WSim tutorial for developers

Loïc Lemaître

November 25, 2010

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`: WSim console mode;
- `/libelf`: interface with the `*.elf` program
- `/libetrace`: trace generation for eSimu;
- `/libgdb`: gdb debugger interface;
- `/libgui`: graphical interface;
- `/liblogger`: error and output messages (wsim.log, error message on terminal);
- `/liblogpkt`: radio packets logger;
- `/libselect`: inputs and outputs between WSim and external application (WConsole, WSnet);
- `/libtracer`: WSim traces (`*.trc`);
- `/libwsnet`: WSnet interface;
- `/machine`: bind 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`: 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 2.1 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()` ⟶ `machine_run()` ⟶ `mcu_run()` ⟶ `msp430_mcu_run_insn()` or `msp430_mcu_run_lpm()` ⟶ `msp430_mcu_update()`;
- Instruction mode: `machine_run_insn()` ⟶ `machine_run()` ⟶ `mcu_run()` ⟶ `msp430_mcu_run_insn()` or `msp430_mcu_run_lpm()` ⟶ `msp430_mcu_update()`;
- Time mode: `machine_run_time` ⟶ `machine_run()` ⟶ `mcu_run()` ⟶ `msp430_mcu_run_insn()` or `msp430_mcu_run_lpm()` ⟶ `msp430_mcu_update()`;
- GDB mode: `libgdb_target_mode_main()` ⟶ `gdbremote_getcmd()` ⟶ `gdbremote_single_step()` or `gdbremote_continue()` ⟶ `machine_run_free()` ⟶ `machine_run()` ⟶ `mcu_run()` ⟶ `msp430_mcu_run_insn()` or `msp430_mcu_run_lpm()` ⟶ `msp430_mcu_update()`;
- Console mode: `console_mode_main()` ⟶ `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.

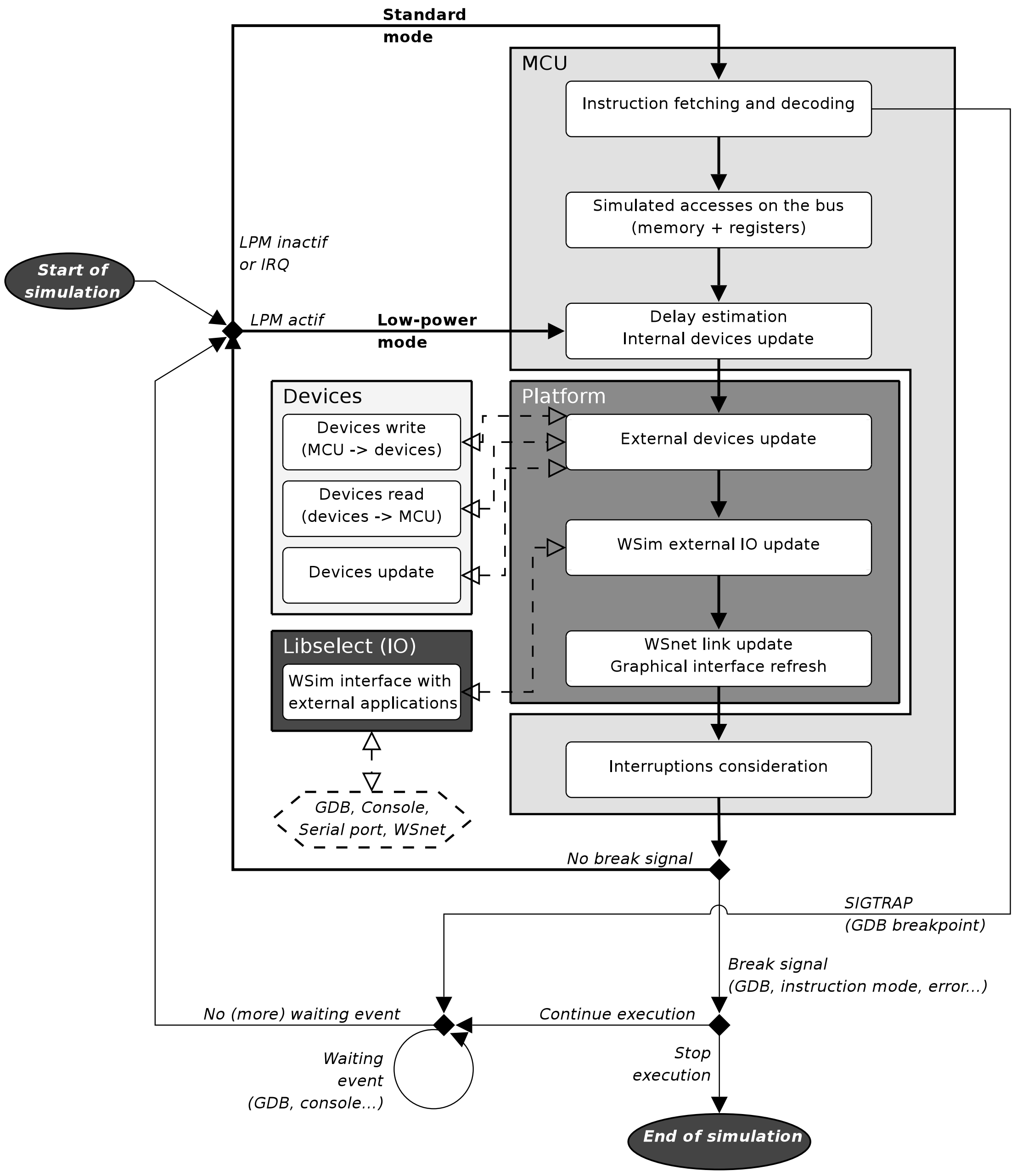
Standard mode
MCU
Instruction fetching and decoding
Simulated accesses on the bus (memory + registers)
LPM inactif or IRQ
Start of simulation
LPM actif
Low-power mode
Delay estimation Internal devices update
Devices
Devices write (MCU -> devices)
Devices read (devices -> MCU)
Devices update
Platform
External devices update
WSim external IO update
WSnet link update Graphical interface refresh
Libselect (IO)
WSim interface with external applications
GDB, Console, Serial port, WSnet
Interruptions consideration
No break signal
SIGTRAP (GDB breakpoint)
Break signal (GDB, instruction mode, error...)
No (more) waiting event
Continue execution
Waiting event (GDB, console...)
Stop execution
End of simulation

