor reset** to restart. If your program may have been altered however, be it a malfunction or a feature, you will have to reload it into the block RAM.

3.3.4 Debugging and interactive register dumps

Microcode debugging

The ZPU 'small' core used in this SoC provides an emulation feature for a few assembly instructions that are not implemented in hardware. Therefore, when interrupting the program, the CPU may be stuck in the emulation routines outside the firmware program. In this case, the backtrace (**bt**) may not show the correct stack trace. To return to a CPU state where the trace is valid, use the **next** command (or just **n**) to continue until exit:

```
(gdb) n
Single stepping until exit from function _neqbranch,
which has no line number information.
0x00000577 in delay (i=32620) at main.c:59
59          while (j--) { asm("nop"); }
```

It is recommended to use breakpoints as far as possible to stop the program at specific points. If fully atomic assembly debugging of a program is required, there are several options:

- Use ZPU variant with full hardware instruction implementation
- Use ZPUng with emulation masking option (EMUMASK bit set, this is the default operation). This feature is not documented.



On the ZPUng, microcode debugging features vary from different versions and are not meant to be used by an application programmer.

The uniproxy debug agent automatically detects the used CPU type via JTAG, thus no further configuration by the user should be required.

Local variables

Typically, when breaking inside a function, it will display the local variables. On the ZPU/Zealot default platform, these variables may not be correct or not be displayed at all if optimization flags are used or if the `--speed=0` option is given to uniproxy (disables pseudo register stack reads). For critical applications that require repeated dumps of variables, it is recommended to use a global variable in memory. The reason for this behaviour is that the ZPU architecture is a stack machine and full context tracking is limited on the current GCC toolchain.

Interactive register inspection

For interactive register inspection, gensoc creates a GDB auxiliary script `soc_mmr.gdb` for the entire SoC register map. This provides a quick method to dump the bit fields or the value of a register.

For example, a verbose dump of the IRQ SIC_ILAT register is executed by the following command sequence:

```
(gdb) source soc_mmr.gdb
(gdb) dump_irq_sic_ilat 0
```

```
Address: 00008304 SIC_ILAT: 00000080
```

The '0' given after the dump call is the device index, in case there are several instances of a device controller. Single device units do not require the '0' on single instance devices.



The script does not explicitly check the bit width of the register being dumped. The MSBs may contain garbage when accessing registers with width < 32 bits.

The `.gdbinit` startup file of the firmware (see `sw/` folder) contains various debug scripts to obtain the current register status. Likewise, manual configuration of hardware registers can take place. For example, the following code sets a GPIO output:

```
source soc_mmr.gdb

define initLED
    set *$Reg_GPIO_GPIO_OUT = 0x8000
    set *$Reg_GPIO_GPIO_DIR = 0x8000
end
```

Peripheral modules

The SoC peripherals are mapped automatically from the device description. Several instances of peripheral modules, such as GPIO controller banks, can be mapped into the MMR address space according to the addressing scheme defined in a specific unit map. For register reference, the absolute address of a register is calculated as the sum of the following values:

- A registermap offset per device unit, listed in Table B.1.
- A register offset, listed in the peripheral address map

For external Wishbone peripheral IP that is not auto-generated from the device description, an address relative to the MMR_OFFSET range applies, thus requiring none of the above calculations.

The register map reference is valid only for the following hardware revision of the SoC device:

Device Revision: 0.3[beta]



Use the 'sysinfo' GDB script to obtain the current HW revision from the SoC (HWVersion register)

Each component has some software support, or optional regression test scripts to test the interface. See Table 4.1 for an overview on source files corresponding to a peripheral.

Description	C source	Regtest script
General tests and board initialization	test_\$(PLATFORM).c	-
System interrupt controller	irqhandler.c	sw/regtest-common/sic.gdb
Timer and PWM	pwm.c	sw/regtest-common/pwm.gdb
Simple UART	uart.c	sw/regtest-common/uart.gdb

Table 4.1: Peripheral module software support

4.1 System control

The system control section covers the platform specific configuration properties. To synchronize the Firmware with the hardware, the HWVersion register should be queried. Typically, this revision code is generated from the system device description XML. One might note that the SysCtrl map follows a different naming scheme than the other agathe peripherals. The reason for this is, that the SysCtrl unit is specific to the system architecture, whereas the basic peripherals are per se system agnostic. For example, if a system supports I/O multiplexing (GPIO and alternate functions) or cache functionality, the SysCtrl unit becomes relevant.



All MaSoCist SoC instances must have at least a SysCtrl unit with the XML id 'sys'.

More complex systems may have a separate I/O multiplexing unit.

