

# WSim tutorial for developers

Loic Lemaitre

March 6, 2009

## **Abstract**

This document is aimed to help developers that would like to create or complement modules on WSim. It is not a full description of WSim source code, but a short introduction to allow to understand its architecture. It also gives some ways to debug your WSim code.

# Contents

# Chapter 1

## Directories list

WSim source code directory (`/wsim`) contains the following subdirectories:

- `/arch`: implementation of the two supported MCU (MSP430 and ATMEGA).
- `/autom4te.cache`:
- `/devices`: implementation of external peripherals;
- `/doc`: wsim website sources;
- `/examples`: example codes to demonstrate WSim main features;
- `/libconsole`: handles the WSim console mode;
- `/libelf`: interface with the `*.elf` program
- `/libetrace`: deals with the trace generation for eSimu;
- `/libgdb`: carries out the link with the gdb debugger;
- `/libgui`: used to show user interface (i.e. the leds blinking);
- `/liblogger`: deals with error and output messages (`wsim.log`, error message on terminal);
- `/libselect`: manages inputs and outputs between WSim and external application (WConsole, WSnet);
- `/libtracer`: generates WSim traces (`*.trc`);
- `/libwsnet`: carries out the link with WSnet application;
- `/machine`: make the link between platform model and simulator;
- `/platforms`: implementation of the different platforms (`wsn430`, `telosb`, `senslab`...);
- `/src`: point of entry of the program. Deal with the wsim options (arguments) too;
- `/utils`: contains compilable sources of useful tools (WTracer, WConsole, ...);

## Chapter 2

# Program execution overview

The point of entry of the program is the `main.c` file located in the `src/` directory.

### 2.1 Initialisation

The initialisation is carried out in the `main()` function of the previous named file.

1. Program starts by adding program options and specific options of the platform (given on command line);
2. Next step is the initialisation of WSim modules: log messages handler (liblogger), interface with external applications (libselect), traces handler (libtracer and libetrace);
3. The machine is then created: machine structure initialisation, and platform (MCU + devices) creation;
4. The \*.elf program is loaded (if WSim is not in debugging mode);
5. Display is created if requested;
6. tracer and etracer are started if requested;
7. WSim is ready to run.

### 2.2 Running

WSim supports 5 different simulation modes:

- **Standard run:** simulation runs until the end of the \*.elf program;
- **Instruction:** simulation runs for a predefined number of MCU instruction;
- **Time:** simulation runs for a predefined time;
- **GDB:** simulation runs in debugging mode, being remoted by GDB;
- **Console:** enables to handle simulation execution from a command line.

At the end of the `main.c` function, the `main_run_mode()` function of the same file is called, in order to select the right simulation mode. At this time WSim is going to execute the first instruction of the \*.elf program. Diagram ?? presents the steps of one instruction execution.

The core of the instruction execution is the `msp430_mcu_run_insn()` function of the `/arch/msp430/msp430_alu.c` file. This function fetches, decodes, executes one instruction, launches devices update and IRQ consideration (performed by `msp430_mcu_update()` function).

If the MCU is in low power mode `msp430_mcu_run_lpm()` of the `/arch/msp430/msp430_alu.c` file is called instead of `msp430_mcu_run_insn()`.

Next paragraph describes the called functions chains, for each WSim mode, to get from the `main.c` file to the execution of one instruction.

- Standard run: `machine_run_free()`  $\longrightarrow$  `machine_run()`  $\longrightarrow$  `mcu_run()`  $\longrightarrow$  `msp430_mcu_run_insn()` or `msp430_mcu_run_lpm()`  $\longrightarrow$  `msp430_mcu_update()`;
- Instruction mode: `machine_run_insn()`  $\longrightarrow$  `machine_run()`  $\longrightarrow$  `mcu_run()`  $\longrightarrow$  `msp430_mcu_run_insn()` or `msp430_mcu_run_lpm()`  $\longrightarrow$  `msp430_mcu_update()`;
- Time mode: `machine_run_time`  $\longrightarrow$  `machine_run()`  $\longrightarrow$  `mcu_run()`  $\longrightarrow$  `msp430_mcu_run_insn()` or `msp430_mcu_run_lpm()`  $\longrightarrow$  `msp430_mcu_update()`;
- GDB mode: `libgdb_target_mode_main()`  $\longrightarrow$  `gdbremote_getcmd()`  $\longrightarrow$  `gdbremote_single_step()` or `gdbremote_continue()`  $\longrightarrow$  `machine_run_free()`  $\longrightarrow$  `machine_run()`  $\longrightarrow$  `mcu_run()`  $\longrightarrow$  `msp430_mcu_run_insn()` or `msp430_mcu_run_lpm()`  $\longrightarrow$  `msp430_mcu_update()`;
- Console mode: `console_mode_main()`  $\longrightarrow$  `console_command()`, then WSim starts one of the four previous mode, depending of the text string you enter on the command line.

## 2.3 End of the simulation

The simulation normally ends if one of these cases is true:

- the `*.elf` program is finished;
- an illegal instruction has been detected in the `*.elf` program;
- the exact number of instructions has been executed (only in instruction mode);
- the simulation time is over (only in time mode);
- "quit" command has been typed (only in console mode);
- GBD is closed before the end of the simulation (only in GDB mode).