Figure 2.1: Steps of one instruction execution

Chapter 3

Main internal features management

3.1 Memory

3.1.1 General

Main storage 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. WSim may need to backtrack in two different cases indeed:

- when an event is added before next rendez-vous, and a node is running beyond this event time. An other meeting point is then set at event time, 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.

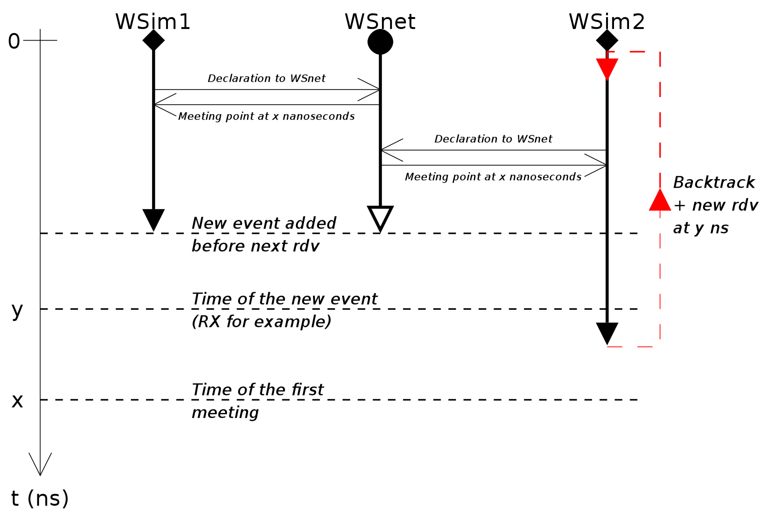

Figure 3.1: WSim backtrack when a new event is added before next rendez-vous

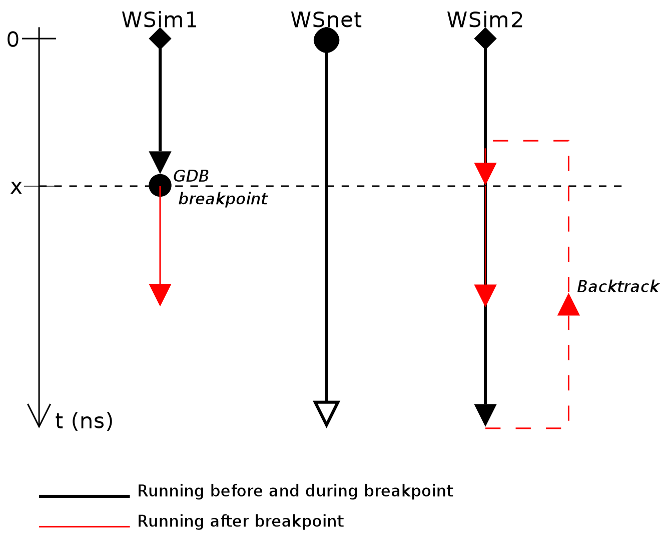

Figure 3.2: WSim backtrack in debugging mode

State backup saves are performed by WSim nodes through the `void machine_state_save()` function, as soon as a synchronisation is successfull 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 3.3 presents the clocks update chain.

