Aery32 Reference Guide

Release

Muiku Oy

March 30, 2014

CHAPTER 1

Getting Started

Aery32 development board and framework is a starting point for AVR32-based projects. The framework provides a project structure and library enabling **rapid prototyping and development**. Aery32 aims to be both **professional** and **fun** that makes it ideal choise for R&D Engineers, Academics and Hobbyists.

The best way to get started is to follow the Quick Start from Aery32's homepage.

1.1 Installation

You don't have to install Aery32 Software Framework as you would do with regular software. Just download the framework, unzip it and start working with it.

Note: Aery32 Framework is developed with Git distributed version control system at GitHub.

1.2 Requirements

- Atmel AVR Toolchain
- Batchisp (for Windows) or DFU-programmer (for Mac OS X & Linux)

1.3 Basics of the embedded software

```
1  #include "board.h"
2  #include <aery32/all.h>
3
4  using namespace aery;
5
6  int main(void)
7  {
8      /*
9       * Put your application initialization sequence here. The default
10      * board_init() sets up the LED pin and the CPU clock to 66 MHz.
11      */
```

```
12      init_board();
13
14      /* All done. Turn the LED on. */
15      gpio_set_pin_high(LED);
16
17      for(;;) {
18          /* Put your application code here */
19
20      }
21
22      return 0;
23  }
```

Above you can see a basic embedded software coded by C++ programming language for Aery32. This piece of source code can be found from the `main.cpp` source file. The `main()` function at line 7 is the first function to execute when the program starts – thus it is called *main*. The `void` keyword inside the brackets of the function, tells that the function does not take any arguments. The main function hardly ever takes arguments in embedded software, so this is a very common situation.

The `int` keyword, before the main function, indicates that the function will return integer type variable. Again, in the real life embedded software, it is very common that there is no use for the return value of main. The return type has been specified here to be integer type, instead of making it `void`, only to keep the compiler happy. Otherwise the compiler would give a warning, which we do not want to see. For the sake of consistency the return value has been set zero at line 21, but the running application should never reach that far, or if it does, some serious error has occurred. Altough there is no use for the input arguments and the return value of the main function, the other functions, of course, may have arguments and can return values which are relevant.

1.3.1 Where to put the application code?

If the program should never reach the line 21, you might guess where the code of actual application is placed. Correct! It is placed inside of the infinite `for(;;)` loop. This loop goes on and on accomplish the code inside of it until the power is switched off. In this particular software there is only a comment line inside of the loop, so pretty much nothing is happening. What happens has been done at lines 10, 11 and 14. These functions will be executed only once, because those do not fall into the infinite for-loop. Furthermore, these function calls comes with the Aery32 Software Framework. The first one initializes the board, which is pretty much about starting the 12 MHz crystal oscillator and then setting up the main clock to 66 MHz. The second function call initializes a general purpose pin named `LED` that is the pin `PC04` to be exact as defined at line 5. The `GPIO_OUTPUT` part of the line states that the `LED` pin will be an output. Making the pin high at line 13 turns the LED on, so in conclusion the job of this software is only to keep the LED burning.

CHAPTER 2

Project structure – where things go?

Aery32 Software Framework provides a complete project structure to start AVR32 development right away. You just start coding and adding your files. The default project directory structure looks like this:

```
projectname/
    aery32/
        ...
    board.cpp
    board.h
    main.cpp
    Makefile
    settings.h
```

It is intended that you work under the root directory most of the time as that is the place where you keep adding your .c, .cpp and .h source files. Notice that Aery32 Framework is a C/C++ framework and thus you can write your code in both C and C++. Just put the C code in .c files and C++ code in .cpp files.

The following subsections define each part of the default project structure in alphabetic order as they are listed above.

2.1 Aery32 library, `aery32/`

The directory called `aery32/` contains the source files of the Aery32 library. The archive of the library (.a file) appears in this directory after the first compile process. The `aery32/` subdirectory within the `aery32/` contains the header files of the library. Additionally, linker scripts, which are essential files to define the MCU memory structure are kept under this directory. Take a look at the `ldscripts/` subdirectory if you are curious.

Note: Although you can, you should not need to hassle with any file under this directory.

2.2 Main source file, `main.cpp`

The main.cpp source file contains the default main function where to start building your project. First the board.h header file has been included. This file includes your application specific function prototypes, which are defined in board.cpp. For your convenience a small `board::init()` function is provided by default. This function is called within the main function at line 15 and is the first function call. The second function call before the empty main loop sets the LED pin high.

```
1 #include <aery32/all.h>
2 #include "board.h"
```

```
3
4  using namespace aery;
5
6  #define LED AVR32_PIN_PC04
7
8  int main(void)
9  {
10     /*
11      * Put your application initialization sequence here. The default
12      * board initializer defines the LED pin as output and sets the CPU
13      * clock speed to 66 MHz.
14      */
15     board::init();
16     gpio_set_pin_high(LED);
17
18     for(;;) {
19         /* Put your application code here */
20
21     }
22
23     return 0;
24 }
```

2.3 Board specific functions, `board.h` and `.cpp`

All board specific functions and macro definitions should be placed in `board.h` header file. A macro definition is for example the following pin declaration

```
#define LED    AVR32_PIN_PC04
```

By doing this you don't need to remember which pin the LED was connected when you want to switch it on. So instead of using this

```
aery::gpio_set_pin_high(AVR32_PIN_PC04);
```

you can use this

```
aery::gpio_set_pin_high(LED);
```

It's intended that you define all your board related functions in board.h and then implement those in board.cpp. *Example programs* coming with the framework are built in one file with the main function in purpose, but when used in real application those should be refactored into board.h and .cpp. For example, consider that you had a device which to communicate via SPI. To take an advance of the board abstraction you could write the following board specific function in board.h

```
inline uint8_t board::write_to_device(uint8_t byte)
{
    return aery::spi_transmit(spi0, 2, byte);
}
```

See how the above function abstracts which SPI peripheral number and slave select your device is connected.

2.3.1 Default board initializer

The default board initializer function, `board::init()`, can be found from the `board.cpp` source file. The prototype of this function is declared in `board.h`.

Here's what it basicly does by default

- Sets all GPIO pins inputs
- Defines LED pin as output
- Starts the external oscillator `OCS0`
- Sets the chip's master (or main) clock frequency to its maximum, which is 66 MHz

If you like to change the master clock frequency or want to change the way how the board is initialized, board::init() is the place where to do it.

Note: All board related functions should use a namespace `board` to not introduce any name collision with other functions added into the project.

2.4 Build system, `Makefile`

Makefile contains all the make recipes for compiling the project and uploading the compiled binary to the board. See more detailed instructions from the *build system* section.

Note: Generally Makefiles don't have a file postfix like `.cpp` and it's a common practice to start its name with capital M.

2.5 Project wide settings, `settings.h`

This file is provided to GCC via `-include` allowing you to set project wide global setting definitions. Aery32 Framework is also aware of these definitions. For example, to get the delay functions work properly you have to define the correct CPU frequency, `F_CPU`, in this file:

```
#define F_OSC0 12000000UL
#define F_OSC1 16000000UL
#define F_CPU  66000000UL
```

Note: If you choose to change the board CPU frequency, make sure to redefine these or otherwise delay functions won't work as expected.

CHAPTER 3

The build system

Aery32 Framework comes with a powerful Makefile that provides a convenient way to compile the project. It also has targets for chip programming.

To compile the project just command:

```
make
```

When the compilation process is done, binaries will appear under the project's root (`aery32.hex` and `aery32.elf`). These two files are used in chip programming or program uploading, or chip flashing. Whatever you like to call it. In addition to the binaries, assembly listing and file mapping files, have been created. `aery32.lst` and `aery32.map`, respectively.

The program size is also showed at the end of the compile, like this:

```
Program size:
   text    data     bss     dec     hex filename
   3700    1340   64196   69236   10e74 aery32.elf
      0    5040       0    5040    13b0 aery32.hex
SDRAM usage: 5512 bytes, 8.41064 %
```

Here the program size has been given in separate sections and the static RAM usage has been calculated.

`.text` correspond the FLASH usage and `.data + .bss` indicates the RAM allocation. `.bss` section is the place where the initialized data is copied at runtime during the startup.

Dynamic RAM usage of stack and heap sections cannot be calculated before hand so make sure that there are always reasonable amount of RAM available for heap. Default stack size is 4 kB and you just have to know if it's enough. You can increase the stack size from linker script if needed.

Note: By default the project is compiled with -O2 optimization. If you run into troubles and your program behaves unpredictly on the chip, first try some other level of optimization

```
make reall COPT="-O0 -fdata-sections -ffunction-sections"
```

3.1 Chip programming

To program the chip with the compiled binary type:

```
make program
```

At this point the Makefile attempts to use batchisp in Windows and dfu-programmer in Linux, so make sure you have those installed. If you also want to start the program immediately type:

```
make program start
```

or in shorter format:

```
make programs
```

If you want to clean the project folder from the binaries call:

```
make clean
```

To recompile all the project files call:

```
make re
```

The above command recompiles only the files from the project root. It does not recompile the Aery32 library, because that would be ridiculous. If you also want to recompile the Aery32 library use `make reall`. There's also `cleanall` that cleans the Aery32 folder in addition to the project's root.

3.2 How to add new source files to the project

New sources files (.c, .cpp and .h) can be added straight into the project's root and the build system will take care of those. If you like to separate your source code into folders, for example, into subdirectory called `my/` then you have to edit the Makefile slightly. After creating the directory, open the Makefile and find the line:

```
SOURCES=$(wildcard *.cpp) $(wildcard *.c)
```

Edit this line so that it looks like this:

```
SOURCES=$(wildcard *.cpp) $(wildcard *.c) $(wildcard my/*.c) $(wildcard my/*.cpp)
```

3.3 Compile with debug statements

There are two additional make targets that helps you in debugging, `make debug` and `make qa`. The most important is:

```
make debug
```

This compiles the project with the debug statements. Use this when you need to do in-system debugging.

For quality assurance check use:

```
make qa
```

This target compiles the project with more pedantic compiler options. It's good to use this every now and then. Particularly when there are problems in your program. This target can also tell you, if your inline functions are not inlined for some reason.

CHAPTER 4

Example programs, `examples/`

Aery32 Framework comes with plenty of example programs, which are placed under the `examples/` directory. Every file is an independent program that does not need other files to work. So it should work out of box if you just replace the default `main.cpp` with the example.

Note: You may remove the `examples/` directory from your project if you don't need it.