Offset [Span]	Name(Id)	Access	Description
0x00 [2]	SysCtrl	RW	System control register
0x04 [4]	SysConfig	RW	System config register [UNUSED in this configuration]
0x08 [2]	SysStatDebug	RO	System state debug register. Not documented in this configuration.
0x0c [1]	SysConsole_W	WO	System Console output register. Normally for simulation/JTAG debugging only.
0x0c [2]	SysConsole_R	RO	System Console Input register. Normally for simulation/JTAG debugging only.
0x14 [4]	Magic	RO	Magic number (typically CPU ID). Used for detection.
0x18 [2]	SocInfo	RO	System type identification field
0x1c [4]	HWVersion	RO	Hardware revision word. Reserved for Simulation: 0xff01

Table 4.2: Address map SysCtrl



In order to facilitate downward compatibility with respect to firmware, it is mandatory for the HWVersion register to remain at the end of the register map. Since the address bits are masked according to the MMR_CFG_SysCtrl definition in the system map VHDL file, it is ensured that queries with higher MSBs set (that are masked out on the older HW revision) still receive the correct HWVersion.

4.1.1 Clock configuration

The agathe configuration has a simple clock scheme for the peripherals. The main system clock is simply divided by a divisor value to obtain the clock for the peripheral. The system clock itself depends on the FPGA hardware and oscillator frequencies used. Some platforms do have a PLL, multiplying the external master clock by a fractional value. The configuration of clock frequencies and dividers is done by the menu configuration. The formula to obtain the system clock (see also `sw/driver.h`):

$$SYSCLK = CLOCK_FREQUENCY * \frac{CONFIG_PLL_MUL}{CONFIG_PLL_DIV} \quad (4.1)$$



You can output the current system clock in Hz within the GNU debugger by `print g_sysclk` when using the standard test firmware.

For platforms without PLL and missing CONFIG_PLL variables you can assume DIV and MUL = 1. Some FPGAs have internal oscillators whose frequencies may not be arbitrary. These have their own CONFIG variables. Handle with care, do not reconfigure those clocks freely or your system may stop to operate correctly.

The software library support coming with your board supply package typically takes care of the clock configuration.

4.1.2 SCache

The SCache is a simple software cache implementation. It is not present in the agathe configuration. This is a stub section.

4.2 Interrupt handling

4.2.1 System interrupt controller

The system interrupt controller (SIC) is a very simple IRQ controller handling up to four channels using the registers listed in Table 4.3

Up to four interrupt sources can be mapped to the channels with a simple prioritized IRQ vector generation.

Offset [Span]	Name(Id)	Access	Description
0x00 [1]	SIC_IMASK	RW	System interrupt mask. When '1', interrupt is enabled
0x04 [1]	SIC_ILAT	RO	IRQ latch register. When IRQ was just received, the corresponding bit is set
0x04 [1]	SIC_ILAT_W1C	WO	When IRQ is latched, clear the corresponding ILAT flag by writing a '1' to this register
0x08 [1]	SIC_IPEND	RO	IRQ pending register. When enabled in SIC_IMASK, a latched IRQ is marked '1' in this register when pending.
0x08 [1]	SIC_IPEND_W1C	WO	When IRQ is pending, clear the corresponding IPEND flag by writing a '1' to this register
0x0c [1]	SIC_IAR	RW	Interrupt assignment register
0x10 [2]	SIC_IV0	RW	Interrupt vector 0
0x14 [2]	SIC_IV1	RW	Interrupt vector 1
0x18 [2]	SIC_IV2	RW	Interrupt vector 2
0x1c [2]	SIC_IV3	RW	Interrupt vector 3
0x20 [2]	SIC_EV0	RW	Exception Vector 0

Table 4.3: Address map IRQ

SICv2 exceptions

The v2 of the SIC has an extra NMI pin plus an EV0 exception vector register. When the NMI pin is asserted, this overrides all pending interrupts and sends an interrupt signal to the CPU. No other interrupts can trigger an event until the condition causing the NMI to be set is cleared. The cause of the exception is not known to the SIC. Instead, it is up to the CPU designer to implement exception cause registers in the SysCtrl map.

Since exceptions are non-reentrant, the programmer must ensure that no exception occurs inside an exception handler. Likewise, it can be problematic if exceptions occur inside interrupt routines that can interrupt a running exception handler. If the exception handler clears the exception cause too early, interrupts causing another exception will likely trash the stack.

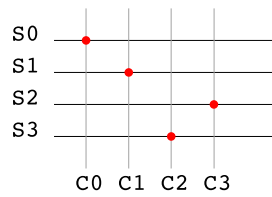


Figure 4.2: Channel assignment scheme

Interrupt handling

When a '1' level occurs on an interrupt source, the corresponding bit in the SIC_ILAT register is set. The corresponding bit in the SIC_IMASK register determines, whether the interrupt triggers an IRQ request to the core. If the SIC_IMASK bit is set, the corresponding SIC_ILAT bit will be latched into SIC_IPEND and the CPU will receive a pulse on the IRQ line. The interrupt vector is generated by the SIC according to the SIC_IAR register.

By default, each interrupt source [0..n-1] is assigned to the corresponding channel [0..4-1] where n is the maximum number of channels (four in this fixed implementation). Channels with lower numbers have higher priority. The channel assignment can be changed by setting the corresponding bit field in the SIC_IAR register.