For standard, instruction and time modes, statistics of the simulation are written in the `wsim.log` file just before leaving WSim. These statistics are general informations about WSim, and more specific about machine, MCU and devices.



## Chapter 3

# Main internal features management

### 3.1 Memory

#### 3.1.1 General

Main storages are MCU state, platform external devices state (radio, leds, ...), and possibly internal platform state. A backup of these states is periodically carried out so that it allows to recover easily the previous saved state of the platform in order to backtrack.

#### Devices and platform

When a platform is built, an instance of each of its devices is created in order to save its states. The memory allocation is done by the `devices_memory_allocate()` function of the `devices.c` file, located in the `/devices` directory. This function is called by the platform description file. In this function, the needed space to save the states of every platform device is computed, in order to store all the data in a contiguous memory spaces. The pointers to access to them are stored into the `machine.devices_state` and the `machine.devices_state_backup` (`machine` is a global structure storing devices states, devices number, devices size and time of simulation).

Moreover saving internal platform states may be necessary. They are then stored in a platform specific structure (of your platform file), you have to define. This structure must be saved in the same contiguous space than the devices, by considering it as a device.

#### MCU

The MCU state and its backup are stored into global structures (`mcu` and `mcu_backup`), since we know the needed space and as there is only one MCU by platform. So we need not indirections (through pointers) to access to the MCU structure, in spite of the device structure.

#### 3.1.2 Backtracks

Backtracks are used when WSim operates with WSnet. Indeed WSim may need to backtrack in two different cases:

- when one of WSim nodes runs beyond a meeting point, since nodes synchronisation is done by an appointment method. An other meeting point is then set, and the state of the node is restored to the last state saved;
- when one of WSim nodes is in debugging mode, and stopped at a breakpoint, the other nodes are still running. As soon as the WSim node in debugging mode is running again, the other nodes are backtracked.



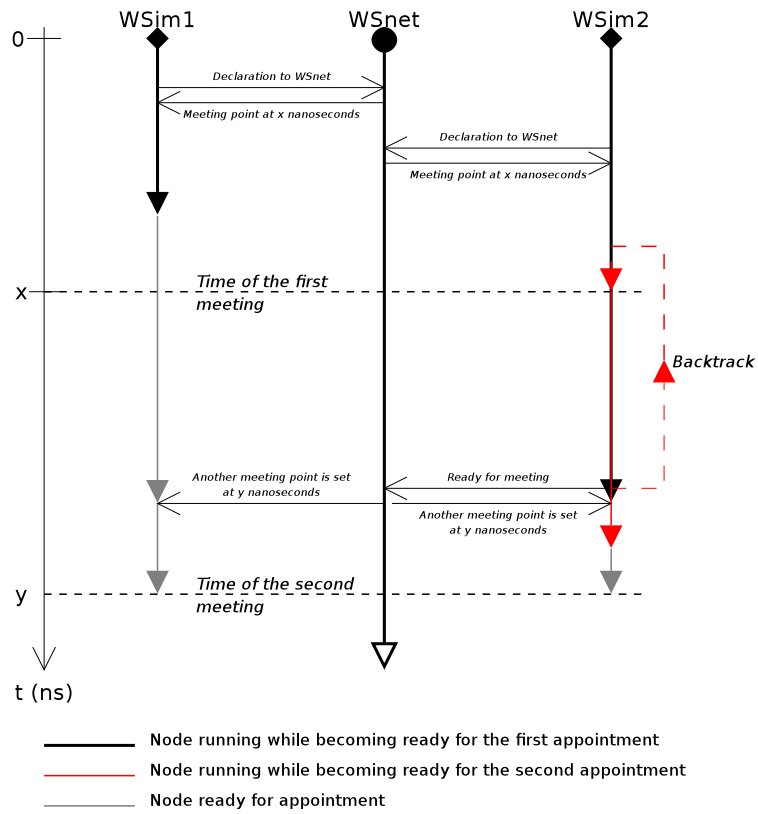


Figure 3.1: WSim backtrack when a node runs beyond a meeting point

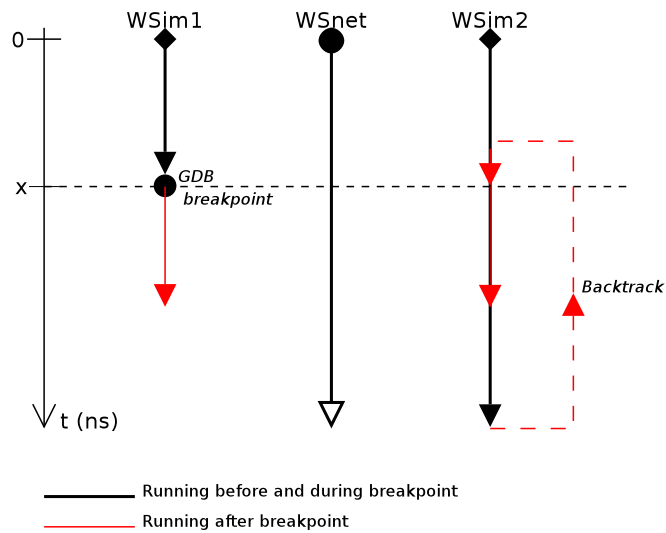


Figure 3.2: WSim backtrack in debugging mode

State backup saves are performed by WSim nodes through the `void machine_state_save()` function, as soon as a synchronisation is successful or a response is received by WSnet from a WSim node. To come back from the present state to the backup one, `void machine_state_restore()` of the `/machine/machine.c` file is called. This function executes the following tasks:

- restoring the state of the MCU (copy of the MCU structure in the MCU backup structure)
- setting the time simulation to the time of the backup
- replacing the content of the present state memory by the content of the backup state one
- restoring traces
- restoring WSnet state

## 3.2 Clocks

MSP430 time resolution is implemented in nanoseconds in WSim. Each time an instruction is executed, the MCLK is incremented and ACLK, ACLKn, SMCLK are computed according to MCLK value.