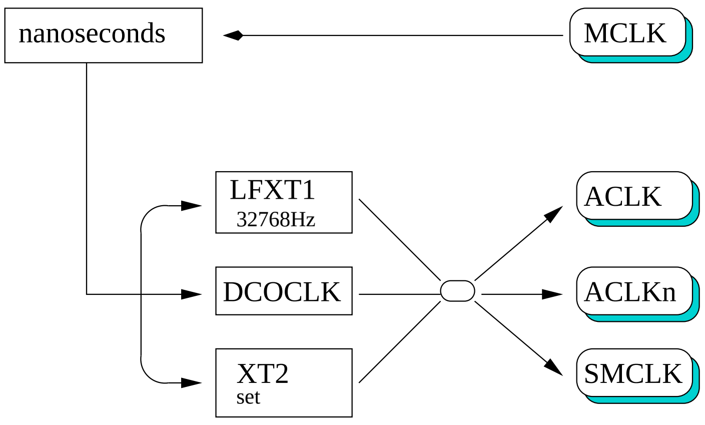

Figure 3.3: WSim MSP430 clocks design

3.3 Signals

Signals allows to signal switches of system states or errors on the *elf program execution. Two main types of signals are implemented:

- The MCU internal signals
- The signal source identifiers

3.3.1 Signals values and usages

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_WORLDSENS_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_WORLDSENS_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.

Table 3.1 describes values and usages of MCU internal signals.

Signal	Value	Usage	Status
`SIG_MCU_HUP`	0x00000001	Disconnection detected on the control terminal or death of the control processus	not used
`SIG_MCU_INT`	0x00000002	Interrupt from keyboard	not used
`SIG_MCU_QUIT`	0x00000004	”Quit” request from keyboard	not used
`SIG_MCU_ILL`	0x00000008	Illegal instruction	used
`SIG_MCU_TRAP`	0x00000010	Breakpoint reached	used
`SIG_MCU_ABRT`	0x00000020	Stop signal from `abort()` function of `libc` library	not used
`SIG_MCU_BUS`	0x00000040	Reading or writing at an unknown or invalid memory address	used
`SIG_MCU_TSTP`	0x00000080	Stop requested from tty	not used
`SIG_MCU_LPM_CHANGE`	0x00000100	Power mode change	not used
`SIG_MCU_ALL`	0x00000200	?	not used

Table 3.1: MCU internal signals

3.3.2 Signals management

Signals are added and removed by using the `mcu_signal_add()` and `mcu_signal_remove()` commands of the `arch/msp430/msp430.c` or `arch/atmega/atmega128.c` file. Signals status are stored in the 32 bits `mcu.alu.signal` variable.

3.4 I/O

3.4.1 General

I/Os between WSim and external applications are handled dynamically by the `libselect` library. This upper layer allows for instance to create easily an I/O for a WSim device with several possibilities of interface (TCP, UDP, STDOUT, ...). Next section lists APIs supplied by `libselect` library.

3.4.2 Public functions

- `libselect_id_t libselect_id_create(char *name, int UNUSED flags)` : enables to add a dynamic external I/O (from a WSim device for example), which may be of type: TCP server (`tcp:s:`), TCP client (`tcp:c:`), UDP (`udp:`), STDIO (`stdio`), STDIN (`stdin`), STDOUT (`stdout`), or (`create`).
- `int libselect_id_is_valid(libselect_id_t id)` : to check if I/O `id` is valid (i.e. created, registered and not closed).
- `int libselect_id_close (libselect_id_t id)` : closes I/O `id` (deletes it of opened I/O list).
- `int libselect_id_register(int id)` : makes I/O `id` available.