The user must initialize the interrupt vectors in SIC_IVx to the IRQ handlers before enabling the interrupts using the SIC_IMASK register.

Channel assignment

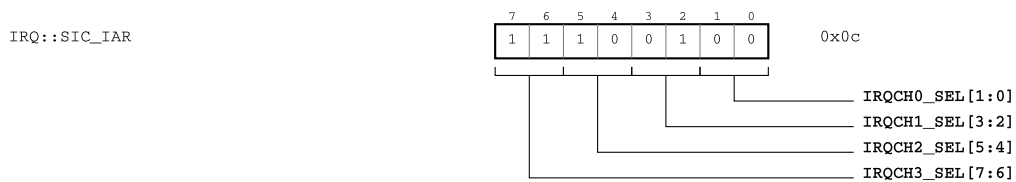


Figure 4.1: SIC_IAR register

Fig. 4.2 shows the connection scheme used by the SIC_IAR register. S[0..3] denote the IRQ sources, C[0..3] the IRQ channels. If S0 is supposed to trigger an IRQ on lowest priority C3, a value of 3 needs to be written to the SIC_IRQCH0_SEL bit field.

If several interrupt sources are assigned to the same channel, they are technically ORed. The developer needs to determine the event source according to the SIC_ILAT register in the irq handler corresponding with the assigned channel.



With the Zealot CPU core, the SIC_IVx settings will be ignored, due to only one global interrupt routine support (zpu_interrupt()). The zpu_interrupt routine will have to determine the IRQ cause by checking the SIC_IPEND register.

Priorities

Simultaneous interrupt events are always won by the higher priority channel. If an interrupt is pending, any other interrupt event will set the flag in the IPEND register (if not masked), however **only** a higher priority event will signal the CPU that another interrupt is to be serviced via the override pin. Once the corresponding IPEND flag is cleared, the next set flag with the

highest priority will cause the irq line to the CPU held high and assert its corresponding interrupt vector to the event vector port to the CPU.

4.2.2 Interrupt source mapping

Currently, only four interrupt channels are used. The mapping is platform specific, see Appendix A.

4.2.3 Core specific interrupt handling



This section applies to the 'ZPUng' CPU architecture only

Typically, an IRQ routine is a just a normal C subroutine with no specific interrupt attributes. However, this only is valid when no memreg instantiations are used inside the IRQ handler. If this is the case, `restore()` and `save()` macros need to be called inside the interrupt routine. Whenever an interrupt is serviced from the handler, it must signal to the SIC by clearing the IPEND bit to the corresponding IRQ channel that the interrupt logic can rearm for the next event to happen on this channel.

The ZPU legacy architecture does not have a protection against reentrancy of an interrupt handler on recurring events, so too frequent interrupt events could trash the stack. The ZPUng logic prevents this by internal usage of an IRQ flag. While this flag is set, no other IRQ source can interrupt a current handler. The flag is always cleared at the end of the interrupt handler, more details below.

Interrupt priorities

Interrupt priorities are sorted out by the System Interrupt Controller (SIC). By default, all SIC implementations handle lower channel numbers with higher priority such that:

- Simultaneous events on two channels will trigger the higher priority event only
- Higher priority events will cause an override action (more below)
- Lower priority events while another interrupt is in service will queue and execute after the current interrupt handler exits

The SIC will assert the override signal if it detects an event with higher priority than the one currently latched. The override signal clears the internal IRQ flag in the ZPUng core and therefore allows reentrancy.

Exceptions

Exceptions are also handled like an interrupt in the ZPUng, however, they only occur on special occasions like memory accesses. An exception can be assigned to any property, however a higher priority IRQ can interrupt the exception handler **only after** the cause for the exception has been cleared. If the exception cause clearing is forgotten inside the exception handler, the system will hang on the next memory access.



Exceptions are not used in the opensource 'agathe' configuration

Writing handlers

Interrupt routines on systems that allow interrupt handler reentrancy must clear the internal interrupt flag before the IRQ routine is left. On the ZPUng v1, the `IRQ_REARM()` macro must be called at the end of the irq handler. Below is a standard example with the v0.1 toolchain release.

```
void irq_generic_handler(void)
{
    MMR(Reg_SIC_IPEND_W1C) = PINMAP_IRQ_GPIO_A; // clear IRQ and rearm
    g_counter++;
    IRQ_REARM();
}
```



With the v0.2 toolchain and ZPUng v1.1, the `IRQ_REARM()` macro is no longer required, instead, the `__attribute__((interrupt))` decorator must be used as preamble to the function declaration.