MSP430 clocks functions are written in the `/arch/msp430/msp430_basic_clock.c` or `/arch/msp430/msp430_fll_clock.c` depending of the MSP430 model.

The figure ?? presents the clocks update chain.

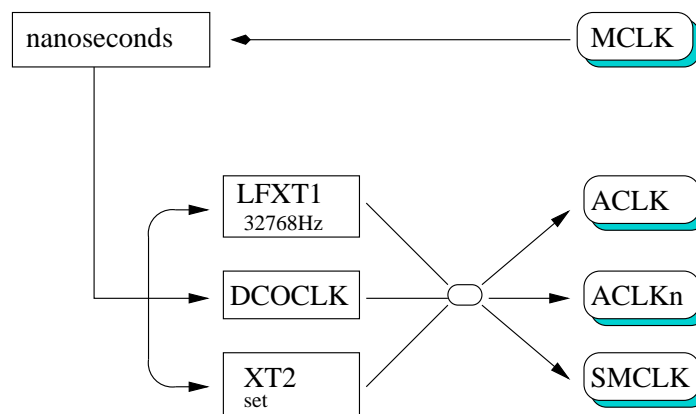


Figure 3.3: WSim MSP430 clocks design

## 3.3 Signals

Signals allows to signal system states switches or errors on the \*elf program executing.

In the `/arch/common/mcu.h` file you can find this following code with its comments:

```

MCU signals
=====

mcu signal is a 32 bits variable that holds signal and
control bitfields

```

|      |      |  |      |      |  |      |      |  |      |      |   |                    |                         |             |
|------|------|--|------|------|--|------|------|--|------|------|---|--------------------|-------------------------|-------------|
| .... | .... |  | .... | .... |  | .... | .... |  | XXXX | XXXX | : | SIG_MCU_xxxx       | or                      | HOST_SIGNAL |
| .... | .... |  | .... | .... |  | .... | ...X |  | .... | .... | : | SIG_MCU_LPM_CHANGE |                         |             |
|      |      |  |      |      |  |      |      |  |      |      |   |                    |                         |             |
| .... | .... |  | .... | ...X |  | .... | .... |  | .... | .... | : | SIG_MCU            | (set if SIG_MCU)        |             |
| .... | .... |  | .... | ..X. |  | .... | .... |  | .... | .... | : | SIG_RUN_INSN       | (insn single step mode) |             |
| .... | .... |  | .... | .X.. |  | .... | .... |  | .... | .... | : | SIG_RUN_TIME       | (time single step mode) |             |
| .... | .... |  | .... | X..  |  | .... | .... |  | .... | .... | : | SIG_GDB_SINGLE     | (gdb single step mode)  |             |
| .... | .... |  | .... | X    |  | .... | .... |  | .... | .... | : | SIG_GDB_IO         | (IO TCP/GDB)            |             |
| .... | .... |  | .... | ..X. |  | .... | .... |  | .... | .... | : | SIG_CON_IO         | (IO Console)            |             |
| .... | .... |  | .... | .X.. |  | .... | .... |  | .... | .... | : | SIG_WORLDSSENS_IO  | (IO UDB/WSNnet)         |             |
| .... | .... |  | .... | X..  |  | .... | .... |  | .... | .... | : | SIG_UI             | (IO UI)                 |             |
|      |      |  |      |      |  |      |      |  |      |      |   |                    |                         |             |
| .... | ...X |  | .... | .... |  | .... | .... |  | .... | .... | : | SIG_HOST           | (Host signal on WSim)   |             |
| .... | ..X. |  | .... | .... |  | .... | .... |  | .... | .... | : | SIG_MAC            | (Memory access control) |             |
| XXXX | X..  |  | .... | .... |  | .... | .... |  | .... | .... | : | SIG_MAC_xxxx       |                         |             |

SIG\_MCU 8bits can be set either for an internal MCU signal (identified by the SIG\_MCU bit) or for an external Unix signal on the WSim process (identified by SIG\_HOST bit).