- `int libselect_id_unregister(int id)` : makes I/O `id` unavailable.
- `int libselect_id_add_callback(int id, libselect_callback callback, void *ptr)` : function to be called if fifo of the I/O has been closed (only available for TCP I/O).
- `uint32_t libselect_id_read(libselect_id_t id, uint8_t *data, uint32_t size)` : gets data from the I/O fifo.
- `uint32_t libselect_id_write(libselect_id_t id, uint8_t *data, uint32_t size)` : puts data into the I/O fifo.
- `int libselect_fd_register(int fd, unsigned int signal)` : registers a file descriptor for an I/O, but just handles with its signals and reports if data is waiting in its file descriptor.
- `int libselect_fd_unregister(int fd)` : unregisters the file descriptor of the I/O `id`.

You may refer to `/devices/ptty/ptty_dev.c`, where some of these functions are used to manage the I/O of the PTTY and to section 4.3.1 page 22 for more informations on the PTTY.

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 2 page 4): 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 WSnet : `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

If your platform embeds graphical device, you have to update and refresh them when required. See the leds example:

```
/* port 5 :                        */
/* ========                        */
/*   P5.7 NC                       */
/*   P5.6 led 3 (Blue)             */
/*   P5.5 led 2 (Green)            */
/*   P5.4 led 1 (Red)              */
/*   P5.3 SPI flash ram UCLK       */
/*   P5.2 SPI flash ram SOMI         */
/*   P5.1 SPI flash ram SIMO         */
/*   P5.0 NC                         */
if (msp430_digiIO_dev_read(PORT5,&val8))
  {
    machine.device[LED1].write(LED1,LED_DATA, ! BIT(val8,4));
    etracer_slot_access(0x0, 1, ETRACER_ACCESS_WRITE, ETRACER_ACCESS_BIT, ETRACER_ACCESS_LVL_GPIO,
    UPDATE(LED1);
    REFRESH(LED1);

    machine.device[LED2].write(LED2,LED_DATA, !  BIT(val8,5));
    etracer_slot_access(0x0, 1, ETRACER_ACCESS_WRITE, ETRACER_ACCESS_BIT, ETRACER_ACCESS_LVL_GPIO,
    UPDATE(LED2);
    REFRESH(LED2);

    machine.device[LED3].write(LED3,LED_DATA, ! BIT(val8,6));
    etracer_slot_access(0x0, 1, ETRACER_ACCESS_WRITE, ETRACER_ACCESS_BIT, ETRACER_ACCESS_LVL_GPIO,
    UPDATE(LED3);
    REFRESH(LED3);
  }
```

On WSN430 platform leds are connected on pins 5.4, 5.5 and 5.6 of the MSP430, and are refreshed as soon as one of port 5 has been updated.

WSim also enables you to print on screen an image associated to your platform. This module is considered as a device, and is thus implemented in the `/devices/uigfx` folder. Its creation is follow the same procedure as the other devices one (`uigfx_device_size()` and `uigfx_device_create()` function).

For both devices UI and screen image, you must set the place of the graphical interface after creation. Use the device APIs `machine.device[DEVICEID].ui_get_pos()` and `machine.device[DEVICEID].ui_set_pos()` to perform this.

Finally if there are one or several graphical UI, the `ui_refresh()` function of the `/libgui/ui.c` file has to be called at the end of the `devices_update()` function. `libgui` module implementation relies on SDL (Simple Direct media Layer) that is a free and open source software multimedia library.

4.1.2 Making your platform compilable and executable

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

/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
   ```

[1] For further informations please see http://www.gnu.org/software/automake/ and http://www.gnu.org/software/autoconf/ websites

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 4.1.2 page 16).

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 dependencies 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 dependencies 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[2]

List of mandatory APIs 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 private 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 future use;
- `int YOURDEVICENAME_power_down()`: for potential future 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).

[2]Templates sources codes are available in appendix 6.2.1 page 28

device_t structure

`device_t` structure stores function addresses described in the previous subsection, in order to make their calls easier. The structure is designed like this:

```
struct device_t
{
  int  (*update)        (int self);
  void (*read)          (int self, uint32_t *mask, uint32_t *value);
  void (*write)         (int self, uint32_t  mask, uint32_t  value);

  // create is static
  // size   is static
  int  (*delete)        (int self);

  int  (*reset)         (int self);
  int  (*power_up)      (int self);
  int  (*power_down)    (int self);

  int  (*ui_draw)       (int self);
  void (*ui_set_pos)    (int self, int  x, int  y);
  void (*ui_get_pos)    (int self, int *x, int *y);
  void (*ui_get_size)   (int self, int *width, int *height);

  void (*statdump)      (int self, int64_t user_nanotime);

  int state_size;
  int dev_num;

  char* name;
  void* data;
};
```

Of course using all items of the structure is not mandatory.

LED example

LED device is the simplest example of device implementation. Only `write()`, `delete()`, `reset()`, and graphical functions are used in the list of `device_t` private functions. However its `create()` function is closed to the other devices ones. It mainly consists in storing private functions addresses in the `machine.device[DEVICEID]` structure.