4.3 GPIO module

Simple general purpose I/O controller. The available registers are shown in Table 4.4. The GPIO bank has a very simple functionality. Configure this bank in your firmware as follows:

1. Choose direction (input or output) using the `GPIO_DIR` register
2. Read a value using the `GPIO_IN` register, write via `GPIO_OUT` register
3. Atomically set GPIO outputs high using the `GPIO_SET` register (write only)
4. Atomically clear GPIO outputs using the `GPIO_CLR` register (write only)

Offset [Span]	Name(Id)	Access	Description
0x00 [2]	GPIO_DIR	RW	IO direction register. '1': Output, '0': input
0x04 [2]	GPIO_IN	RO	GPIO input (read) value
0x08 [2]	GPIO_OUT	RW	GPIO output value
0x0c [2]	GPIO_SET	WO	Write 1 to set
0x10 [2]	GPIO_CLR	WO	Write 1 to clear

Table 4.4: Address map GPIO

4.4 The TIMER and PWM module

The Timer module implements a global timer control plus several instances of PWM units (up to 16 per Timer unit). There are different variants:

PWMSimple

Simple PWM implementation with only one counter mode

PWMPlus

More complex PWM with extended counter modes and clock sourcing options

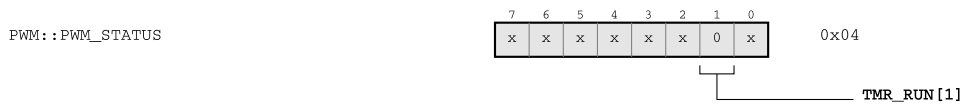


Figure 4.4: PWM_STATUS register

4.4.1 Timer configuration

The timer module contains the configuration that apply to all PWMs at the same time. It derives the PWM clock from the incoming system clock which is divided by the the PWMCLKDIV register. This contains a divisor value minus one. The typical setup procedure follows this scheme:

1. Configure PWMCLKDIV
2. Configure PWM units needed
3. Start PWMs using the TIMER_START register

A write to the TIMER_START register starts the PWM unit whose corresponding bit is set. Likewise, a write to TIMER_STOP stops the corresponding PWMs.

Offset [Span]	Name(Id)	Access	Description
0x00 [2]	TIMER_START	WO	Write a '1' for each timer unit to be started.
0x04 [2]	TIMER_STOP	WO	Write a '1' for each timer unit to be stopped.
0x08 [2]	TIMER_STATUS	RO	Timer status register (reserved)
0x0c [2]	TIMER_CONFIG	RW	Timer config register

Table 4.5: Address map TIMER

4.4.2 PWM configuration

Each PWM unit has a corresponding register map as shown below. The PWM_WIDTH register defines the width of the full PWM interval. The PWM_PERIOD value determines when the timer event occurs and the PWM output level is toggled.

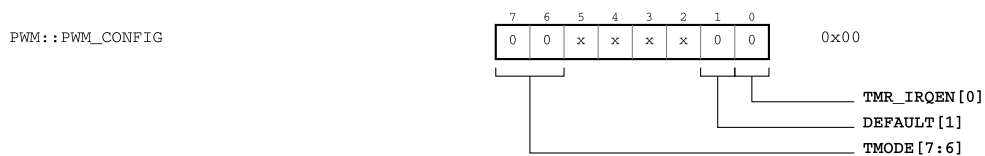


Figure 4.3: PWM_CONFIG register

The TMODE field is ignored by the PWMSimple module and reserved for other PWM implementations. By default, initialize TMODE with 0. When the TMR_IRQEN bit is set, the PWM will trigger an internal timer interrupt event when the PWM counter has reached the PWM_PERIOD value.

When the TIMER_CONFIG::CRESET bit is set, a write to the TIMER_STATUS register will cause the internal clock divider counter to be reset to zero. This ensures that changes to TIMER_CONFIG::PWMCLKDIV are in immediate effect once the timers are started, plus the latency from the start event to the next rising PWM edge is deterministic.

Offset [Span]	Name(Id)	Access	Description
0x00 [1]	PWM_CONFIG	RW	PWM configuration register
0x04 [1]	PWM_STATUS	RO	PWM status
0x08 [2]	PWM_WIDTH	RW	The PWM interval length minus one. When the counter reaches this value, it resets to zero.
0x0c [2]	PWM_PERIOD	RW	The PWM period value (<= PWM_WIDTH) minus one
0x10 [2]	PWM_COUNTER	RO	Current PWM counter value. Volatile.

Table 4.6: Address map PWM

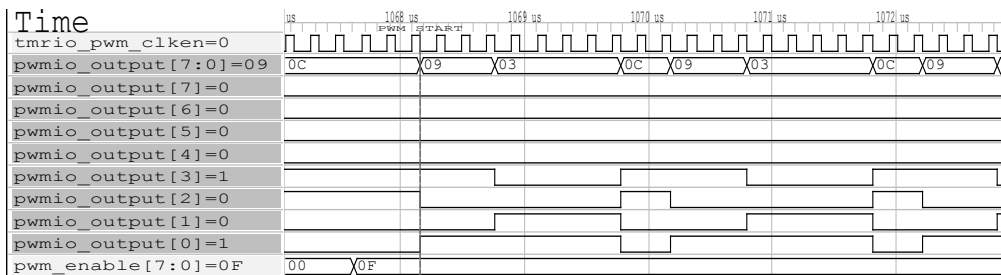


Figure 4.5: PWM example

4.4.3 Example

Fig. 4.5 shows an example configuration of four simultaneously started PWMs. Channel 2 and 3 have the DEFAULT bit set, causing an inverted signal with respect to channel 0 and 1. The configuration values for the channels are listed in Table 4.7.