For example, let's see how the LED toggling demo looks like

```
1  #include "board.h"
2  using namespace aery;
3
4  int main(void)
5  {
6      board::init();
7      for(;;) {
8          gpio_toggle_pin(LED);
9          delay_ms(500);
10     }
11
12     return 0;
13 }
```

To test this example open the Command Prompt and command:

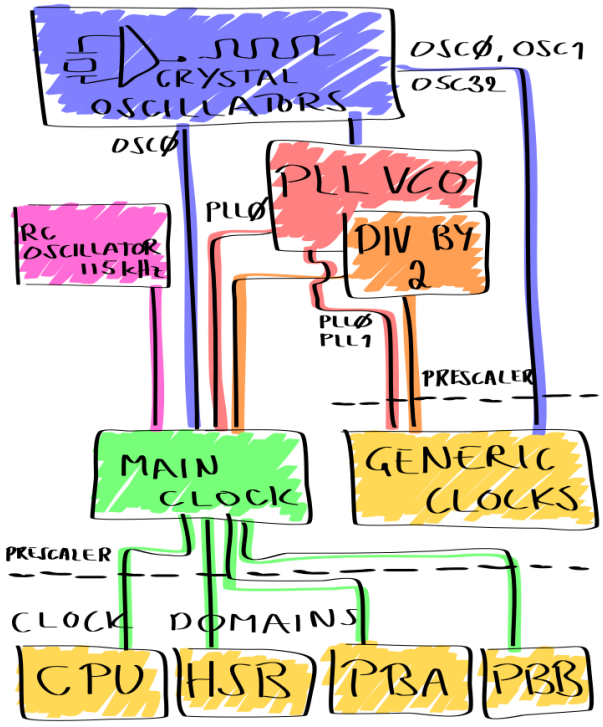
```
cp examples\toggle_led.cpp main.cpp
make programs
```

The following lines of commands overwrite the present `main.cpp` file from the project root with the example, compiles the project and uploads the new binary to the development board. The program starts running immediately and you should see how the LED pin blinks every half a second.

Note: The quickest way to access Command Prompt in Windows is to press Windows-key and R (Win+R) at the same time, and type cmd.

CHAPTER 5

UC3A0/1 system clocks described



UC3A0/1 chips have quite a complex clock system as it can be seen from the figure. At start-up the UC3 runs at its internal RC oscillator that's 115 kHz. This means that the Main clock has been clocked from this RC oscillator and that the CPU frequency equals to 115 kHz. That's because by default the clock domain prescaler has been disabled. This also means that all the other clock domains, that are HSB, PBA and PBB, runs at 115 kHz frequency as well.

5.1 Main clock means the CPU clock

It's important to understand that all the clock domains are derived from the Main (or Master) clock. The main clock can be clocked from couple of other sources in addition to the RC oscillator. It can be clocked from the OSC0 or PLL0. PLL0 clock frequency can equal to its VCO frequency or VCO/2. As the PLL can have a very high clock frequency it's important to set the prescaler of the clock domains properly. The maximum frequency is 66 MHz. It's also good to know that **CPU and HSB domains must equal each other and that the PBA and PBB frequencies have to be**

smaller than or equal to CPU.

PLLs can be driven only from the external crystal oscillators and its output can be used, in addition to the Main clock, for the Generic clocks as well. Generic clocks are multi-purpose clocks. You can use those, for example, to clock your external devices by connecting the clock pin of the device to the proper GPIO pin that can act as an output for the Generic clock. The internal USB and ABDAC peripherals get their clock from the Generic clock module too.

Read more about how to operate with these clocks from the Power Manager within the Modules section.

CHAPTER 6

Delay functions, `#include <aery32/delay.h>`

There are three convenience delay functions that are intented to be used for short delays, from microseconds to milliseconds.

```
delay_us(10); /* wait 10 microseconds */
delay_ms(500); /* wait half a second */
```

These functions are dependent on the CPU clock frequency that has to be provided via `F_CPU` definition before the `delay.h` header file has been included, like this

```
#define F_CPU 66000000UL
#include <aery32/delay.h>
```

If `F_CPU` is not defined, you can still use a low level delay function that takes the number of master clock cycles as a parameter

```
delay_cycles(1000); /* wait 1000 master clock cycles */
```

Hint: You can make smaller than 1 us delays with `delay_cycles()`.

6.1 Use RTC for long delays

Realtime counter (RTC) is better for long time delays or countdowns that last minutes, hours, days or even months or years. For example, to toggle the LED every second one can use `rtc_delay_cycles()` function.

```
#include <aery32/gpio.h>
#include <aery32/rtc.h>
#include "board.h"

using namespace aery;

int main(void)
{
    init_board();
    gpio_init_pin(LED, GPIO_OUTPUT);

    rtc_init(0, 0xffffffff, 0, RTC_SOURCE_RC);
    rtc_enable(false);

    for(;;) {
        gpio_toggle_pin(LED);
```

```
17          rtc_delay_cycles(57500);
18      }
19
20      return 0;
21  }
```

The application above first initialize the RTC to start counting from zero to 4 294 967 295, which is `0xffffffff` in hexadecimal. The third parameter is a prescaler for the RTC clock, which was set to be the in-built RC clock. The RC clock within the AVR32 UC3A1 works at 115 kHz frequency, but it's frequency is always divided by two when used for RTC. This means that the counter will be incremented every 1/(115000/2) second (approx. every 17.4 us).

When initialized, the RTC has to be enabled before it starts counting. This has been done by calling the `rtc_enable()` function. The first parameter indicates whether the interrupts are enabled too. Here we did not use those and thus false was passed as a parameter. However, when much longer delays or time calculations are desired, interrupts should be used. Below is an example program that toggles the LED every minute through the RTC interrupt event function, `isrhandler_rtc()`.

```
1   #include <aery32/gpio.h>
2   #include <aery32/intc.h>
3   #include <aery32/rtc.h>
4   #include "board.h"
5
6   using namespace aery;
7
8   void isrhandler_rtc(void)
9   {
10      gpio_toggle_pin(LED);
11      rtc_clear_interrupt(); /* Remember to clear RTC interrupt */
12  }
13
14  int main(void)
15  {
16      init_board();
17      gpio_init_pin(LED, GPIO_OUTPUT|GPIO_HIGH);
18
19      rtc_init(0, 60*115000/2, 0, RTC_SOURCE_RC);
20
21      intc_init();
22      intc_register_isrhandler(&isrhandler_rtc, 1, 0);
23      intc_enable_globally();
24
25      rtc_enable(true);
26
27      for(;;) {
28      }
29
30      return 0;
31  }
```

CHAPTER 7

String functions, `#include <aery32/string.h>`

Generally you should use sprintf/sscanf family of functions to do string manipulations but sometimes it would be nice to use some smaller alternatives to save memory. In addition, you may like to avoid using these standard functions because of some nasty side effects like stack overflown etc.

7.1 Integer number to string

Use this snippet to convert integer number to string

```
char str[10] = "";
itoa(100, str);
```

`itoa()` returns the pointer to the `str` so you can use it with print functions. If you like to know how many characters were written in conversion, you can save the number into additional variable, like this

```
int n;
char str[10] = "";
itoa(100, str, &n);
```

`utoa()` works similarly with `itoa()` but only for unsigned integers.

7.2 Double to string

This example converts pi to string with 8 decimal numbers.

```
char str[10] = "";
dtoa(3.14159265, 8, str);
```

CHAPTER 8

Contributor's guide

The development of Aery32 Framework happens in GitHub, which is a web-based hosting service that use the Git revision control system. To contribute the code you have to have a GitHub account. After that you can fork Aery32 repository and start contribute by sending pull request.

GitHub has a good beginners guide about how to set up Git in Windows.

8.1 Sending a pull request (creating a patch)

Fixes are sent as pull requests. Before you start fixing the Aery32 library, create a new branch and code your patch in this new branch. If there is a reported issue you are fixing, name the branch according to the GitHub issue number, e.g. gh-02.

The benefit of this approach is that you can have plenty of fixes which are isolated from each others, especially from the master branch.

8.2 Coding standards

Follow Linux kernel coding style.

```
#include <avr32/io.h>

typedef struct Foo Foo;
typedef struct Bar Bar;

struct Foo {
    Bar *bar;
};

struct Bar {
    Foo *foo;
};

/**
 * Initializes io pins (define the function briefly at the first line)
 * \param allinput If true all pins initialized as inputs
 *
 * More detailed description comes here. Remember to use param names
 * within the function prototypes too.
 */
```

```
void init_io(bool allinput);

void init_io(bool allinput)      /* Layout functions like this */
{
    int i;

    /* Use space between the syntactic keywords and opening parenthesis */
    for (i = 0; i < 2; i++) {
        AVR32_GPIO.port[i].gpers = 0xffffffff;

        if (allinput == 0)      /* Do not use curly braces if not needed */
            AVR32_GPIO.port[i].oders = 0xffffffff;
        else
            AVR32_GPIO.port[i].oderc = 0xffffffff;
    }
}

int main(void)
{
    char *c;
    Foo foo;
    Bar bar;

    bar.foo = &foo;

    /*
     * This is multiline comment that reminds you not to use compound literals
     * in Aery32 library, because avr32-g++ does not support those. What's
     * compound literal? This is: bar = (Bar) {.foo = &foo};
     */

    for (;;) {      /* This is how infinite loops are written */
    }

    return 0;
}
```

8.3 Writing the documentation

The documentation is constructed by Sphinx. Sphinx is a Python documentation generator but works fine for C as well.

The source files of this documentation are located at the separate GitHub repository, https://github.com/aery32/aery32-refguide. To build local html version of this documentation use make:

```
make html
```

The following commands assume you have Sphinx installed – if not, see the installation instructions below. Now browse to `source/` directory and open the `index.rst`. This is the master document serving as a welcome page and "*table of contents tree*". To edit these source files just open the file in your favorite editor and be sure to edit in UTF-8 mode. To understand reSt syntax start from http://sphinx.pocoo.org/rest.html.

8.3.1 Installing Sphinx

In Windows

Case 1: I do have Python already installed

If you do have Python installed already, then you likely have setuptools installed as well. In this case install Sphinx with easy_install. Fire your command prompt (Win+R cmd) and command:

```
easy_install -U Sphinx
```

Otherwise follow steps below to install Python first and then Sphinx.

Case 2: I don't have Python installed

Note: We do not install setuptools here and thus do not use easy_install to install Sphinx. However you will get it installed along Sphinx installer and it is recommended to use it later when installing other Python packages.

- Create temporary directory (e.g. myfoo) where to download the following things:
  - Python 2.7.x from http://python.org/download/
  - Sphinx 1.1.2 from http://pypi.python.org/pypi/Sphinx
- When the both download processes have been completed, you should have these two files:
  - `python-2.7.2.msi` or `python-2.7.2.amd64.msi` if you downloaded 64-bit version
  - `Sphinx-1.1.2.tar.gz`
- First install Python by double clicking Python installer
- After successful installation of Python, untar `Sphinx-1.1.2.tar.gz` into temporary directory
  - The exctarction process creates the `Sphinx-1.1.2` directory, change to that directory and double click setup to install Sphinx
  - Once the Sphinx installation is complete, you will find sphinx-xxx executables in your Python Scripts subdirectory, `C:\Python27\Scripts`. Be sure to add this directory to your PATH environment variable. As you can see, this directory includes now also easy_install executable, which you should use later to install other Python packages.
- You can now remove the temporary directory

CHAPTER 9

Module functions

Module functions are library components that operate straightly with the MCU's internal peripheral modules.