```
int led_device_create(int dev_num, uint32_t on, uint32_t off, uint32_t bg, char* name)
{
  struct led_t *dev = (struct led_t*) machine.device[dev_num].data;

  dev->img      = led_img_create(on,off,bg);
  dev->update   = 0;
  dev->val      = 0;
  dev->trid     = tracer_event_add_id(1, name, "led");

  machine.device[dev_num].reset         = led_reset;
  machine.device[dev_num].delete        = led_delete;

  machine.device[dev_num].write         = led_write;
  machine.device[dev_num].update        = led_update;

  machine.device[dev_num].ui_draw       = led_ui_draw;
  machine.device[dev_num].ui_get_size   = led_ui_get_size;
  machine.device[dev_num].ui_set_pos    = led_ui_set_pos;
  machine.device[dev_num].ui_get_pos    = led_ui_get_pos;

  machine.device[dev_num].state_size    = led_device_size();
  machine.device[dev_num].name          = "Led display";

  return 0;
}
```

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 4.1 and figure 4.2 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

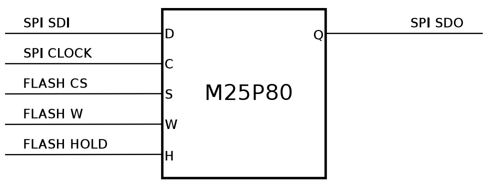

Table 4.2: M25P80 pins configuration

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. 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.
As explained in subsection 4.1.1 page 14, 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 4.1 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 4.1.2 page 15 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
```

/devices/Makefile.am

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

```
SUBDIRS =    \
    7seg     \
    at45db   \
    bargraph\
```