When the corresponding TIMER_START bit is set (pwm_enable[] signals), a PWM channel starts operating on the next falling edge of the internally generated clock enable signal (which is global for all PWM units in the agathe configuration). Changing the PWMCLKDIV value while the timers are running produces nondeterministic results.

Channel	Values
0	PWM_WIDTH=10-1, PWM_PERIOD=2-1, DEFAULT=0
1	PWM_WIDTH=10-1, PWM_PERIOD=5-1, DEFAULT=0
2	PWM_WIDTH=10-1, PWM_PERIOD=2-1, DEFAULT=1
3	PWM_WIDTH=10-1, PWM_PERIOD=5-1, DEFAULT=1

Table 4.7: Example configuration

4.5 Simple UART

The UARTSimple module implements a primitive UART with limited functionality:

- 8 bit frame size only, one start, one stop bit
- No parity support

- No break detection
- No handshaking

The UART is configured by setting the UART_CONTROL::UART_CLKDIV bitfield to a value of $\text{sysclk}/16/\text{baudrate} - 1$. Whenever the clock is changed, the (UART_CONTROL::UART_RESET) bit needs to be set and released as well for proper clock operation.

When the RX_IRQ_ENABLE bit is set, UART receive events trigger an IRQ on the UART irq line. See Appendix A how this IRQ pin corresponds to the SIC interrupt channels.

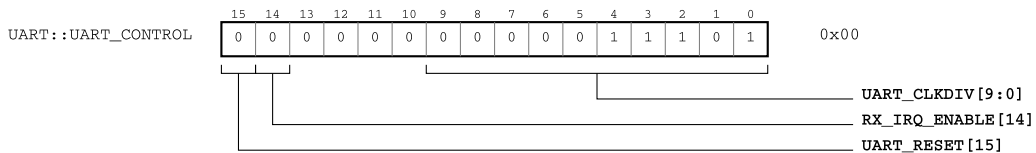


Figure 4.6: UART control register

For simple operation, the RXREADY/TXREADY bits of the UART_STATUS are polled before writing to UART_RXR and UART_TXR, respectively. Error bits are reset upon issue of a UART_CONTROL::UART_RESET toggle. The meaning of the status/error bits is as follows:

TXBUSY

When set, characters are left to transmit in the FIFO queue. However, you can keep writing data as long as TXREADY is asserted high

FRERR

When set, a framing error occurred (less than 8 bits transmitted, or glitch)

TXOVR

The transmit FIFO was overrun, i.e. a write occurred although TXREADY is low

RXOVR

The receive FIFO was overrun, because the UART_RXR register was not read in time



The non-pipelined Zealot CPU may have difficulties keeping up with high baud rates. There is no guarantee of a maximum baud rate and depends on the deployed system configuration and software

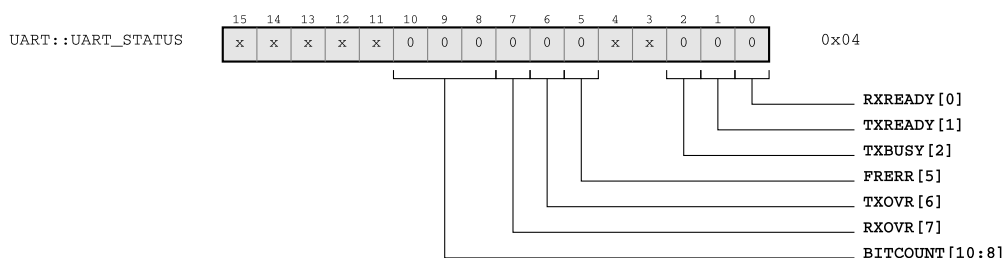


Figure 4.7: UART Status register

Offset [Span]	Name(Id)	Access	Description
0x00 [2]	UART_CONTROL	RW	UART Control register
0x04 [2]	UART_STATUS	RO	UART status register
0x08 [2]	UART_RXR	RO	UART receiver register
0x08 [1]	UART_TXR	WO	UART transmitter register

Table 4.8: Address map UART

4.6 SPI Master

This module implements a simple SPI master controller with the register map shown in Table 4.9.

The SPI master can be configured to use a RX/TX register width up to 32 bits. This upper limit is defined by CONFIG_SPI_BITS_POWER. See below on bit width configuration.