9.1 Naming conventions of module functions

Every module function has its own namespace according to the peripheral name. For example, Power Manager has module namespace of `pm_`, Serial Peripheral Interface falls under the `spi_` namespace and so on. To use the module just include its header file. So, for example, to include and use functions that operate with the Power Manager include `<aery32/pm.h>`.

A common calling order for module functions is following: 1) initialize, 2) do some extra setuping and after then 3) enable the module. In pseudo code these three steps would looks like this

```
module_init();
module_setup_something();
module_enable();
```

The init function may also look like `module_init_something()`, for example, the SPI can be initialized as a master or slave, so the naming convention declares two init functions for SPI module: `spi_init_master()` and `spi_init_slave()`.

If the module has been disabled, by using `module_disable()` function, it can be re-enabled without calling the init or setup functions. Most of the modules can also be reinitialized without disabling it before. For example, general clock frequencies can be changed by just calling the init function again – this is also the quickest way to change the frequency

```
pm_init_gclk(GCLK0, GCLK_SOURCE_PLL1, 1);
pm_enable_gclk(GCLK0);

/* Change the frequency divider */
pm_init_gclk(GCLK0, GCLK_SOURCE_PLL1, 6);
```

If you have read through the MCU datasheet, you may wonder why you cannot set all the possible settings with the initialization and setup functions. This is because these functions set sane default values for those properties. These default values should work for 80-90% of use cases. However, sometimes you may have to fine tune these properties to match your needs. This can be done by bitbanging the module registers after you have called the init or setup function. For example, the SPI chip select baudrate is hard coded to *MCK/255* within the `spi_setup_npcs()` function. To make SPI bus faster you can bitbang the *SCRB bit* within *CSRn register*, where *n* is the NPCS number.

```
spi_setup_npcs(spi0, 0, SPI_MODE1, 16);
spi0->CSR0.scbr = 32; /* SPI baudrate for the CS0 is now MCK/32 */
```

Note: Modules never take care of pin initialization, except GPIO module that's for this specific purpose. So, for example, when initializing SPI you have to take care of pin configuration!

```
#define SPI0_GPIO_MASK ((1 << 10) | (1 << 11) | (1 << 12) | (1 << 13))
gpio_init_pins(porta, SPI0_GPIO_MASK, GPIO_FUNCTION_A);
```

9.1.1 Global variables

Every module declares global shortcut variables to the main registers of the module. For example, the GPIO module declares `porta`, `b` and `c` global pointers to the MCU ports by default. Otherwise, you should have been more verbose and use `&AVR32_GPIO.port[0]`, `&AVR32_GPIO.port[1]` and `&AVR32_GPIO.port[2]`, respectively. Similarly, `pll0` and `pll1` declared in PM module provide quick access to MCU PLL registers etc.

Hint: As `porta`, `b` and `c` are pointers to the GPIO port, you can access its registers with arrow operator, for example, instead of using function `gpio_toggle_pin(AVR32_PIN_PC04)` you could have written `portc->ovrt = (1 << 4);` This is also way how you can set/unset/read/toggle multiple pins at once. Refer to the UC3A0/1 datasheet pages 175–177 for GPIO Register Map.

9.1.2 Error handling

All module functions will return -1 on general error. This will happen most probably because of invalid parameter values. Greater nagative return values have a specific meaning and a macro definition in the module's header file. For example, `flashc_save_page()` of Flash Controller may return -2 and -3, which have been defined with `E` prefixed names `EFLASH_PAGE_LOCKED` and `EFLASH_PROG_ERR`, respectively.

9.2 General Periheral Input/Output, #include <aery32/gpio.h>

To initialize any pin to be output high, there is a oneliner which can be used

```
gpio_init_pin(AVR32_PIN_PC04, GPIO_OUTPUT|GPIO_HIGH);
```

The first argument is the GPIO pin number and the second one is for options. For 100 pin Atmel AVR32UC3, the GPIO pin number is a decimal number from 0 to 69. Fortunately, you do not have to remember which number represent what port and pin. Instead you can use predefined aliases as it was done above with the pin PC04 (5th pin in port C if the PC00 is the 1st).

The available pin init options are:

- `GPIO_OUTPUT`
- `GPIO_INPUT`
- `GPIO_HIGH`
- `GPIO_LOW`
- `GPIO_FUNCTION_A`
- `GPIO_FUNCTION_B`
- `GPIO_FUNCTION_C`
- `GPIO_FUNCTION_D`
- `GPIO_INT_PIN_CHANGE`
- `GPIO_INT_RAISING_EDGE`
- `GPIO_INT_FALLING_EDGE`
- `GPIO_PULLUP`

- GPIO_OPENDRAIN
- GPIO_GLITCH_FILTER
- GPIO_HIZ

These options can be combined with the pipe operator (boolean OR) to carry out several commands at once. Without this feature the above oneliner should be written with two lines of code:

```
gpio_init_pin(AVR32_PIN_PC04, GPIO_OUTPUT);
gpio_set_pin_high(AVR32_PIN_PC04);
```

Well now you also know how to set pin high, so you may guess that the following function sets it low

```
gpio_set_pin_low(AVR32_PIN_PC04);
```

and that the following toggles it

```
gpio_toggle_pin(AVR32_PIN_PC04);
```

and finally it should not be surprise that there is a read function too

```
state = gpio_read_pin(AVR32_PIN_PC04);
```

But before going any further, let's quickly go through those pin init options. FUNCTION_A, B, C and D assing the pin to the specific peripheral function, see datasheet pages 45–48. INT_PIN_CHANGE, RAISING_EDGE and FALLING_EDGE enables interrupt events on the pin. Interrupts are trigged on pin change, at the rising edge or at falling edge, respectively. GPIO_PULLUP connects pin to the internal pull up resistor. GPIO_OPENDRAIN in turn makes the pin operate as an open drain mode. This mode is gererally used with pull up resistors to guarantee a high level on line when no driver is active. Lastly GPIO_GLITCH_FILTER activates the glitch filter and GPIO_HIZ makes the pin high impedance.

Usually you want to init several pins at once – not only one pin. This can be done for the pins that have the same port.

```
gpio_init_pins(porta, 0xffffffff, GPIO_INPUT); /* initializes all pins input */
```

The first argument is a pointer to the port register and the second one is the pin mask.

Note: Most of the combinations of GPIO init pin options do not make sense and have unknown consecuences.

9.2.1 Local GPIO bus

AVR32 includes so called local bus interface that connects its CPU to device-specific high-speed systems, such as floating-point units and fast GPIO ports. To enable local bus call

```
gpio_enable_localbus();
```

When enabled you have to operate with *local* GPIO registers. That is because, the convenience functions described above does not work local bus. To ease operating with local bus Aery32 GPIO module provides shortcuts to local ports by declaring lporta, b and c global pointers. Use these to read and write local port registers. For example, to toggle pin through local bus you can write

```
lporta->ovrt = (1 << 4);
```

Note: CPU clock has to match with PBB clock to make local bus functional

---

To disable local bus and go back to normal operation call

```
gpio_disable_localbus();
```

9.3 Power Manager, #include <aery32/pm.h>

Power Manager controls integrated oscillators and PLLs among other power related things. By default the MCU runs on the internal RC oscillator (115 kHz). However, it's often preferred to switch to the higher CPU clock frequency, so one of the first things what to do after the power up, is the initialization of oscillators. Aery32 Development Board has 12 MHz crystal oscillator connected to the OSC0. This can be started as

```
pm_start_osc(
    0,               /* oscillator number */
    OSC_MODE_GAIN3,  /* oscillator mode, see datasheet p.74 */
    OSC_STARTUP_36ms /* oscillator startup time */
);
pm_wait_osc_to_stabilize(0);
```

When the oscillator has been stabilized it can be used for the master/main clock

```
pm_select_mck(MCK_SOURCE_OSC0);
```

Now the CPU runs at 12 MHz frequency. The other possible source selections for the master clock are:

- `MCK_SOURCE_OSC0`
- `MCK_SOURCE_PLL0`
- `MCK_SOURCE_PLL1`

9.3.1 Use PLLs to achieve higher clock frequencies

Aery32 devboard can run at 66 MHz its fastest. To achieve these higher clock frequencies one must use PLLs. PLL has a voltage controlled oscillator (VCO) that has to be initialized first. After then the PLL itself can be enabled.

Important: PLL VCO frequency has to fall between 80–180 MHz or 160–240 MHz with high frequency disabled or enabled, respectively. From these rules, one can realize that the smallest available PLL frequency is 40 MHz (the VCO frequency can be divided by two afterwards).

```
pm_init_pllvco(
    pll0,            /* pointer to pll address */
    PLL_SOURCE_OSC0, /* source clock */
    11,              /* multiplier */
    1,               /* divider */
    false            /* high frequency */
);
```

- If `div > 0` then `f_vco = f_src * mul / div`
- If `div = 0` then `f_vco = 2 * mul * f_src`

The above initialization sets PLL VCO frequency of PLL0 to 132 MHz – that's `12 MHz * 11 / 1 = 132 MHz`. After then PLL can be enabled and the VCO frequency appears on the PLL output. Remember that you can now also divide VCO frequency by two.

```
pm_enable_pll(pll0, true  /* divide by two */); /* 132 MHz / 2 = 66 MHz */
pm_wait_pll_to_lock(pll0);
```

Finally one can change the master clock (or main clock) to be clocked from the PLL0 that's 66 MHz.

```
pm_select_mck(MCK_SOURCE_PLL0);
```

9.3.2 Fine tune the CPU and Periheral BUS frequencies

By default the clock domains, that are CPU and the Peripheral Busses (PBA and PBB) equal to the master clock. To fine tune these clock domains, the PM has a 3-bit prescaler, which can be used to divide the master clock, before it has been used for the specific domain. Using the prescaler you can choose the CPU clock between the OSC0 frequency and 40 MHz, that was the lower limit of the PLL. Assuming that the master clock was 66 MHz, the following function call changes the CPU and the bus frequencies to 33 MHz:

```
pm_setup_clkdomain(1, CLKDOMAIN_ALL);
```

The first parameter defines the prescaler value and the second one selects the clock domain which to set up. Here all the domains are set to equal. The formula is `f_mck / (2^prescaler)`. With the prescaler selection 0, the prescaler block will be disabled and the selected clock domain equals to the master clock that was the default setting.

The possible clock domain selections are

- `CLKDOMAIN_CPU`
- `CLKDOMAIN_PBA`
- `CLKDOMAIN_PBB`
- `CLKDOMAIN_ALL`

Important: PBA and PBB clocks have to be less or equal to CPU clock. Morever, the flash wait state has to been taken into account at this point. If the CPU clock is over 33 MHz, the Flash controller has to be initialized with one wait state, like this `flashc_init(FLASH_1WS, true)`. If the CPU clock speed is less or equal than 33 MHz, zero wait state is the correct setting for the flash.