\*\*\*\*\*/

```

/* mcu internal signal id */
#define SIG_MCU_HUP          0x00000001
#define SIG_MCU_INT          0x00000002
#define SIG_MCU_QUIT         0x00000004 /* used */
#define SIG_MCU_ILL           0x00000008 /* used */
#define SIG_MCU_TRAP          0x00000010 /* used */
#define SIG_MCU_ABRT          0x00000020
#define SIG_MCU_BUS           0x00000040 /* used */
#define SIG_MCU_TSTP          0x00000080
#define SIG_MCU_LPM_CHANGE    0x00000100
#define SIG_MCU_ALL           0x000001ff

#define SIG_HOST_SIGNAL      0x000000ff

/* signal source identifier */
#define SIG_MCU               0x00010000 /* mcu internal signal */
#define SIG_RUN_INSN          0x00020000 /* insn mode */
#define SIG_RUN_TIME          0x00040000 /* time mode */
#define SIG_GDB_SINGLE        0x00080000 /* simul trap for GDB */

#define SIG_GDB_IO            0x00100000 /* gdb tcp io request */
#define SIG_CON_IO            0x00200000 /* console mode */
#define SIG_WORLDSSENS_IO     0x00400000 /* worldsens network io */
#define SIG_UI                0x00800000 /* ui signal (keyboard) */
#define SIG_HOST              0x01000000 /* host signal */
#define SIG_MAC               0x02000000 /* mem breakpoint */
#define SIG_BREAK_MEM_XX      0xf0000000 /*

```

This code defines signals values.

## 3.4 I/O

/libselect

## Chapter 4

# Implementing new WSim modules

### 4.1 Platform

In the WSim design, the platform file is aimed to describe it, but also to make a link between MCU and devices.

#### 4.1.1 Implementing your platform

##### Headers to include

You have to include the following files in your platform implementation:

- The MCU common header (`#include "arch/common/hardware.h"`) and the MCU specific header (`#include "arch/msp430/msp430.h"` or `#include "arch/atmega/atmega128.h"`);
- The device common header (`#include "devices/devices.h"`) and the header of each implemented devices of the platform;
- `#include "src/options.h"` if you want to add platform specific options.

##### List of mandatory functions to implement

- `int devices_options_add(void)`: adds platform specific options with the `option_add()` function;
- `int devices_create(void)`: computes the needed memory space for devices and initialise them;
- `int devices_reset_post(void)`: function called after devices reset, so devices init conditions must be written here;
- `int devices_update(void)`: function called after every MCU instruction execution, this function handles input and output between MCU and devices ports.

##### Intructions in `devices_create()` function

This function is called only once at the simulation initialisation. This intructions sequence should be followed:

1. You have first to take into consideration potential specific options, that might have been provided as a command line argument (only if you implement specific option in `devices_option_add()`). This is done by checking the `value` item of each option structure.
2. MCU must be initialised by calling its `MCUNAME_create()` function;
3. Fix each device size and store it in the `machine.device_size` table. Now call the `devices_memory_allocate()` function of the `/devices/devices.c` file;

4. Create each device with its `DEVICENAME_device_create()` function;
5. Initialise UIs by getting their sizes (`machine.device[DEVICEID].ui_get_id()`) and set their positions (`machine.device[DEVICEID].ui_set_pos()`).

### Intructions in `devices_update()` function

This function is the core of the platform because it describes GPIO and SPI connections between MCU and . Thus every time the MCU decodes an intruction, this function will be called. Instructions sequence is important and you have to follow this order as explained previously (cf chapter ?? page ??): MCU to devices transfer, devices to MCU transfer, devices update. Otherwise SPI communications might experience dysfunctions.

1. First you begin by reading the MCU pins with this function: `MCUNAME_digiIO_dev_read()`, that reads the 8 pins of a MCU port. Then depending of the devices pins configuration, transfer the received value on the right devices pins, by using `machine.device[DEVICEID].write()`. Reiterate the sequence as many times as the number of MCU ports;
2. Do the same operation with the UART or/and SPI ports: use `msp430_UARTORSPI+ID_dev_read_UARTORSPI()` to get pins value and `machine.device[DEVICEID].write()` to send it to the connected device;
3. Now repeat the two first steps in the opposite direction, that is to say from devices to MCU pins. Use `machine.device[DEVICEID].read()` to read devices pins and `msp430_usart+ID_dev_write_UARTORSPI()` to write MCU SPI or UART pins, or `msp430_digiIO_dev_write()` to write MCU GPIOs.
4. Finally these modules must be updated:
  - libselect to update external I/O of WSim: `LIBSELECT_UPDATE()`;
  - libwsnet to update link with WSnnet : `LIBWSNET_UPDATE()`;
  - platform devices to update their internal states: `machine.device[DEVICEID].update()` for each device.

**Remark:** To make your platform more reliable, and give value-added to the simulation, you may add tests for illegal operations not to happen, for example:

- Checking if MCU is in SPI mode before reading a SPI device;
- Checking if MCU is in UART mode before reading an UART device;
- If there are more than one device on one SPI, checking that their CSs are not enabled at the same time;
- Any other test you need...

### SDL/UI

WSim enables you to print on screen an picture associated to your platform. This module is considered as a device, and is thus implemented in the `/devices/uigfx` folder.

#### 4.1.2 Making your platform compilable and executable

To compile WSim, makefiles are generated with the help of the GNU Project Autotools (automake  $\geq$  1.10, autoconf  $\geq$  2.61) <sup>1</sup>. You have to modify three files to compile your platform : `./configure.ac`, `./platforms/Makefile.am`, `./platforms/YOURPLATFORMFOLDER/Makefile.am`.