Offset [Span]	Name(Id)	Access	Description
0x00 [2]	SPI_CONTROL	RW	SPI control register
0x04 [1]	SPI_STATUS	RO	SPI status register
0x08 [1]	SPI_TX	WO	SPI transmit register
0x08 [1]	SPI_RX	RO	SPI receive register
0x0c [2]	SPI_CLKDIV	RW	SPI clock divider register

Table 4.9: Address map SPI

First, the SPI_CLKDIV register should be set with the desired clock divider. Again, the resulting SPI clock is the system clock divided by 4 divided by (SPI_CLKDIV-1). When the PUMP bit is set, a write to SPI_TX or read from SPI_RX will trigger a SPI send/receive cycle.

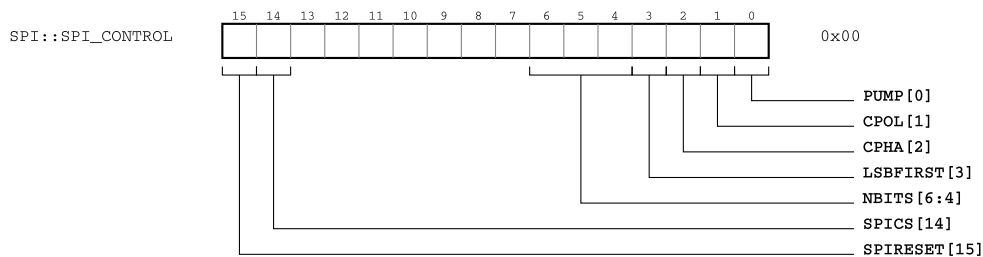


Figure 4.8: SPI_CONTROL register

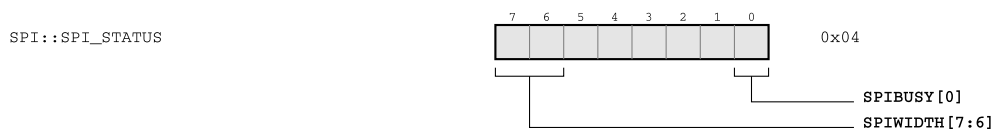


Figure 4.9: SPI_STATUS register

During a RX/TX cycle, the SPIBUSY bit is set. A driver routine has to poll this bit before issuing the next command.

When doing simultaneous transfers, the user will have to turn off the PUMP bit for either of the SPI_TX or SPI_RX access. More advanced peripherals as the UniSI core offer a few more configuration options and DMA support for improved throughput. Other configuration bits:

CPOL

Invert clock polarity

CPHA

When set, assert first bit on first clock cycle

LSBFIRST

When set, transmit LSB first

NBITS

Number of bits to transmit minus one

SPIRESET

Toggle to reset the SPI controller

4.6.1 Bit width configurations

The SPI peripheral can be hard-configured for a maximum transfer width of 8, 16 and 32 bits using the CONFIG_SPI_BITS_POWER variable. When the PUMP bit is set and a value to the SPI_TX register is written, the controller will start transferring the number of bits specified by the NBITS field, starting with the MSB by default (LSBFIRST = 0). Even if the number of transferred bits is lower than the maximum register width, the data has to be MSB-aligned. For example, assuming you have configured the maximum width to 32 bits (CONFIG_SPI_BITS_POWER = 5) and you need to write a byte value of 0xaa, you have to set NBITS to 7 and SPI_TX to 0xaa000000. It is recommended to drive the SPI controller in full 32 bit mode on the ZPU small architecture for greater speed.

If you have LSBFIRST set, the data is LSB-aligned, i.e. if you are writing a burst sequence 0xaa, 0xbb, 0xcc of 8 bits each, you have to write 0x00ccbbaa to the SPI_TX register and set NBITS to 23.

Typically, the software should be configured to match the hardware configuration. If unsure or if you need to have a generic driver routine that determines the SPI bit width at runtime, you need to evaluate the SPI_STATUS::SPIWIDTH bit field.

Custom SoC documentation

A.1 OpenSource/eval distribution

The OpenSource distribution for evaluation has limited documentation on the platforms. System specifics (such as pin mappings) are listed here.

A.1.1 Breakout-Board (MACHXO2)

IRQ mapping

Table A.1 shows the pin mapping for this platform.

A.1.2 Papilio (Spartan3)

Two setups are supported for this system:

agathe

Basic peripheral configuration with Zealot CPU

beatrice

Advanced peripherals (LCD driver, Cache) with ZPUng CPU (default)

. The tables display the IRQ mapping (as a pseudo register, where the bits correspond to the SIC IRQ channels)

'agathe' setup

Bit(s)	Name	Description
3:1	PINMAP_IRQ_TIMER	TIMER IRQ
0	PINMAP_IRQ_UART0	UART0 IRQ

Table A.1: *IRQ_DEFAULT_MAP* register IRQ default pin map

'beatrice' setup

Bit(s)	Name	Description
3	PINMAP_IRQ_GPIO_B	GPIO Bank B interrupt
2	PINMAP_IRQ_GPIO_A	GPIO Bank A interrupt
1	PINMAP_IRQ_TIMER_A	TIMER interrupt
0	PINMAP_IRQ_SCACHE	SCache interrupt

Table A.2: *IRQ_SCACHE* register IRQ pin map for Board with SCache option

A.1.3 Virtual Silicon / Virtual Board

The virtual board is a simulation-only platform. The evaluation virtual board comes with virtual RAM which is accessible by a backdoor via netpp. This enables somewhat faster R/W access to the RAM as compared to simulated In Circuit Emulation. For further documentation about virtual interfaces, refer to the GHDLex documentation.