Hint: You can combine the clock domain selections with the pipe operator, like this `CLKDOMAIN_CPU|CLKDOMAIN_PBB`. With this selection the PBA clock frequency won't be changed, but the CPU and PBB will be set up accordingly.

9.3.3 General clocks

PM can generate dedicated general clocks. These clocks can be assigned to GPIO pins or used for internal peripherals such as USB that needs 48 MHz clock to work. To offer this 48 MHz for the USB peripheral, you have to initialize either of the PLLs to work at 96 MHz frequency. As the PLL0 is commonly used for the master clock, PLL1 has been dedicated for general clocks. First initialize the VCO frequency and then enable the PLL

```
pm_init_pllvco(pll1, PLL_SOURCE_OSC0, 16, 1, true); /* f_pll1_vco = 192 MHz */
pm_enable_pll(pll1, true); /* f_pll1 = 96 MHz */
pm_wait_pll_to_lock(pll1);
```

After then init and enable the USB generic clock

```
pm_init_gclk(
    GCLK_USBB,        /* generic clock number */
    GCLK_SOURCE_PLL1, /* clock source for the generic clock */
    1                 /* divider */
);
pm_enable_gclk(GCLK_USBB);
```

- If `div > 0` then `f_gclk = f_src/(2*div)`

- If `div = 0` then `f_gclk = f_src`

There are five possible general clocks to be initialized:

- `GCLK0`
- `GCLK1`
- `GCLK2`
- `GCLK3`
- `GCLK_USBB`
- `GCLK_ABDAC`

`GCLK_ABDAC` is for Audio Bitstream DAC, `GCLK0`, `GCLK1`, etc. can be attached to GPIO pin, so that you can easily clock external devices. For example, to set generic clock to be at the output of GPIO pin, first init the desired GPIO pin appropriately and then enable the generic clock at this pin. You can do this, for example, to check that USB clock enabled above is correct

```
gpio_init_pin(AVR32_PIN_PB19, GPIO_FUNCTION_B);
pm_init_gclk(GCLK0, GCLK_SOURCE_PLL1, 1);
pm_enable_gclk(GCLK0);
```

Hint: Generic clock can be changed when its running by just initializing it again. You do not have to disable it before doing this and you do not have to enable it again.

9.3.4 Save power and use only the peripherals that you need

By default all modules are enabled. You might be interested in to disable modules you are not using. This can done via the peripheral clock masking. The following example disables clocks from the TWI, PWM, SSC, TC, ABDAC and all the USART modules

```
#define PBAMASK_DEFAULT 0x0F
pm->pbamask = PBAMASK_DEFAULT;
```

Remember to wait when the change has been completed

```
while (!(pm->isr & AVR32_PM_ISR_MSKRDY_MASK));
    /* Clocks are now masked according to (CPU/HSB/PBA/PBB)_MASK
     * registers. */
```

9.3.5 How much is the clock?

Sometimes the current clock frequencies has to be checked programmatically. To get the main clock use the `pm_get_fmck()` function

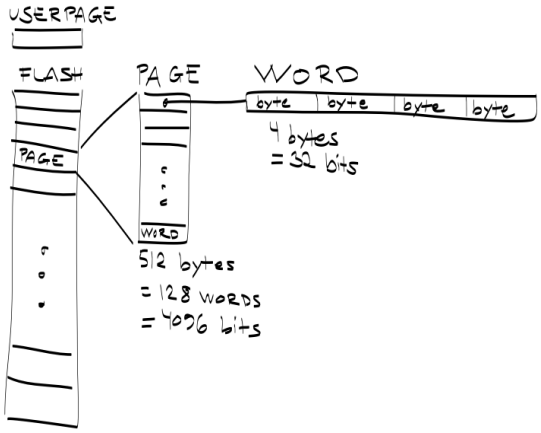
```
main_hz = pm_get_fmck();
```

Respectively, the clock domains can be fetched like this

```
cpu_hz = pm_get_fclkdomain(CLKDOMAIN_CPU);
pba_hz = pm_get_fclkdomain(CLKDOMAIN_PBA);
pbb_hz = pm_get_fclkdomain(CLKDOMAIN_PBB);
```

These functions assume that OSC0 and OSC1 frequencies are 12 MHz and 16 MHz, respectively. If other oscillator frequencies are used, change the default values by editing `CXXFLAGS` in `aery32/Makefile`.

9.4 Flash Controller, #include <aery32/flashc.h>



Flash Controller provides low-level access to the chip's internal flash memory, whose structure has been sketched in the figure above. The init function of the Flash Controller sets the flash wait state (zero or one, `FLASH_0WS` or `FLASH_1WS`, respectively). The last param of the init function enables/disables the sense amplifiers of flash controller.

```
flashc_init(FLASH_1WS, true);
```

Warning: Setting up the correct flash wait state is extremely important! Note that this has to be set correctly even if the flash read and write operations, described below, are not used. If CPU clock speed is higher than 33 MHz you have to use one wait state for flash. Otherwise you can use zero wait state.

Warning: The uploaded program is also stored into the flash, so it is possible to overwrite it by using the Flash controller. The best practice for flash programming, is starting from the top. `FLASH_LAST_PAGE` macro definition gives the number of the last page in the flash. For 128 KB flash this would be 255.

Warning: According to Atmel UC3A1's internal flash supports approximately 100,000 write cycles and it has 15-year data retention. However, you can easily make a for-loop with 100,000 writes to the same spot of flash and destroy your chip in a second. So be careful!

9.4.1 Read and write operations

Flash memory is accessed via pages that are 512 bytes long. This means that you have to make sure that your page buffer is large enough to read and write pages, like this

```
#include <cstring>

char buf[512];
flashc_read_page(FLASH_LAST_PAGE, buf); /* Read the last page to separate page buffer */
strcpy(buf, "foo");                     /* Save string "foo" to page buffer */
flashc_save_page(FLASH_LAST_PAGE, buf); /* Write page buffer back to flash */
```

You can also read and write values with different types as long as the page buffer size is that 512 bytes. For example like this

```
extern "C" #include <inttypes.h>
```

```
uint16_t buf16[256];
uint32_t buf32[128];
```

9.4.2 Page locking

The flash page can be locked to prevent the write and erase sequences. To lock the first page (page number 0), call

```
flashc_lock_page(0);
```

Locking is performed on a per-region basis, so the above statement does not lock only page zero, but all pages within the region. There are 16 pages per region so the above command locks also pages 1-15.

To unlock the page, call

```
flashc_unlock_page(0);
```

You can also use, `flashc_lock_page_region()` and `flashc_unlock_page_region()`. to lock and unlock pages by region. Furthermore, there is a function to check if the page is empty

```
flashc_isempty(0);
```

9.4.3 User page

The User page is an additional page, outside the regular flash array, that can be used to store various data, like calibration data and serial numbers. This page is not erased by regular chip erase. The User page can only be read and write by proprietary commands, which are

```
uint8_t buf[512];
flashc_read_userpage(buf);
```

and

```
flashc_save_userpage(buf);
```

To check whether the user page is empty or not call

```
flashc_userpage_isempty();
```

> **Warning:** Aery32 board ships with the preprogrammed bootloader, which uses the configuration data saved to the first word (32 bits) of user page. This configuration data includes the pin configuration for slide switch – the switch that you have been using to vary between bootloader and program modes. So to keep the board slide switch working just do not alter the first word of user page.

9.4.4 General purpose fuse bits

You can read all 32 fuse bits into 32 bit variable by using the following command

```
uint32_t fusebits;
fusebits = flashc_read_fusebits();
```

To write one bit true or false use this:

```
flashc_write_fusebit(uint16_t fusebit, bool value);
```

You can also write fuse bits by a byte at a time, like this

```
flashc_write_fusebyte(0, 0xff);
flashc_write_fusebyte(1, 0xff);
flashc_write_fusebyte(2, 0xff);
flashc_write_fusebyte(3, 0xff);
```

Now all fuse bits are written to 1. The first parameter is the byte address that can be 0-3 in a 32-bit word and the second one is the byte value.

9.5 Interrupt Controller, #include <aery32/intc.h>

Before enabling interrupts define and register your interrupt service routine (ISR) functions. First write ISR function as you would do for any other functions

```
void myisr_for_group1(void) {
    /* do something */
}
```

Then register this function

```
intc_register_isrhandler(&myisr_for_group1, 1, 0);
```

Here the first parameter is a function pointer to your `myisr_for_group1()` function. The second parameter defines the which interrupt group calls this function and the last one tells the priority level.

Hint: Refer Table 12-3 (Interrupt Request Signal Map) in datasheet page 41 to see what peripheral belongs to which group. For example, RTC belongs to group 1.

When all the ISR functions have been declared it is time to initialize interrupts. Use the following init function to do all the magic

```
intc_init();
```

After initialization you can enable and disable interrupts globally by using these functions

```
intc_enable_globally();

intc_disable_globally();
```

9.6 Analog-to-Digital Converter, #include <aery32/adc.h>

AVR32 UC3A0/1 microcontrollers have eight 10-bit analog-to-digital converters. The maximum ADC clock frequency for the 10-bit precision is 5 MHz and for 8-bit precision it's 8 MHz. This sampling frequency is related to the frequency of the Peripheral Bus A (PBA), so take care when setting ADC clock prescaler values. Skip to example.

9.6.1 Initialization

```
adc_init(
    7,    /* prescal, adclk = pba_clk / (2 * (prescal+1)) */
    true, /* hires, 10-bit (false would be 8-bit) */
    0,    /* shtim, sample and hold time = (shtim + 1) / adclk */
```

```
    0     /* startup, startup time = (startup + 1) * 8 / adclk */
);
```

The initialization statement given above, uses the prescaler value 7, so if the PBA clock was 66 MHz, the ADC clock would be 4.125 MHz. After initialization, you have to enable the channels that you like to use for the conversion. For example, to enable channel three call

```
adc_enable(1 << 3);
```

9.6.2 Reading the conversion

Now you can start the conversion. Be sure to wait that the conversion is ready before reading the conversion value.

```
uint16_t result;

adc_start_cnv();
while (adc_isbusy(1 << 3));
result = adc_read_cnv(3);
```

If you only want to read the latest conversion, whatever was the channel, you can omit the channel mask for the busy function and read the conversion with another function like this

```
while (adc_isbusy());
result = adc_read_lastcnv();
```

9.6.3 ADC hardware triggers

To setup the ADC hardware trigger, call `adc_setup_trigger()` after init

```
adc_setup_trigger(EXTERNAL_TRG);
```

Other possible trigger sources, that can be used for example with the Timer/Counter module, are

- `INTERNAL_TRG0`
- `INTERNAL_TRG1`
- `INTERNAL_TRG3`
- `INTERNAL_TRG4`
- `INTERNAL_TRG5`

Note: You always have to call `adc_start_cnv()` individually for every started conversion. If you suspect that your conversions may have overrun, you can check this with the `adc_has_overrun(chamask)` function. If you omit the channel mask input param, all the channels will be checked, being essentially the same than calling `adc_has_overrun(0xff)`.