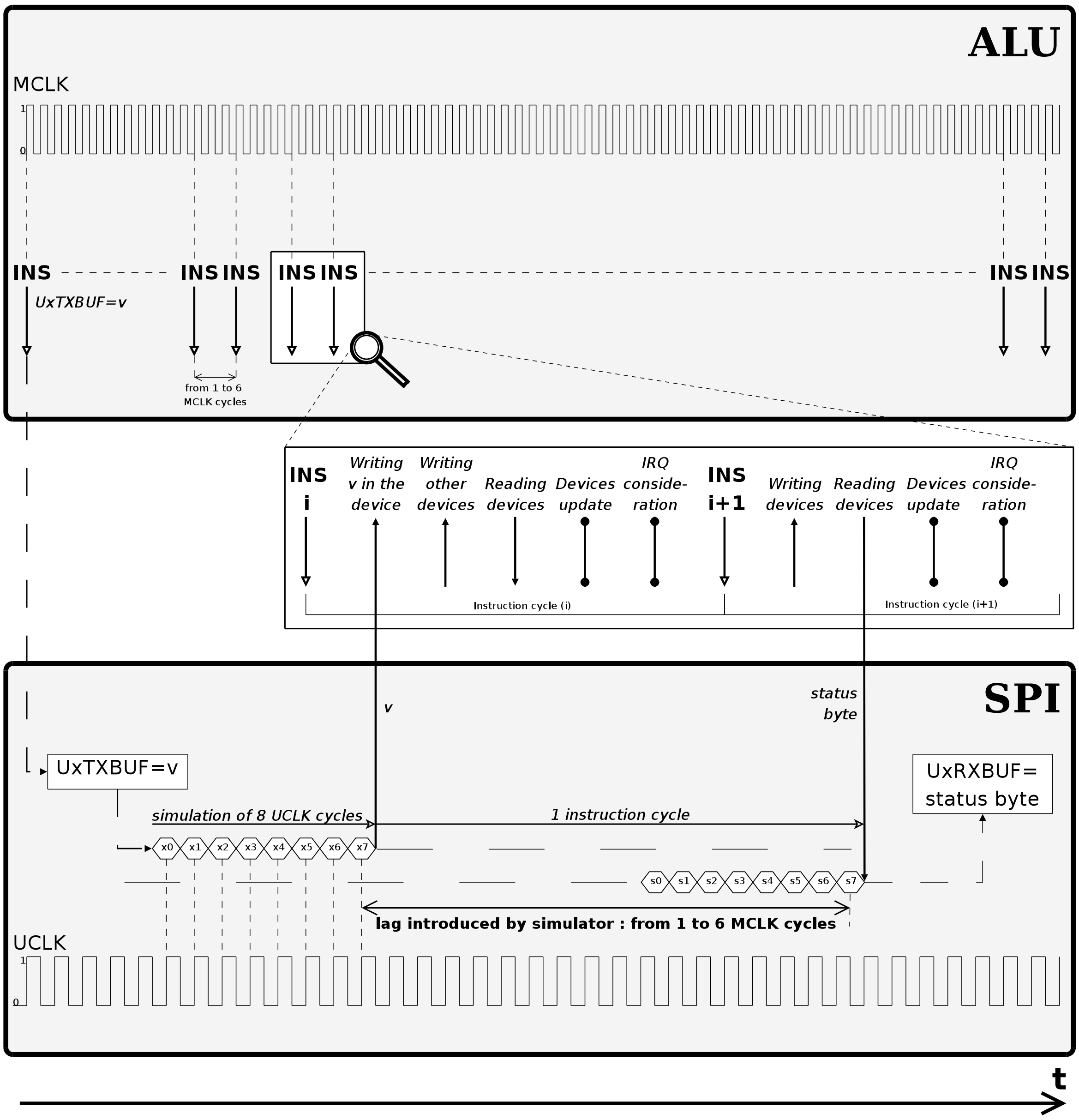

Figure 4.1: WSim SPI and device communication particular case

```
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

We may distinguish 2 special devices: the pseudo serial PTTY and the graphical interface for platform UIGFX. They are special in the meaning that they do not stand for a hardware component. Nevertheless their implementation is quite the same as a standard device, especially their API functions (please see section 4.2.1 page 17 for more details). They have to be initialised and updated in the platform file like other devices indeed.

4.3.1 Pseudo TTY

The PTTY (Pseudo TTY) peripheral model is a special peripheral that can be used to connect a platform to a RS232 serial port. The model has the capability to open a full duplex communication stream with the operating system in order to connect the simulator to external tools.

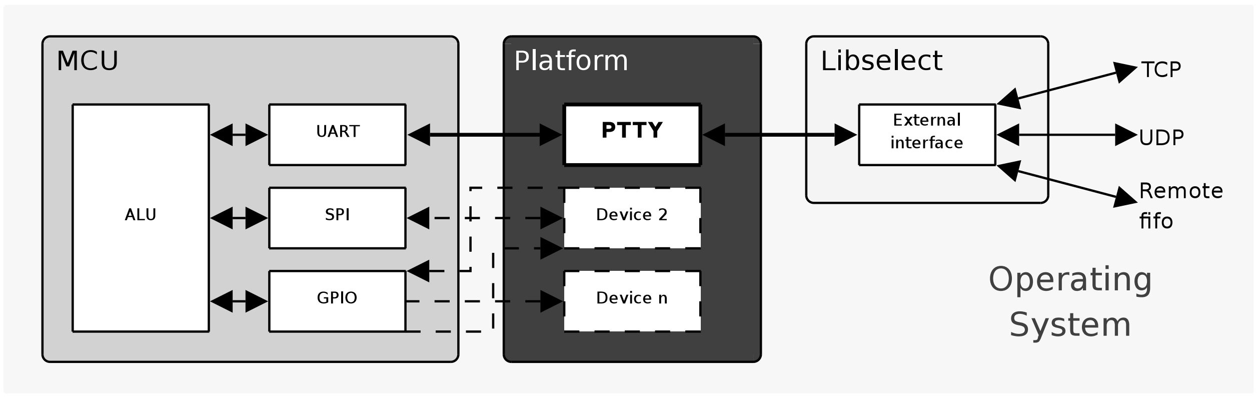

Figure 4.2: WSim PTTY device integration

The PTTY mainly makes the link between the platform and `libselect` functionalities.

- To send data to the OS: PTTY device calls `libselect_id_write()`.
- To get data from the OS: PTTY device calls `libselect_id_read()`.

- To delete the PTTY device: PTTY delete the `libselect` connection with the `libselect_id_close()` function.

These 3 additional functions are designed to avoid and handle with errors :

- `ptty_dummy_read()` : sets value and mask to 0.
- `ptty_dummy_write()` : writes to `/dev/null`.
- `ptty_libselect_callback()` : not used.

4.3.2 Graphical interface for platform UIGFX

Please refer to section 4.1.1 page 14 for some further informations.

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 be 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.

Note that the `wsim.log` file name may be changed into `NAMEOFYOURCHOICE.log` by using the option `--logfile=NAMEOFYOURCHOICE.log`. Moreover WSim `--logfile` option also supports `stdout` and `stdin` redirection (for instance `--logfile=stderr`).

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

PTTY = 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 = MSP430 SPI register used for reception

UxTXBUF = MSP430 SPI register used for transmission

6.2 Templates

6.2.1 SPI device

These templates are aimed to help you to develop your own SPI device model. You just have to replace `spidev` string by your device model name, and to implement the body of the functions you need. `spidev_dev.c` and `spidev_dev.h` are the minimal files to write a device model. However depending on its complexity, you may need additional files.

These source codes are also available in the `/devices/spidev` folder.

spidev_dev.c