<sup>1</sup>For further informations please see <http://www.gnu.org/software/automake/> and <http://www.gnu.org/software/autoconf/> websites

## **/configure.ac**

1. In the `platform` model part, add an option to enable to compile only your platform by using the command `./configure --enable-platform-yourplatformname`:

```
dnl yourplatformname
AC_ARG_ENABLE([platform-yourplatformname],AS_HELP_STRING([--enable-\
platform-yourplatformname],[yourplatformname platform]))
if test "${enable_platform_senslab}" = "yes" ; then
enable_mcu_msp430=yes dnl or enable_mcu_atmega=yes
NPLATFORM=$((NPLATFORM + 1))
PLATFORMNAMES="yourplatformname"
fi
```

2. And at the end of the `platform` model part insert the following line:

```
AM_CONDITIONAL([BUILD_YOURPLATFORMNAME], [test "${enable_platform_yourplatformname}" \
= "yes" -o "$ALL" = "yes" ])
```

This line initialises the `BUILD_YOURPLATFORMNAME` variable to 1 if your platform only or all the platforms must be built, else to 0 (the purpose of this variable is developed in paragraph ?? page ??).

3. Add the path to your platform makefile into `AC_CONFIG_FILES` of the output part:

```
platforms/YOURPLATFORMFOLDER/Makefile
```

## **/platforms/Makefile.am**

Simply add the name of your platform directory in the `SUBDIRS` variable.

```
SUBDIRS=wsn430 ot2006 otsetre ez430 tests telosb \
mosar mica2 micaz iclbsn wisenode senslab yourplatformname
```

## **/platforms/YOURPLATFORMFOLDER/Makefile.am**

1. First test if your platform must be built or not, thanks to the `BUILD_YOURPLATFORMNAME` variable defined with the `AM_CONDITIONAL` command in the `configure.ac` file:

```
if BUILD_YOURPLATFORMNAME
```

2. Next set the program name and the `worldsens` program name (program running with `WSnet`) for your platform:

```
bin_PROGRAMS=wsim-yourplatformname
if BUILD_WORLDSENS
bin_PROGRAMS+=worldsens-yourplatformname
endif
```

3. This line adds preprocessor arguments (here it enables to include `/wsim` top directory at compilation):

```
INCLUDES=-I$(top_srcdir)
```

4. Then define the MCU and devices libraries dependences paths. For instance:

```
YOURPLATFORMNAME_MCU= ../../arch/msp430/libmsp430f1611.a
YOURPLATFORMNAME_DEV= ../../devices/led/libled.a      \
                      ../../devices/ds2411/libds2411.a \
                      ../../devices/m25p80/libm25p80.a \
                      ../../devices/ptty/libptty.a    \
                      ../../devices/uigfx/libuigfx.a   \
                      ../../devices/cc1100/libcc1100.a
```

5. Add specific compilation flags:

```
wsim_yourplatformname_CFLAGS=-DMSP430f1611
```

This flag will define the global macro MSP430f1611 during the compilation.

6. Declare the name of your platform source file:

```
wsim_yourplatformname_SOURCES=yourplatformname.c
```

7. Declare the libraries dependences path set below:

```
wsim_yourplatformname_LDADD=${YOURPLATFORMNAME_DEV} ${WSIMADD} ${YOURPLATFORMNAME_MCU}
```

`${WSIMADD}` is defined in the `/platform/Makefile.am` file, and sets some general WSim library dependence.

8. Finally do not forget to close the `if BUILD_YOURPLATFORMNAME`

```
endif
```

## 4.2 Device

### 4.2.1 Adding a new device model

#### List of mandatory functions to implement

- `int YOURDEVICENAME_add_options()`: enables to add device specific options when starting a simulation.
- `int YOURDEVICENAME_device_size()`: returns size the device structure needs to store its internal states.
- `int YOURDEVICENAME_device_create()`: creates an instance of the device, by initialising the states in `machine.device[DEVICEID].data` and storing in `machine.device[DEVICEID]` device private functions to be called during the simulation.

#### List of optional functions to implement

Depending of devices features some of these private functions have to be implemented:

- `int YOURDEVICENAME_write()`: transmits MCU informations to the device
- `int YOURDEVICENAME_read()`: transmits device informations to the MCU (for example leds need not this function);



- `int YOURDEVICENAME_update()`: updates internal state of the device after read or write action (for instance emptying radio TX buffer after its content has been transmitted to the MCU);
- `int YOURDEVICENAME_delete()`: frees memory space filled by the device (excepted the device states);
- `int YOURDEVICENAME_reset()`: resets the device (at its default states);
- `int YOURDEVICENAME_power_up()`: for potential futur use;
- `int YOURDEVICENAME_power_down()`: for potential futur use;
- `int YOURDEVICENAME_ui_draw()`: draws device graphical interface;
- `int YOURDEVICENAME_ui_set_pos()`: sets the position of device graphical interface;
- `int YOURDEVICENAME_ui_get_pos()`: gives the position of device graphical interface;
- `int YOURDEVICENAME_ui_get_size()`: gives the size of device graphical interface;
- `int YOURDEVICENAME_statdump()`: may be used to return device statistics (called only at the end of the simulation).