9.7 Pulse Width Modulation, #include <aery32/pwm.h>

Start by initializing the PWM channel which you want to use

```
pwm_init_channel(2, MCK);
```

The above initializer sets channel's two PWM frequency equal to the main clock and omits the duration and period for default values. The default values for the duration and period are 0 and 0xFFFFF, respectively. If you like to start the channel with different values, you could have defined those too like this

```
pwm_init_channel(2, MCK, 50, 100);
```

This gives you duty cycle of 50% from start. The maximum value for both the duration and the period is 0xFFFFF. It is also worth noting that when the period is set to its maximum value, the channel's duty cycle can be set most accurately.

The above initializers set the channel's frequency equal to the main clock. The other possible frequency selections are

- `MCK_DIVIDED_BY_2`
- `MCK_DIVIDED_BY_4`
- `MCK_DIVIDED_BY_8`
- `MCK_DIVIDED_BY_16`
- `MCK_DIVIDED_BY_32`
- `MCK_DIVIDED_BY_64`
- `MCK_DIVIDED_BY_128`
- `MCK_DIVIDED_BY_256`
- `MCK_DIVIDED_BY_512`
- `MCK_DIVIDED_BY_1024`
- `PWM_CLKA`
- `PWM_CLKB`

`PWM_CLKA` and `PWM_CLKB` are two extra PWM clock sources. The difference to other sources is an additional linear divider block that comes after the MCK prescaler. To initialize the divider block for the `PWM_CLKA` and `PWM_CLKB` call

```
pwm_init_divab(MCK, 10, MCK_DIVIDED_BY_2, 10);
```

Now `PWM_CLKA` has the frequency of *MCK / 10* Hz and `PWM_CLKB` is *MCK / 2 / 10* Hz. If you don't care about `CLKB`, you can omit the last two of the parameters like this

```
pwm_init_divab(MCK, 10);
```

Note: If the divider of `PWM_CLKA` or `PWM_CLKB` has been set zero, then the PWM clock will equal to the `MCK`, `MCK_DIVIDED_BY_2`, etc. Whatever was the chosen prescaler. So it does not make sense to set the divider of the extra PWM clock zero, because then you don't have any extra clock selection.

9.7.1 Setting up PWM mode

Before enabling the initialized PWM channel or channels, you may like to setup the channel mode to set PWM alignment and polarity

```
pwm_setup_chamode(2, LEFT_ALIGNED, START_HIGH);
```

The alignment (left or center, `LEFT_ALIGNED` and `CENTER_ALIGNED`, respectively) defines the shape of PWM function, see datasheet page 680. The polarity defines the polarity of the duty cycle. With `START_HIGH`, the duty cycle is 100% when *duration / period* of the PWM function gives 1. With `START_LOW` you would get 100% duty cycle when the *duration / period* is 0.

9.7.2 Enabling and disabling the PWM

PWM is enabled and disabled by channels. Several channels can be enabled at once to get synchronized output. To enable channels two and four call

```
pwm_enable((1 << 2)|(1 << 4));
```

Same goes for the disabling the channels. The following call will disable the channel two

```
pwm_disable(1 << 2);
```

The parameter of the enable and disable functions is a bitmask of the channels to be enabled or disabled. There is also function to check if the channel has been enabled already. The following snippet will do something if the channel two was already enabled

```
if (pwm_is_enabled(1 << 2)) {
    /* Do something */
}
```

9.7.3 Modulating the PWM output waveform

You can modulate the PWM output waveform when it is active by changing its duty cycle like this

```
pwm_update_dutycl(2, 0.5);
```

The above function call will update the channel's two duty cycle to 50%. In case you want to specify completely new values for the period and duration use these two functions

```
pwm_update_period(2, 0x1000);
pwm_update_duration(2, 0x10);
```

Furthermore, to keep PWM output at the desired state for the amount of periods, before changing its state again, use the wait function. This also allows you to do updates from the beginning of the next period and thus avoiding to overwrite the value too soon. For example, to wait 100 periods on channel two call

```
pwm_wait_periods(2, 100);
```

With the combination of the update functions and the wait function, you can make a smoohtly blinking LED, just like this

```
uint8_t channel = 2;
uint32_t duration = 0;
uint32_t period = 0x1000;

for (;;) {
    for (; duration < period; duration++) {
        pwm_update_duration(channel, duration);
        pwm_wait_periods(channel, 500);
    }
    for (; duration > 0; duration--) {
        pwm_update_duration(channel, duration);
        pwm_wait_periods(channel, 500);
    }
}
```

Note: Duration has to be smaller or equal to period.

9.8 Real-time Counter, #include <aery32/rtc.h>

Real-time counter is for accurate real-time measurements. It enables periodic interrupts at long intervals and the measurement of real-time sequences. RTC has to be init to start counting from the chosen value to the chosen top value. This can be done in this way

```
rtc_init(
    RTC_SOURCE_RC, /* source oscillator */
    0,             /* prescaler for RTC clock */
    0,             /* value where to start counting */
    0xffffffff     /* top value where to count */
);
```

The available source oscillators are:

- `RTC_SOURCE_RC` (115 kHz RC oscillator within the AVR32)
- `RTC_SOURCE_OSC32` (external low-frequency xtal, not assembled in Aery32 Devboard)

When initialized, remember to enable it too

```
rtc_enable(false);
```

The boolean parameter here, tells if the interrupts are enabled or not. Here the interrupts are not enabled so it is your job to poll RTC to check whether the top value has been reached or not.

9.9 Serial Peripheral Interface, #include <aery32/spi.h>

Aery32 includes two separate SPI buses, SPI0 and SPI1. To initialize SPI bus it is good practice to define pin mask for the SPI related pins. Refering to datasheet page 45, SPI0 operates from PORTA:

- PA07, NPCS3
- PA08, NPCS1
- PA09, NPCS2
- PA10, NPCS0
- PA11, MISO
- PA12, MOSI
- PA13, SCK

So let's define the pin mask for SPI0 with NPCS0 (Numeric Processor Chip Select, also known as slave select or chip select):

```
#define SPI0_GPIO_MASK ((1 << 10) | (1 << 11) | (1 << 12) | (1 << 13))
```

Next we have to assing these pins to the right peripheral function that is FUNCTION A. To do that use pin initializer from GPIO module:

```
gpio_init_pins(porta, SPI0_GPIO_MASK, GPIO_FUNCTION_A);
```

Now the GPIO pins have been assigned appropriately and we are ready to initialize SPI0. Let's init it as a master:

```
spi_init_master(spi0);
```

The only parameter is a pointer to the SPI register. Aery32 declares `spi0` and `spi1` global pointers by default.

Hint: If the four SPI CS pins are not enough, you can use CS pins in multiplexed mode (of course you need an external multiplexer circuit then) and expand number of CS lines to 16. This can be done by bitbanging PCSDEC bit in SPI MR register after the initialization:

```
spi_init_master(spi0);
spi0->MR.pcsdec = 1;
```

When the SPI peripheral has been initialized as a master, we still have to setup its CS line 0 (NPCS0) with the desired SPI mode and shift register width. To set these to SPI mode 0 and 16 bit, call the npcs setup function with the following parameters

```
spi_setup_npcs(spi0, 0, SPI_MODE0, 16);
```

The minimum and maximum shift register widths are 8 and 16 bits, respectively, but you can still *use arbitrary wide transmission*.

Hint: Chip select baudrate is hard coded to MCK/255. To make it faster you can bitbang the SCRB bit in the CSRX register, where X is the NPCS number:

```
spi_setup_npcs(spi0, 0, SPI_MODE0, 16);
spi0->CSR0.scbr = 32; /* baudrate is now MCK/32 */
```

Hint: Different CS lines can have separate SPI mode, baudrate and shift register width.

Now we are ready to enable SPI peripheral

```
spi_enable(spi0);
```

There's also function for disabling the desired SPI peripheral, `spi_disable()`.

To read and write data use the SPI transmit function

```
uint16_t rd;
rd = spi_transmit(spi0, 0, 0x55);
```

The above call to `spi_transmit` writes 0x55 to SPI0 using NPCS0 slave (or chip) select pin. Notice that `spi_transmit()` writes and reads the SPI bus simultaneusly. If you only want to read data, just ignore write data by sending dummy bits.

Here is the complete code for the above SPI initialization and transmission:

```
#include "board.h"
#include <aery32/gpio.h>
#include <aery32/spi.h>

using namespace aery;

#define SPI0_GPIO_MASK ((1 << 10) | (1 << 11) | (1 << 12) | (1 << 13))

int main(void)
{
    uint16_t rd; /* received data */

    init_board();

    gpio_init_pins(porta, SPI0_GPIO_MASK, GPIO_FUNCTION_A);
```

```
16      spi_init_master(spi0);
17      spi_setup_npcs(spi0, 0, SPI_MODE0, 16);
18      spi_enable(spi0);
19
20      for (;;) {
21          rd = spi_transmit(spi0, 0, 0x55);
22      }
23
24      return 0;
25  }
```

9.9.1 Sending arbitrary wide SPI data

`spi_transmit()` supports arbitrary wide SPI transmits through its optional last parameter named as `islast`. This param indicates if the chip select line should be left low or can be pulled high. Otherway around it tells whether the current transmission was the last one or not. Thus the param is called `islast`. By default `islast` is set true and the CS line rises immediately when the last bit has been written. If `islast` is defined false, CS line is left low for the next transmission that should occur immediately after the previous one. This feature allows SPI to operate with arbitrary wide shift registers. For example, to read and write 24 bit wide SPI data you can do this:

```
uint32_t rd;

spi_setup_npcs(spi0, 0, SPI_MODE0, 8);

rd = spi_transmit(spi0, 0, 0x55, false);
rd |= spi_transmit(spi0, 0, 0xf0, false) << 8;
rd |= spi_transmit(spi0, 0, 0x0f, true) << 16; /* Complete. Asserts the chip select */
```

9.10 Two-wire (I2C) Interface, #include <aery32/twi.h>

TWI interface is initialized as a master like this

```
twi_init_master();
```

By default the initializer sets the bus' SCL frequency to 100 kHz. The maximum supported frequency is 400 kHz. This can be set up with `twi_setup_clkwaveform()` after you have called the init function. For example, to set SCL to 400 kHz with 50% dutycycle, call

```
twi_setup_clkwaveform(1, 0x3f, 0x3f);
```

The first parameter is the clock divider. The second and third one define the dividers for the clock low and high states, respectively. Altering these divider values for clock high and low states you can modify the SCL waveform. However, you most likely want to set those equal to get 50% dutycycle for the clock.

Note: TWI module does not have an enable function as other modules. The module is enabled when TWI pins are initialized with GPIO module:

```
#define TWI_PINS ((1 << 29) | (1 << 30))
gpio_init_pins(porta, TWI_PINS, GPIO_FUNCTION_A | GPIO_OPENDRAIN);
```

Warning: Important! Don't forget to connect the appropriate size external pull-up resistors to SDA and SCL pins. Try for example 4k7 value resistors. The exact optimal value depends on the SCL and the parasitic capacitance of the bus.