```
/**
 *  \file   spidev_dev.c
 *  \brief  SPI device example
 *  \author Antoine Fraboulet
 *  \date   2009
 **/

#include <stdio.h>
#include <stdlib.h>
#include <string.h>

#include "arch/common/hardware.h"
#include "devices/spidev/spidev_dev.h"
#include "src/options.h"

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

#define NAME      "spidev"

tracer_id_t TRACER_SPIDEV_STATE;
tracer_id_t TRACER_SPIDEV_STROBE;

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

#ifdef DEBUG
#define MSG_DEVICES       2
#define DEBUG_ME_HARDER
#define HW_DMSG_SPI(x...) VERBOSE(MSG_DEVICES,x)
#else
#define HW_DMSG_SPI(x...) do {} while(0)
#endif

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

struct spidev_t
{
  uint8_t select_bit;           /* chip select   */
  uint8_t hold_bit;             /* hold          */
  uint8_t write_protect_bit;    /* write protect */

  /* data just written */
  uint8_t data_buffer;
  uint8_t data_buffer_ok;

  /* data to be read */
  uint8_t data_ready;
  uint8_t data_val;

  /* busy timing */
  uint64_t end_of_busy_time;

  /* clock pin : unused */
  uint8_t clock;
};

#define SPIDEV_DATA        ((struct spidev_t*)(machine.device[dev].data))

/***************************************************/
/** Flash external entry points *******************/
/***************************************************/

int  spidev_reset       (int dev);
int  spidev_delete      (int dev);
int  spidev_power_up    (int dev);
int  spidev_power_down  (int dev);
void spidev_read        (int dev, uint32_t *mask, uint32_t *value);
void spidev_write       (int dev, uint32_t  mask, uint32_t  value);
int  spidev_update      (int dev);
int  spidev_ui_draw     (int dev);
void spidev_ui_get_size (int dev, int *w, int *h);
void spidev_ui_set_pos  (int dev, int  x, int  y);
void spidev_ui_get_pos  (int dev, int *x, int *y);

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

#define MAXNAME 1024

int spidev_add_options(int UNUSED dev_num, int dev_id, const char UNUSED *dev_name)
{
  if (dev_id >= 1)
    {
      ERROR("spidev: too much devices, please rewrite option handling\n");
      return -1;
    }

  return 0;
}
```

```
/**************************************************/
/**************************************************/
/**************************************************/

int spidev_device_size()
{
  return sizeof(struct spidev_t);
}

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

int spidev_device_create(int dev, int UNUSED id)
{
  machine.device[dev].reset         = spidev_reset;
  machine.device[dev].delete        = spidev_delete;
  machine.device[dev].power_up      = spidev_power_up;
  machine.device[dev].power_down    = spidev_power_down;

  machine.device[dev].read          = spidev_read;
  machine.device[dev].write         = spidev_write;
  machine.device[dev].update        = spidev_update;

  machine.device[dev].ui_draw       = spidev_ui_draw;
  machine.device[dev].ui_get_size   = spidev_ui_get_size;
  machine.device[dev].ui_set_pos    = spidev_ui_set_pos;

  machine.device[dev].state_size    = spidev_device_size();
  machine.device[dev].name          = NAME " example";