A.2 Core specific maps

A.2.1 Zealot

The address mapping from Table A.3 applies, the actual memory size is defined by the CONFIG_BRAM_ADDR_WIDTH setting. The stack pointer is always initialized at the end of this memory block.

Memory range	Description
0x0000-0x7fff	Program memory, Heap and stack pointer
0x8000-0xffff	Memory mapped registers (MMR)

Table A.3: MaSoCist 'agathe' coarse address map

A.2.2 ZPUng v1

The ZPUng has a no shared Program/Data/Heap and stack memory, instead, it uses a separate stack as shown in Table A.4.

Address range	Description
0x0000-0x3fff	Program memory and Heap
0x4000-0x41ff	Stack memory
0x8000-0xffff	Memory mapped registers (MMR)

Table A.4: ZPUng v1 RAM default address map



The stack memory start address always immediately follows the program memory (which is configured by CONFIG_BRAM_ADDR_WIDTH)

For a system with SCache configuration, the address map is displayed in Table A.5.

Address range	Description
0x00000-0x01fff	Program memory and Heap
0x02000-0x02fff	DCache area
0x03000-0x03fff	ICache area when ICACHE enabled
0x04000-0x041ff	Stack memory (CONFIG_STACK_ADDR_WIDTH=9)
0x08000-0x0ffff	Extended program memory (ROM) Bank A
0x10000-0x1ffff	Extended program Bank B (reserved)
0x20000-0x3ffff	Extended data memory (ROM)
0x78000-0x7ffff	Memory mapped registers (MMR)

Table A.5: ZPUng v1 RAM 's-cache' address map

Register map reference

B.1 Device peripheral map

Table B.1 shows the address map offsets for the peripheral units in this SoC.

Device id	Offset	Description	Details
SysCtrl	0x0F8000	System controller	Section 4.1
GPIO0	0x0F8200	GPIO bank controller	Section 4.3
GPIO1	0x0F8220	GPIO bank controller	Section 4.3
IRQ	0x0F8100	System interrupt controller	Section 4.2
SPI	0x0F8300	Simple SPI controller	Section 4.6
PWM[0..CONFIG_NUM_PWM]	0x0F8400, 0x0F8420, ..	PWM controller (Simple)	Section 4.4.2
TIMER	0x0F8500	Global timer unit	Section 4.4.1
UART	0x0F8600	Simple UART controller	Section 4.5

Table B.1: Device map MaSoCist_agathe

B.2 Register details

B.2.1 MaSoCist_agathe registers

'SysCtrl' core registers

Bit(s)	Name	Description
2	SYS_RESET	System reset. Resets entire CPU (1 active)
0	PERIO_RESET	1: Reset peripheral I/O section

Table B.2: SysCtrl register (RW) Offset: 0x00

Bit(s)	Name	Description
8	DATA_VALID	When this bit is set, CONSOLE_DATA is valid
7:0	CONSOLE_DATA	System console input character data

Table B.3: SysConsole_R register (RO) Offset: 0x0c

Bit(s)	Name	Description
15:8	CPUFLAGS	Core specific CPU flags (undocumented)
7:4	CPUARCH	CPU architecture field (undocumented)
3:0	SOCTYPE	SoC type (0: local bus, 1: Wishbone, 2: AHB lite)

Table B.4: SocInfo register (RO) Offset: 0x18

Bit(s)	Name	Description
31:16	REV_RESERVED	Reserved for further revision tags
15:8	REV_MAJOR	Major hardware revision number
7:0	REV_MINOR	Minor hardware revision number

Table B.5: HWVersion register (RO) Offset: 0x1c

'IRQ' core registers

Bit(s)	Name	Description
7:6	IRQCH3_SEL	IRQ channel assignment [0..3]
5:4	IRQCH2_SEL	IRQ channel assignment [0..3]
3:2	IRQCH1_SEL	IRQ channel assignment [0..3]
1:0	IRQCH0_SEL	IRQ channel assignment [0..3]

Table B.6: SIC_IAR register (RW) Offset: 0x0c

'SPI' core registers

'TIMER' core registers

'PWM' core registers

'UART' core registers

Bit(s)	Name	Description
15	SPIRESET	Set 1 to reset SPI, 0 to resume
14	SPICS	Set '0' to select SPI
6:4	NBITS	Number of bits to transfer
3	LSBFIRST	1; Send LSB first, 0: MSB first
2	CPHA	Capture phase bit
1	CPOL	Invert clock polarity
0	PUMP	When set, access (R or W) to the SPIData register pumps a new transfer