9.10.1 Read and write operations

The read and write operations for a single byte works like this

```
uint8_t rd;

twi_write_byte(0x04);
twi_read_byte(&rd);
```

Both functions return the number of successfully written or read bytes. So for one byte read/write operation the return value would be 0 on error and 1 on success.

To read and write multiple bytes use `twi_read/write_nbytes()`, like this

```
uint8_t wd[3] = { 0x02, 0x04, 0x06 };
uint8_t rd[3];

twi_write_nbytes(wd, 3);
twi_read_nbytes(rd, 3);
```

9.10.2 Using internal device address

Both read and write functions can take an optional internal device address in their last param. Internal device address is the slave's internal register where data is written or read from. For example, the following snippet writes a byte `0x04` to the slave register (or in other words slave's internal address) `0x80`.

```
uint8_t byte = 0x04;
uint8_t iadr = 0x80;

twi_write_byte(byte, iadr);
```

When optional address has been given the same address is used in every read and write operations that follows the previous operation even if the address is omitted from the function call. To clear this behaviour, call `twi_clear_internal_address()`.

If you want to use a wider than 8-bit internal device addresses, you have to indicate the address lenght via additional third parameter. For example to use 2 bytes long address, you may call the write function like this

```
uint8_t byte = 0x04;
uint16_t iadr = 0x8080;

twi_write_byte(byte, iadr, 2);
```

The largest supported internal device address length is three bytes long.

9.11 Universal Sync/Asynchronous Receiver/Transmitter, #include <aery32/usart.h>

USART module can be used for several kind of communication. The RS-232 serial communication with a personal computer (PC) is likely the most common one.

Note: There's also a class driver for serial port communication. Skip to *Serial Port class driver*.

9.11.1 Initialization

To initialize the USART0 for serial communication call

```
usart_init_serial(usart0, USART_PARITY_NONE, USART_STOPBITS_1);
```

Serial communication initialized with no parity and 1 stop bits is the default setup for most of the devices, so you may omit the last two parameters if you like. Other options are `EVEN`, `ODD`, `MARKED` and `SPACE` for parity, and `1p5` and `2` for stop bits.

After then you have to set up the baud rate. The baud rate is derived from clock of the peripheral bus B (PBA) or at external pin. This source clock is then diveded in divider for integer part and additionally for fractional part to achieve even smaller baudrate error. Assuming that the PBA bus speed is 66 MHz we have to divide it by 71 to get the baud rate of 115 200 bit/s (error 0.8%).

```
usart_setup_speed(usart0, USART_CLK_PBA, 71, 0);
```

The last parameter is the divider for the fractional part which could have been also omitted because it was set to zero. We could have set the fractional part here to 5 to get even smaller baud rate error (0.16%).

At last you have to enable RX and TX channels separately for receiving and transmitting data

```
usart_enable_rx(usart0);
usart_enable_tx(usart0);
```

CHAPTER 10

Class drivers

Aery32 class drivers, abbreviated as *clsdrv*, are high level C++ classes. Thus the name class driver. Class drivers provide a convenient and feature rich APIs, so you most likely want to use these when ever possible.

10.1 Peripheral Input/Output DMA, #include <aery32/periph_iodma.h>

AVR32 microcontrollers have a PDCA module (Peripheral DMA Controller), which allows direct memory access (DMA) between peripheral hardware and MCU's memory independently of the central processing unit (CPU). This feature is useful when the CPU cannot keep up with the rate of data transfer, or where the CPU needs to perform useful work while waiting for a relatively slow I/O data transfer.

Aery32 Framework implements two types of peripheral DMA drivers, input and output. Input type driver can transfer data from a peripheral to memory whereas output type driver transfers data from memory to a peripheral. For example, when using USART peripheral to communicate with PC, input DMA transmit data from PC to MCU and output DMA from MCU to PC.

10.1.1 Class instantiation

To instantiate input or output type DMA class driver you need to tell which DMA channel number *chnum* and peripheral identifier *pid* will be used. Additionally you have to give a pointer to the preallocated buffer and tell its size **in bytes**.

```
volatile uint8_t ibuf[128] = {};
volatile uint8_t obuf[128] = {};

periph_idma input = periph_idma(0, ipid, ibuf, sizeof(ibuf));
periph_odma output = periph_odma(1, opid, obuf, sizeof(obuf));
```

Instantiation does not enable the DMA by default, so you have to enable it before the driver activates the DMA channel

```
input.enable();
output.enable();
```

The total number of DMA channels in UC3A type MCUs is 15 (0-14). Any channel can be assigned to any peripheral id. Be specific with the peripheral id and class driver type, RX is input and TX is output. The possible pid values are:

- AVR32_PDCA_PID_ADC_RX
- AVR32_PDCA_PID_ABDAC_TX
- AVR32_PDCA_PID_SPI0_RX
- AVR32_PDCA_PID_SPI0_TX

- AVR32_PDCA_PID_SPI1_RX
- AVR32_PDCA_PID_SPI1_TX
- AVR32_PDCA_PID_SSC_RX
- AVR32_PDCA_PID_SSC_TX
- AVR32_PDCA_PID_TWI_RX
- AVR32_PDCA_PID_TWI_TX
- AVR32_PDCA_PID_USART0_RX
- AVR32_PDCA_PID_USART0_TX
- AVR32_PDCA_PID_USART1_RX
- AVR32_PDCA_PID_USART1_TX
- AVR32_PDCA_PID_USART2_RX
- AVR32_PDCA_PID_USART2_TX
- AVR32_PDCA_PID_USART3_RX
- AVR32_PDCA_PID_USART3_TX

Note: The peripheral id cannot be assigned more than one channel.

10.1.2 Size of transfer

Peripheral DMA can work with 8, 16 or 32-bit wide size of transfers. The size of transfer is set to 8-bit by default, but can be changed with the `set_sizeof_transfer()` member function. Either a byte, half-word or word can be used (8-bit, 16-bit or 32-bit respectively). The example code below shows how to use 32-bit size of transfer with the analog-to-digital converter.

```
volatile uint32_t buf[32] = {};
periph_idma dma0 = periph_idma(0, AVR32_PDCA_PID_ADC_RX, buf, sizeof(buf));

dma0.set_sizeof_transfer(PDCA_TRANSFER_SIZE_WORD);
dma0.enable();
```

10.1.3 Reading the input DMA, `periph_idma`

The read member function of the Peripheral Input DMA returns the total number of elements moved from the DMA input buffer to a new destination *dest*. If there was nothing to move zero is returned.

```
uint8_t dest;
if (input.read(&dest, 1))
    // one byte read
else
    // there was nothing to read
```

To poll the input buffer whether there are bytes which to read call

```
input.bytes_available();
```

If you suspect that the buffer has been overflown and thus needs to be reset you can do the reset like this:

```
if (input.has_overflown())
    input.reset();
```

In case you want to remove all bytes from the input buffer once and for all call `flush()`:

```
input.flush();
```

Note: With 32-bit size of transfer one read operation will increase the available bytes by 4, because one word (32-bit) is 4 * 8-bit. 16-bit size of transfer in turn would increase the available bytes by 2.

10.1.4 Writing to the output DMA, `periph_odma`

The write member function of the Peripheral Output DMA fills the output buffer, but does not start the transmission yet. To start the transmission you have to call `flush()`.

```
output.write(src, 1);
output.flush();
```

After calling `flush()` you can follow the send process like this:

```
while (output.bytes_in_progress())
    // still trasmitting
```

If you are unsure how many bytes you have written into the output buffer, you can check it like this:

```
if (output.bytes_in_buffer() == output.bufsize)
    output.flush();  // buffer is full, flush it
```

10.2 Serial Port class driver, #include <aery32/serial_port_clsdrv.h>

Serial Port class driver implements serial port communication using *USART module functions* and *Peripheral Input/Output DMA class drivers*. The driver can be used to communicate with PC via COM port or with other integrated chips (ICs) which provide RX and TX signal pins. Hardware handshaking (the use of RTS and CTS signal pins), which requires DMA to work, is also supported by the class. Skip to example.

10.2.1 Class instantiation

To instantiate a Serial Port class driver you need to tell its constructor which USART module you like to use. Additionally input and output DMA buffers, *idma* and *odma*, are needed.

First allocate some space for the the DMA buffers:

```
volatile uint8_t bufdma0[128] = {};
volatile uint8_t bufdma1[128] = {};
```

After then instantiate Peripheral DMA class drivers by using the buffers you just created:

```
periph_idma dma0 = periph_idma(0, AVR32_PDCA_PID_USART0_RX, bufdma0, sizeof(bufdma0));
periph_odma dma1 = periph_odma(1, AVR32_PDCA_PID_USART0_TX, bufdma1, sizeof(bufdma1));
```

The DMA pid value, which is the second parameter of the *periph_idma* and *periph_odma* constructors, defines the USART data direction, so be sure to select Peripheral DMA class' direction properly.

When both DMA drivers have been instantiated, instantiate the Serial Port class driver:

```
serial_port pc = serial_port(usart0, dma0 /* input */, dma1 /* output */);
pc.enable();
```

Note: The object name `pc` was used here, because the connection is intended to be use with PC. See the *Setting up the terminal software in PC side* below.

By default the speed is set to 115200 bit/s (baud error 0.16% with 66 MHz PBA frequency). The default setting for parity is none. Stop and data bits are 1 and 8, respectively. All these settings can be changed with the class member functions.

To change speed call

```
pc.set_speed(speed); // speed in bit/s
```

Everytime you change the speed, the baud error rate is set to the public `error` member and can be checked by calling `pc.error`.

Parity and stop bits can be set like this:

```
pc.set_parity(USART_PARITY_NONE);
pc.set_stopbits(USART_STOPBITS_1);
```

The possible parity options are `USART_PARITY_EVEN`, `USART_PARITY_ODD`, `USART_PARITY_MARKED` and `USART_PARITY_SPACE`. The number of stop bits can be `USART_STOPBITS_1`, `USART_STOPBITS_1p5` or `USART_STOPBITS_2`.

The Serial Port class driver supports several data bits values from 5 to 9. Generally 8 data bits is used, but it can be changed with `set_databits()` member function:

```
pc.set_databits(USART_DATABITS_5);
```

Warning: Keep in mind that if 9 data bits is used, you also have to change the size of transfer of the used *periph_idma* and *periph_odma* class drivers, because 9 bits do not fit in one byte, which is the default DMA transfer size.

10.2.2 Hello World!

When the Serial Port class driver is enabled it's ready to be used. The well known "Hello World!" example would work like this:

```
pc << "Hello Aery" << 32;
```

or like this:

```
pc.printf("Hello Aery%d", 32);
```

A single character can be read and write like this:

```
char c = pc.getc();
pc.putc(c);
```

If you like to put the read character back to the read buffer call

```
pc.putback(c);
```

10.2.3 Getline and line termination

You can read user input in lines like this:

```
char line[32] = {};
pc.getline(line);
```

`getline()` will extract characters to *line* C string until either the DMA input buffer is full or the delimiting character, which is `\r\n` by defaut, is found. Characters that precede the char (del), which is a backspace (decimal value 127), are discarded from the line.