#if defined(DEBUG_ME_HARDER)
  HW_DMSG_SPI(NAME ": =================================== \n");
  HW_DMSG_SPI(NAME ": 0000 CHSW dddd dddd == MASK         \n");
  HW_DMSG_SPI(NAME ":      C              : Clock         \n");
  HW_DMSG_SPI(NAME ":       M             : MiSo          \n");
  HW_DMSG_SPI(NAME ":        S            : Chip Select   \n");
  HW_DMSG_SPI(NAME ":         W           : Write Protect \n");
  HW_DMSG_SPI(NAME ":           dddd dddd : SPI data      \n");
  HW_DMSG_SPI(NAME ": =================================== \n");
#endif

  TRACER_SPIDEV_STATE  = tracer_event_add_id(8, "state"    , NAME);
  TRACER_SPIDEV_STROBE = tracer_event_add_id(8, "function" , NAME);

  return 0;
}

/**************************************************/
/**************************************************/
/**************************************************/
```

```
int spidev_reset(int UNUSED dev)
{
  HW_DMSG_SPI(NAME ": device reset\n");
  return 0;
}

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

int spidev_delete(int UNUSED dev)
{
  HW_DMSG_SPI(NAME ": device delete\n");
  return 0;
}

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

int spidev_power_up(int UNUSED dev)
{
  HW_DMSG_SPI(NAME ": device power up\n");
  return 0;
}

int spidev_power_down(int UNUSED dev)
{
  HW_DMSG_SPI(NAME ": device power down\n");
  return 0;
}

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

/*
 * read is done only on a junk write
 *
 */

void spidev_read(int UNUSED dev, uint32_t *mask, uint32_t *value)
{
  *mask  = 0;
  *value = 0;

  if (*mask != 0)
    {
      HW_DMSG_SPI(NAME ": device write to mcu [val=0x%02x,mask=0x%04x] \n", *value, *mask);
    }
}

/**************************************************/
```

```
/****************************************************/
/****************************************************/

void spidev_write(int dev, uint32_t mask, uint32_t value)
{
  HW_DMSG_SPI(NAME ": device write from mcu value 0x%04x mask 0x%04x\n",value, mask);

  /***************************
   * Control pins. WRITE PROTECT
   ***************************/

  if (mask & SPIDEV_W) /* write protect netgated */
    {
      SPIDEV_DATA->write_protect_bit  = ! (value & SPIDEV_W);
      HW_DMSG_SPI(NAME ":    flash write protect W = %d\n",SPIDEV_DATA->write_protect_bit);
    }

  /***************************
   * Control pins. CLOCK
   ***************************/

  if (mask & SPIDEV_C) /* clock */
    {
      ERROR(NAME ":    clock pin should not be used during simulation\n");
      HW_DMSG_SPI(NAME ":    clock pin should not be used during simulation\n");
      SPIDEV_DATA->clock = (value >> SPIDEV_C_SHIFT) & 0x1;
    }
}

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

int spidev_update(int UNUSED dev)
{
  return 0;
}

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

int spidev_ui_draw      (int UNUSED dev)
{
  return 0;
}

void spidev_ui_get_size (int UNUSED dev, int *w, int *h)
{
  w = 0;
  h = 0;
}
```

```
void spidev_ui_set_pos  (int UNUSED dev, int UNUSED x, int UNUSED y)
{

}

void spidev_ui_get_pos  (int UNUSED dev, int *x, int *y)
{
  *x = 0;
  *y = 0;
}

/****************************************************/
/****************************************************/
/****************************************************/
```

spidev_dev.h

```
/**
 *  \file   spidev_dev.h
 *  \brief  SPI device example
 *  \author Antoine Fraboulet
 *  \date   2009
 **/

#ifndef SPIDEV_DEVICES_H
#define SPIDEV_DEVICES_H

#define SPIDEV_W_SHIFT  8 /* write protect */
#define SPIDEV_S_SHIFT  9 /* select        */
#define SPIDEV_M_SHIFT 10 /* MISO          */
#define SPIDEV_C_SHIFT 11 /* clock         */

#define SPIDEV_D  0x00ff                 /** data 8 bits           **/
#define SPIDEV_W  (1 << SPIDEV_W_SHIFT)  /** write protect negated **/
#define SPIDEV_S  (1 << SPIDEV_S_SHIFT)  /** chip select negated   **/
#define SPIDEV_M  (1 << SPIDEV_M_SHIFT)  /** miso                  **/
#define SPIDEV_C  (1 << SPIDEV_C_SHIFT)  /** clock                 **/

int  spidev_add_options   (int dev_num, int dev_id, const char *name);
int  spidev_device_size   ();
int  spidev_device_create (int dev_num, int dev_id);

#endif
```