Table B.7: SPI_CONTROL register (RW) Offset: 0x00

Bit(s)	Name	Description
7:6	SPIWIDTH	SPI data width: 0: 8 bit, 1: 16 bit, 2: 32 bit
0	SPIBUSY	When 1, SPI is busy transferring data

Table B.8: SPI_STATUS register (RO) Offset: 0x04

Bit(s)	Name	Description
15	CLKDIV_BYPASS	When set, bypass clock divider
7:0	CLKDIV	Clock divider value

Table B.9: SPI_CLKDIV register (RW) Offset: 0x0c

Bit(s)	Name	Description
15	CRESET	When set, a write to TIMER_START resets clock div counter
14:0	PWMCLKDIV	Clock divider for PWM

Table B.10: TIMER_CONFIG register (RW) Offset: 0x0c

Bit(s)	Name	Description
7:6	TMODE	Timer mode. Not used in simple PWM implementation.
1	DEFAULT	Default pin value at PWM output
0	TMR_IRQEN	Enable system IRQ for timer unit if '1'

Table B.11: PWM_CONFIG register (RW) Offset: 0x00

Bit(s)	Name	Description
1	TMR_RUN	'1' when timer running

Table B.12: PWM_STATUS register (RO) Offset: 0x04

Bit(s)	Name	Description
15	UART_RESET	'1': Reset UART. Clear to run.
14	RX_IRQ_ENABLE	Enable receive IRQ
9:0	UART_CLKDIV	(16x) Clock divider

Table B.13: UART_CONTROL register (RW) Offset: 0x00

Bit(s)	Name	Description
10:8	BITCOUNT	RX bit counter
7	RXOVR	Receiver FIFO overrun. Cleared by UART_RESET.
6	TXOVR	Transmitter FIFO overrun. Cleared by UART_RESET.
5	FRERR	Sticky framing error. Set when stop bit not null. Reset by UART_RESET.
2	TXBUSY	'1' when TX in progress
1	TXREADY	Set when TX FIFO ready for data
0	RXREADY	Set when data ready in RX FIFO

Table B.14: UART_STATUS register (RO) Offset: 0x04

Bit(s)	Name	Description
8	DVALID	1 when data valid (mirror of RXREADY bit)
7:0	RXDATA	UART receive data

Table B.15: UART_RXR register (RO) Offset: 0x08

Known issues

This appendix lists known bugs for cores and peripheral hardware versions.

C.1 CPU cores

C.1.1 Zealot small

For these issues, no workaround is known. Also, fixes are not planned.

- Too frequent interrupts can trash the stack
- Emulation steps inside microcode

C.1.2 ZPUng v0

All issues below are fixed in v1

Issues without workarounds

- Exception/IRQ handlers require at least four cycles from clearing the IPEND register until return
- ICE single stepping would under certain conditions freeze the system (IRQ during immediate opcode)

Issues with workarounds

Interrupts may no longer be triggered when using the reentrancy software trick

Make sure to check for other interrupts inside a handler being serviced and clear ALL IPEND bits at the end of the routine

C.1.3 ZPUng v1 rev0

Issues without workarounds

- Stack trashing (underrun) not guarded. Program will just crash.
 - Fix: Use exception (NMI) for stack underrun

Issues with workarounds

Code in extended program memory can not be debugged

Place the routines in the .text default area first. Once you have found them to be stable, you can move them to the extended program memory. For examining the data, see below.

Experimental software based workarounds do exist, however they don't work with the standard debugger.

Under scrutiny.

Data in extended ROM area can not be dumped directly

Set a break point in the cache handler after the data has been read from the ROM and examine the data from the cache area. Use gdb helper scripts to calculate the proper address.

C.2 Peripheral hardware 'agathe' and 'beatrix'

C.2.1 Revision 0.1

This section applies to the hardware definition v0.1. All listed issues are fixed in the v0.3 revision, if not noted otherwise.

SIC

- No hardware support for reentrant interrupt services

Timer and PWM

- Changing the PWMCLKDIV register would not reset the timer. This could cause a somewhat nondeterministic behaviour of the PWM

SPI

- Pre-Division of system clock for SPI is too high, therefore SPI max. clock is limited
 - 0.2 uses predivider 4 instead of 16
 - Suggestion for later versions: BYPASS bit

C.2.2 Revision 0.2

SIC

- When exception handling interrupt had lower priority and a simultaneous high priority event occurred, the higher priority interrupt service handler would be executed instead
 - 0.3 SICv2 uses specific exception (NMI) input with highest priority, but allows lower priority IRQs to occur once the exception cause has been cleared.



Exceptions are BY DEFAULT non-reentrant!

Timer and PWM

- PWM_WIDTH and PWM_PERIOD were swapped in behaviour. This did not have any effect with the used pwm.c driver. Fixed in 0.3, make sure to use the functions in pwm.c

C.3 Peripheral hardware 'bertram'

C.3.1 Revision 0.3

PWMPlus

- Experimental: Has swapped register behaviour with PWM. Fixed in 0.4