The total number of the read characters can be saved like this:

```
size_t nread;
pc.getline(line, &nread);
```

Delimitation character and `\0` aren't added to *nread*.

The default *delim* can be set by calling `set_default_delim()` member function in this way:

```
pc.set_default_delim('\n');
pc.set_default_delim("\r\n");
```

Note that the delimitation character *delim* can be either a single character or two sequential characters. If you need to use occasionally some other delimitation character, define it as a third argument like this:

```
pc.getline(line, &nread, '\n');
```

Note: Be specific with the `''` and `""` notation. For example, `set_default_delim("\n");` would set the default line termination to `\n\0` instead of `\n`.

Hint: For input scanning, it's a good practice first fetch the line and then use `sscanf()` for that.

```
int i = 0;
pc.getline(line);
sscanf(line, "%d", &i);
```

Hint: In main for loop you can skip empty lines in this way

```
for (;;) {
    pc.getline(line, &nread);
    if (nread == 0)
        continue; // skip

    // ...
}
```

10.2.4 Flush and other supportive functions

Sometimes you need to flush the input buffer from all read bytes. This can be done with `flush()` member function. If you like to know how many bytes have been received, call `bytes_available()`.

It's also possible that the input buffer gets overflown. This can be checked by calling `has_overflown()`. If the buffer has been overflown, you can reset the serial port by calling `reset()`.

10.2.5 Hardware handshaking

To enable hardware handshaking just call:

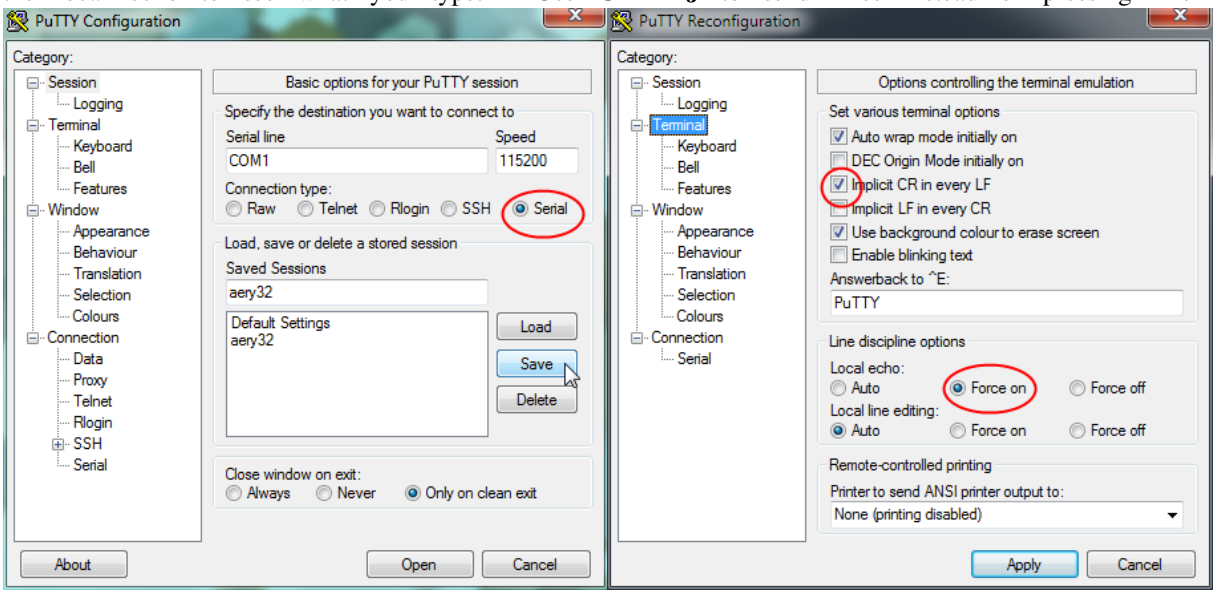
```
pc.enable_hw_handshaking();
```

When the handshaking is enabled the receiver drives the RTS pin and the level on the CTS pin modifies the behavior of the transmitter.

10.2.6 Setting up the terminal software in PC side

There are several free terminal emulator software which to use in Windows. PuTTY and Tera Term are most known and widely used.

If you choose to use PuTTY, select serial and set up the port (serial line) and speed. Before saving the session go to the Terminal slide and enable *Implicit CR in every LF*. Additionally force the local echo to see what you type. Use **CTRL+J** to send lines instead of pressing **ENTER**.



CHAPTER 11

Aery32 framework in your favorite editor

Aery32 framework is not restricted to any specific integrated development environment (IDE) or editor. We respect your taste. The best one is the one you like best and to figure out which one you like best you have to try a few. We like Sublime Text 2 and thus the framework comes with a preset project-file for this editor. However, we have kindly made instructions to use Aery32 in Eclipse as well :)

11.1 Eclipse Juno

11.1.1 Installation

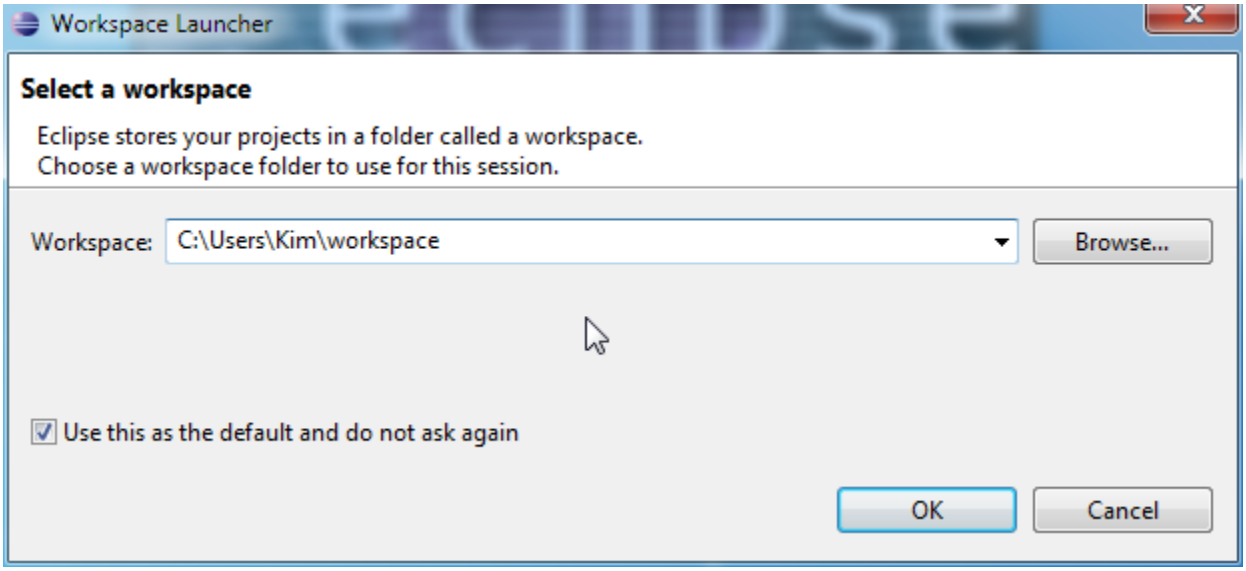
First make sure that you have Java runtime environment (Java JRE) installed. You can download Java from Oracle. After then download Eclipse IDE for C/C++ Developers from Eclipse's homepage. Be sure to select the same bit version than your JRE.

To continue browse to the folder where you downloaded Eclipse and unzip the file there. Now you have a folder called `eclipse`. You can just browse to that folder and double click `eclipse.exe` to start Eclipse. However, before doing that you may like to move the `eclipse` folder some other place (away from your download directory) and then make a shortcut to `eclipse.exe`.

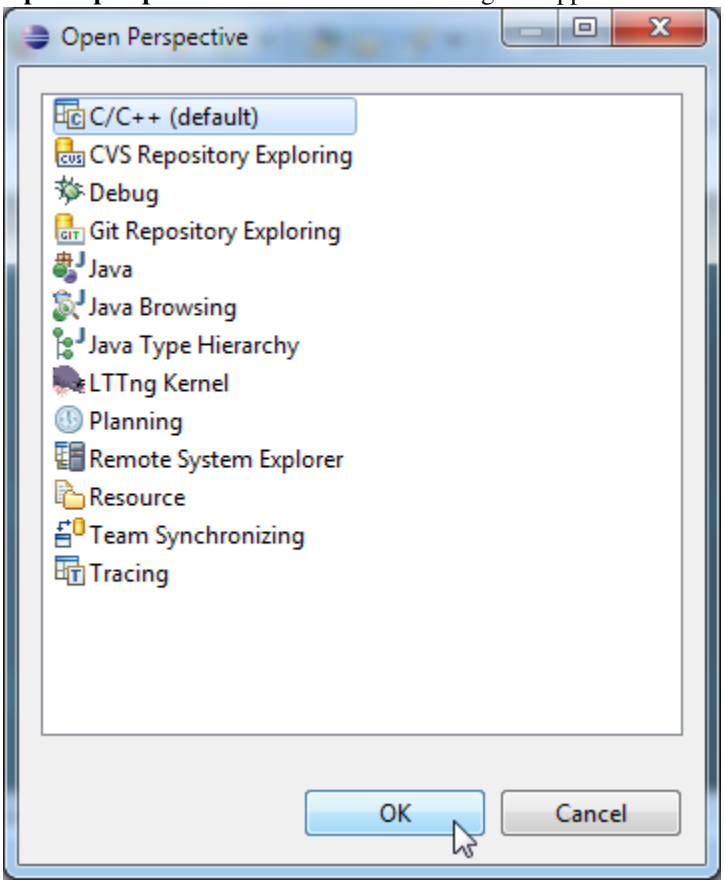
That was all about the installation of Eclipse.