## 4.2.2 IO pins interface management

### General

There are 2 main class of IO, GPIOs for general use, and USART. IO device pins are the interface between device and MCU. In the platform file, these pins are going to be read and write from/to MCU, thanks to read and write device functions as described below. To select the device pin to read or write, a mask variable may be used. For example, the following masks are defined for the M25P80 flash memory device (in the `./devices/m25p80/m25p80.h` file):

```
#define M25P_W_SHIFT 8 /* write protect */
#define M25P_S_SHIFT 9 /* select */
#define M25P_H_SHIFT 10 /* hold */
#define M25P_C_SHIFT 11 /* clock */

#define M25P_D 0x00ff /* data 8 bits */
#define M25P_W (1 << M25P_W_SHIFT) /* write protect negated */
#define M25P_S (1 << M25P_S_SHIFT) /* chip select negated */
#define M25P_H (1 << M25P_H_SHIFT) /* hold negated */
#define M25P_C (1 << M25P_C_SHIFT) /* clock */
```

Table ?? and figure ?? clarifies this C code:

| Name   | Value               | Pin(s)           |
|--------|---------------------|------------------|
| M25P_D | 0000 0000 1111 1111 | SPI SDI, SPI SDO |
| M25P_W | 0000 0001 0000 0000 | FLASH W          |
| M25P_S | 0000 0010 0000 0000 | FLASH CS         |
| M25P_H | 0000 0100 0000 0000 | FLASH HOLD       |
| M25P_C | 0000 1000 0000 0000 | SPI CLOCK        |

Table 4.1: Value of the M25P80 masks

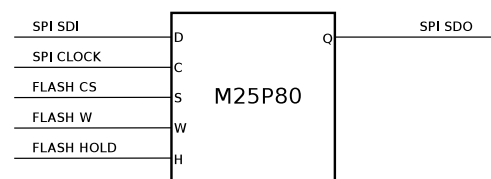


Table 4.2: M25P80 pins configuration

Thus, writing CS and HOLD pins might be implement like this:

```
uint32_t value = 0000 0110 0000 0000;
machine.device[FLASH].write(FLASH, M25P_H | M25P_S, value);
```

## USART

On the MSP430, USART may be configured in three modes: SPI, UART and I2C. However I2C is not implemented in WSim yet.

The WSim simulation of USART does not follow exactly the real hardware running. Indeed instead of sending data bit by bit to devices, WSim USART transmits byte by byte, in simulating the time to send one byte. Hence a byte is only ready to be transmitted after 8 UCLK cycles have been counting. The same method is used for USART data reception from devices. It allows to simplify USART transfer implementation and to save computation time by avoiding functions calls.

This choice of implementation has no effects on the reliability of the simulation.

### SPI particularity

SPI bus works always in full duplex, that is to say every time data is transmitted to a SPI device, this one sends a synchronized response at the same time. Thus, in order to get data from a SPI device if nothing has to be transfered to it, you must send a dummy data to trigger the device response.

In WSim, SPI interface is implemented in the `arch/msp430/msp430_usart.c` file for the MSP430 MCU. This implementation do not check if SPI communication are full duplex. It may be done in your platform or in your device code.

The better way to implement SPI full duplex communications, is to take it into consideration in your device model code. As soon as your device `YOURDEVICENAME_write()` function is called by your platform for SPI writings, data must be immediately available for `YOURDEVICENAME_read()` function.

Indeed, as explained in subsection ?? page ??, the `device_update()` function of the platform performs writings (on devices, so readings on MCU), readings (on devices, so writings on MCU), and finally update devices.

However all devices models may not be implemented in that way. For some devices (cc2420, cc1100, ...), writing on their SPI input (SI) is taken into consideration only when the device is updated at the end of a updating platform cycle. As the reading action is performed after the writing one, no response is sent on the device SPI output (SO) yet.

This leads to a small lag between the simulation and the reality, since the SPI `UxRXBUF` register receives the response on the next `device_update()` call, that is to say few MCU cycles later (from 1 to 6). The figure ?? illustrates that.

In most cases this small lag will not be annoying.

### 4.2.3 Making your device model compilable

The procedure to make your device model compilable is quite similar to the platform one (please refer to the subsection ?? page ?? for more details). Thus you have to modify these following files to compile your device : `./configure.ac`, `./devices/Makefile.am`, `./devices/YOURDEVICEFOLDER/Makefile.am`.

#### `/configure.ac`

Add the path to your device makefile into `AC_CONFIG_FILES` of the output part:

`devices/YOURDEVICEFOLDER/Makefile`

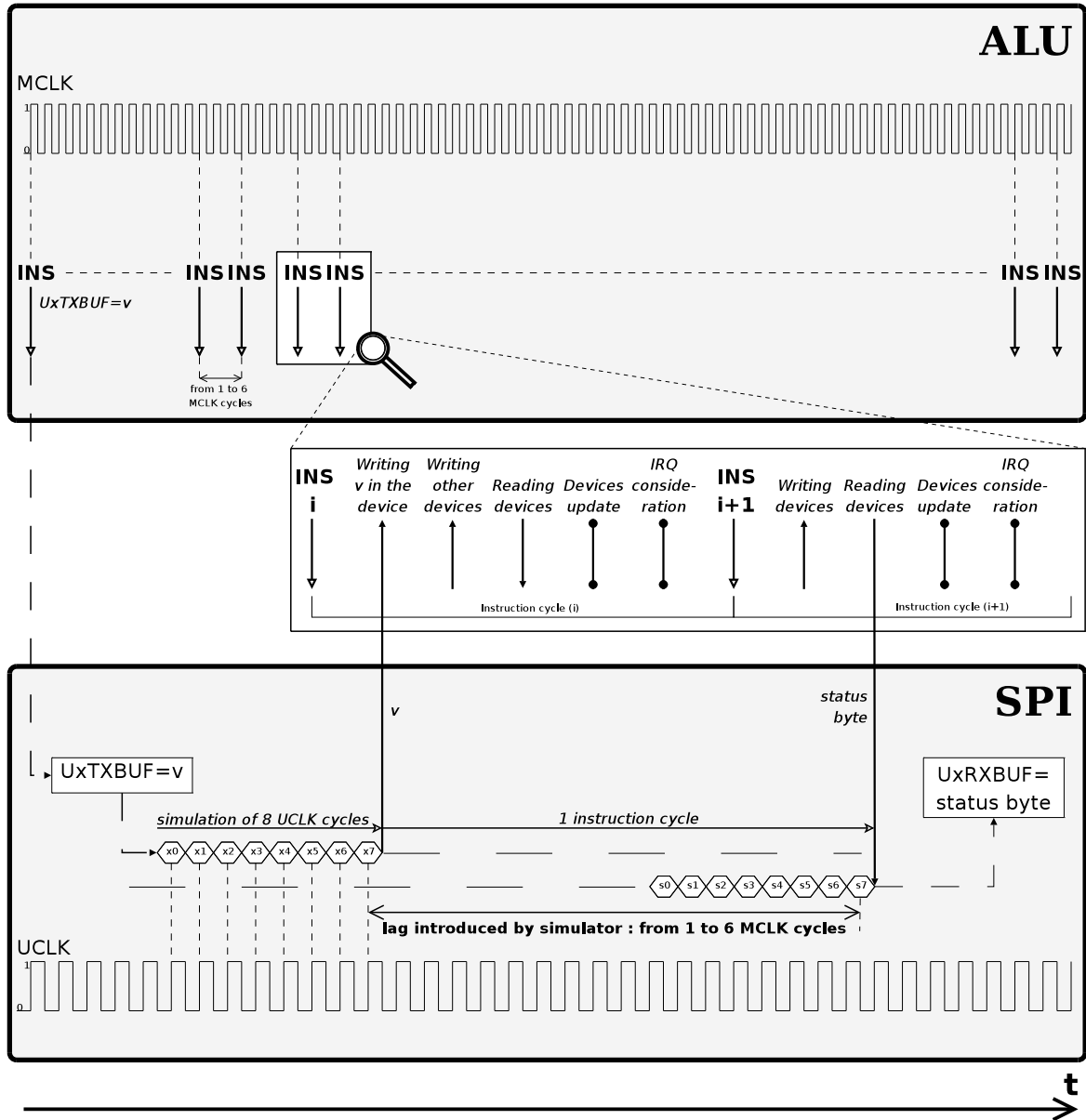


Figure 4.1: WSim SPI and device communication particular case

/devices/Makefile.am

Simply add the name of your platform directory in the SUBDIRS variable.

```
SUBDIRS = \
  7seg \
  at45db \
  bargraph\
  cc1100 \
  cc2420 \
  ds2411 \
  gdm1602a\
```

```
hd44780 \
led      \
m25p80   \
ptty     \
uigfx    \
yourdevicename
```

**/devices/YOURDEVICEFOLDER/Makefile.am**

Compilation of your device model source has to build a static library \*.a, to make it reusable.

1. First set the name of your device library and require it not be installed:

```
noinst_LIBRARIES=libyourdevicename.a
```

2. Next add this line to give additional preprocessor arguments (here it enables to include /wsim top directory at compilation):

```
INCLUDES=-I$(top_srcdir)
```

3. Declare the name of your device model source files, for example:

```
libled_a_SOURCES=yourdevicename_source1.h yourdevicename_source1.c\
yourdevicename_source2.h yourdevicename_source2.c\
yourdevicename_source3.h yourdevicename_source4.h\
```

## 4.3 Special device

### 4.3.1 Pseudo serial PTTY

## 4.4 Microcontroller

## Chapter 5

# Debugging WSim

There are several methods to debug WSim code source.

### 5.1 wsim.log file

At each simulation a log file is generated by WSim, named `wsim.log` and located in the directory where WSim has been launched. This file contains debugging information. To enable debugging information, you have to define the macro `DEBUG`. There are two ways to do so:

- when launching `configure` file before compiling WSim add the option `./configure --enable-debug`;
- or define directly the macro `DEBUG` in the WSim file `xxx_debug.h` located in the same directory than the file you want to debug, as shown in the following example:

```
/**
 * \file   cc2420_debug.h
 * \brief  CC2420 debug messages
 * \author Nicolas Boulicault
 * \date   2007
 **/

/*
 * cc1100_debug.h
 *
 *
 * Created by Nicolas Boulicault on 04/06/07.
 * Copyright 2007 __WorldSens__. All rights reserved.
 *
 */

#ifndef _CC2420_DEBUG_H_
#define _CC2420_DEBUG_H_

/*****
/*****
/*****

#define DEBUG
#if defined(DEBUG)
#define CC2420_DEBUG(x...)    VERBOSE(2,x)
```

```

#define CC2420_DBG_RX(x...)    VERBOSE(2,x)
#define CC2420_DBG_TX(x...)    VERBOSE(2,x)
#else
#define CC2420_DEBUG(x...)     do { } while (0)
#define CC2420_DBG_RX(x...)    do { } while (0)
#define CC2420_DBG_TX(x...)    do { } while (0)
#endif

/*****/
/*****/
/*****/

#define CC2420_PINS_DEBUG

```

The second method has the advantage to print only debug messages of the file where `DEBUG` is defined, contrary to the first one that behaves like `DEBUG` were defined in each file of the program.

By default, debugging information in `wsim.log` file is minimal (default verbose level equal to 0). Nevertheless more information may be output by increasing verbose level when starting simulation: `--verbose=6` for instance. In the previous code, we have `#define CC2420_DBG_RX(x...) VERBOSE(2,x)`. This means that the `CC2420_DBG_RX` will be print in the `wsim.log` file only if the verbose level is superior or equal to 2.

Thus you understand that it is quite easy to add your own debug informations, if you need. You just have to define the print debugging function with its verbose level at the beginning of your file (or its associated \*.h file). Then insert it with appropriate debugging message (a `printf()` function) at the right place.

Notice that the `wsim.log` file name may be changed into `NAMEOFOYOURCHOICE.log` by using the option `--logfile=NAMEOFOYOURCHOICE.log`.

## 5.2 ERROR() function

The `ERROR()` function is closed to the `VERBOSE()` function, excepted that its result is printed in the `wsim.log` file and in the standard error output too, that is to say the terminal you launch WSim. You can insert `ERROR()` function where you want without needs to declare it. Its syntax is as a `printf()` function. Moreover some `ERROR` messages are printed only if the macro `DEBUG_ME_HARDER` is defined:

```

#if defined(DEBUG_ME_HARDER)
    ERROR("senslab:devices: read data on radio while not in SPI mode ?\n");
#endif

```

## 5.3 Wsim trace

The `tracer_event_record()` function, insered in the Wsim code, allows to save states of a variable during the simulation. To enable the trace storage, you just have to add the following option when starting a simulation: `--trace` (example: `wsim-wsn430 --ui --trace --mode=time --modearg=100000000000 wsn430-leds.elf`). The trace is then stored in the `wsim.trc` file and located in the directory where WSim has been launched.

To make the `wsim.trc` file usable, you have to convert it with the external `wtracer` application, according to the following syntax. For instance:

- `wtracer --in=wsim.trc --out=wsim.gp --format=gplot` generates the gnuplot `wsim.gp` file.
- `wtracer --in=wsim.trc --out=wsim.vcd --format=vcd` generates the vcd `wsim.vcd` file, readable by GTKwave.

## 5.4 Using GDB

It is also possible to debug WSim by using GDB. Launch GDB in the folder where the \*.elf file is located, set your breakpoints, and execute the run command followed by the WSim arguments, including your \*.elf file. Here is an example:

```
loic@loic-laptop:~/Documents/Senstools/temp/senslab/node/wsn430_SW/wsn430-drivers/
wsn430-cc2420\$ gdb wsim-senslabv14
GNU gdb 6.8-debian
Copyright (C) 2008 Free Software Foundation, Inc.
License GPLv3+: GNU GPL version 3 or later <http://gnu.org/licenses/gpl.html>
This is free software: you are free to change and redistribute it.
There is NO WARRANTY, to the extent permitted by law. Type "show copying"
and "show warranty" for details.
This GDB was configured as "i486-linux-gnu"...
(gdb) b main.c:200
Breakpoint 1 at 0x8054568: file main.c, line 200.
(gdb) r cc2420-tx.elf
Starting program: /usr/local/bin/wsim-senslabv14 cc2420-tx.elf
[Thread debugging using libthread_db enabled]
[New Thread 0xb7c168c0 (LWP 23589)]
[Switching to Thread 0xb7c168c0 (LWP 23589)]

Breakpoint 1, main (argc=2, argv=0xbf976f24) at main.c:200
200  options_start();
(gdb) n
201  ui_options_add();
(gdb)
```

# Chapter 6

## Appendix

### 6.1 Abbreviations

**ACLK** = Auxiliary Clock (of the MCU)

**CS** = Chip Select

**GPIO** = General Purpose Input Output

**I2C** = Inter Integrated Circuit

**IO** = Input Output

**IRQ** = Interrupt Request

**LPM** = Low Power Mode (for the MCU)

**MCLK** = Master Clock (of the MCU)

**MCU** = MicroController Unit

**PTY** = Pseudo Terminal Type

**SI** = SPI device input

**SMCLK** = Sub Main Clock (of the MCU)

**SO** = SPI device output

**SPI** = Serial Peripheral Interface

**UART** = Universal Asynchronous Receiver Transmitter

**UCLK** = Uncore Clock

**UI** = User Interface

**USART** = Universal Synchronous/Asynchronous Receiver Transmitter

**UxRXBUF** = SPI register used for reception

**UxTXBUF** = SPI register used for transmission