First start

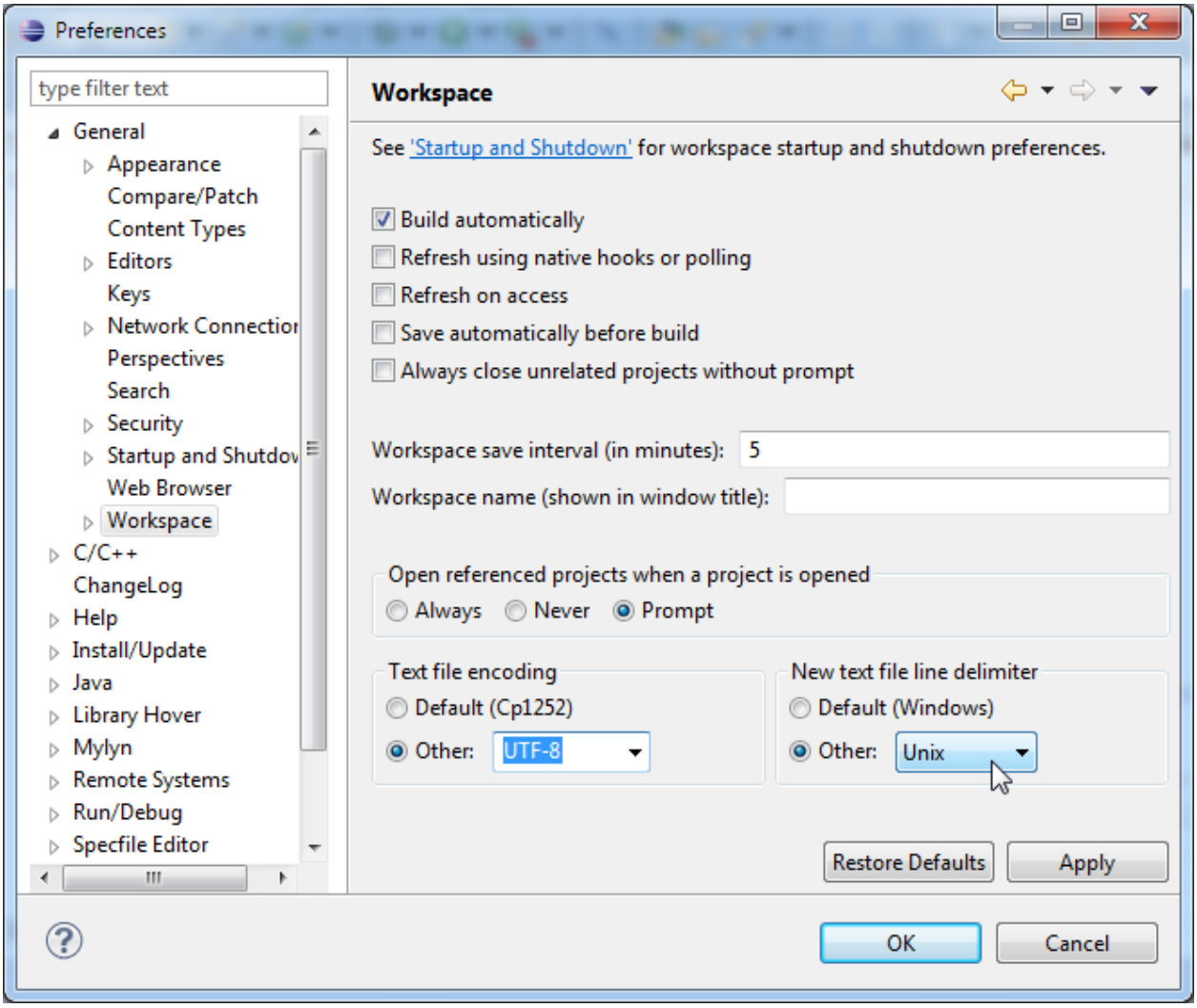
Eclipse stores your projects in a folder called a workspace. You may have several workspaces around your hard disk. Each time you start Eclipse the location of the workspace has been asked. If you like to work in one workspace only, check the box that says **Use this as the default and do not ask again**.



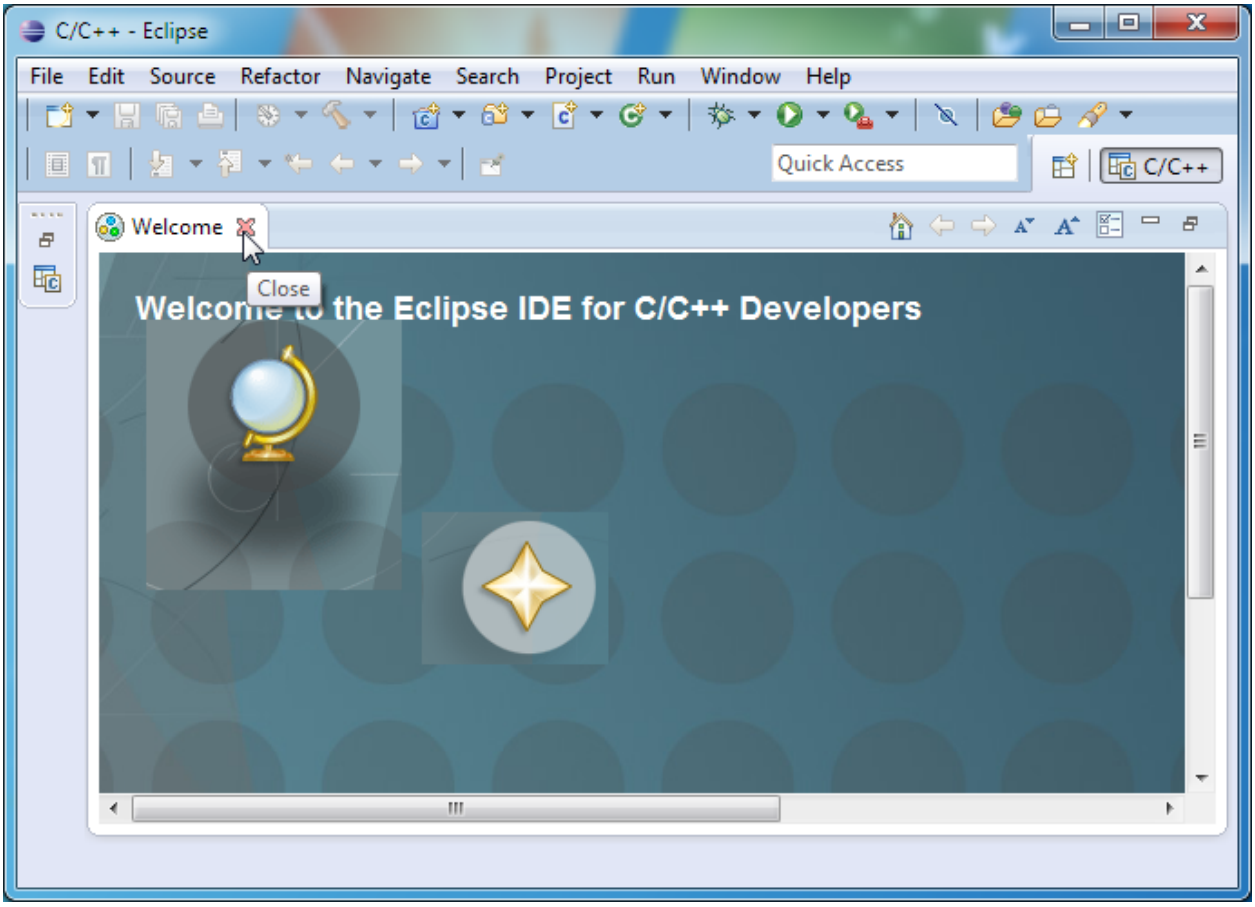
When Eclipse has been started, change the perspective from Java to C/C++ by clicking the **Open perspective** icon from the right upper corner. Then select **C/C++** and press OK.



It is also good idea to use **UTF-8** text file encoding and Unix style new text file line delimiter (Aery32 Framework uses these settings). You can change these preferences globally for the workspace you are working from **Window / Preferences**. From the left-hand side list select **General / Workspace**.

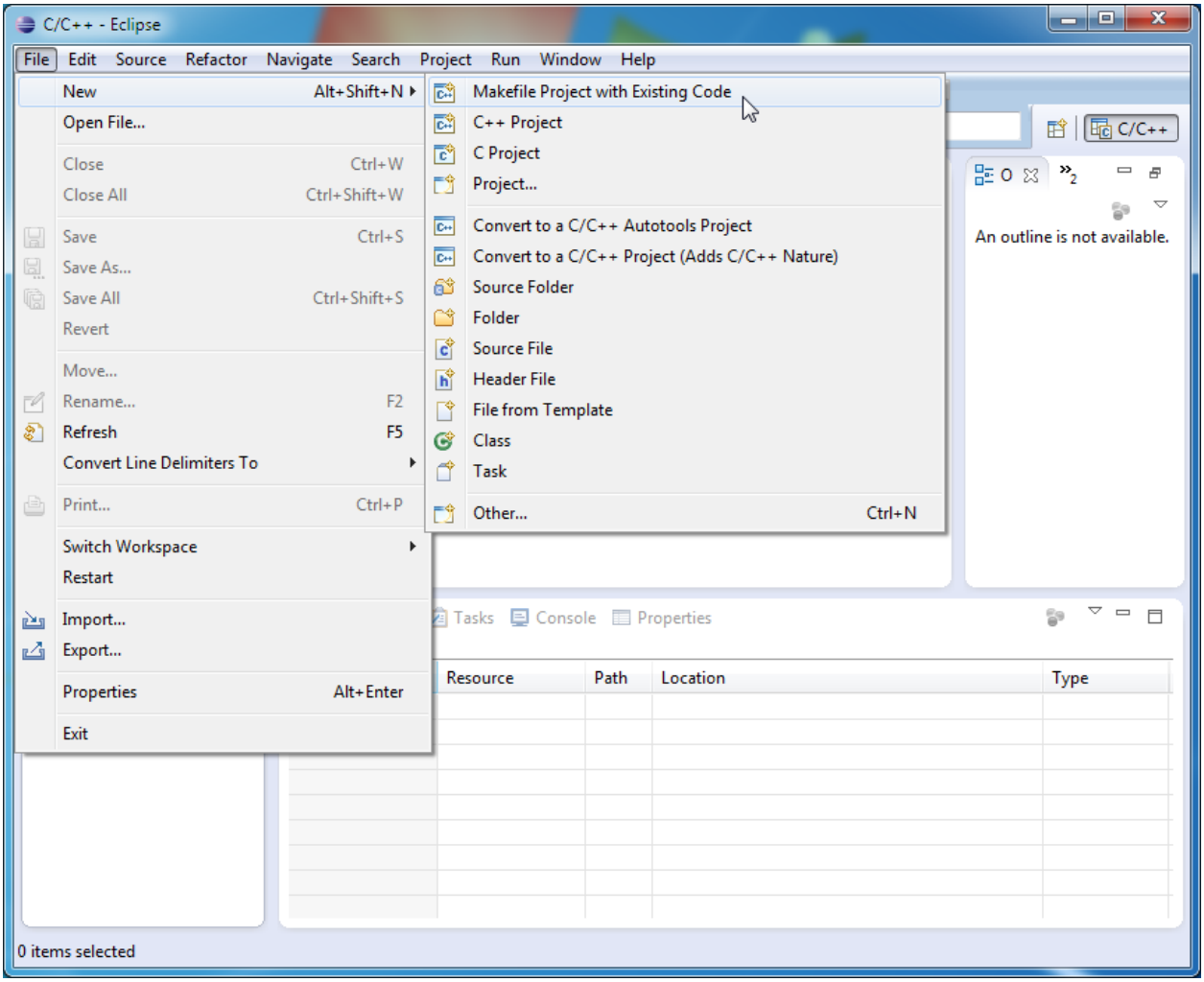


Lastly get rid of the welcome screen by closing it. You can open it again, if you like from **Help / Welcome**

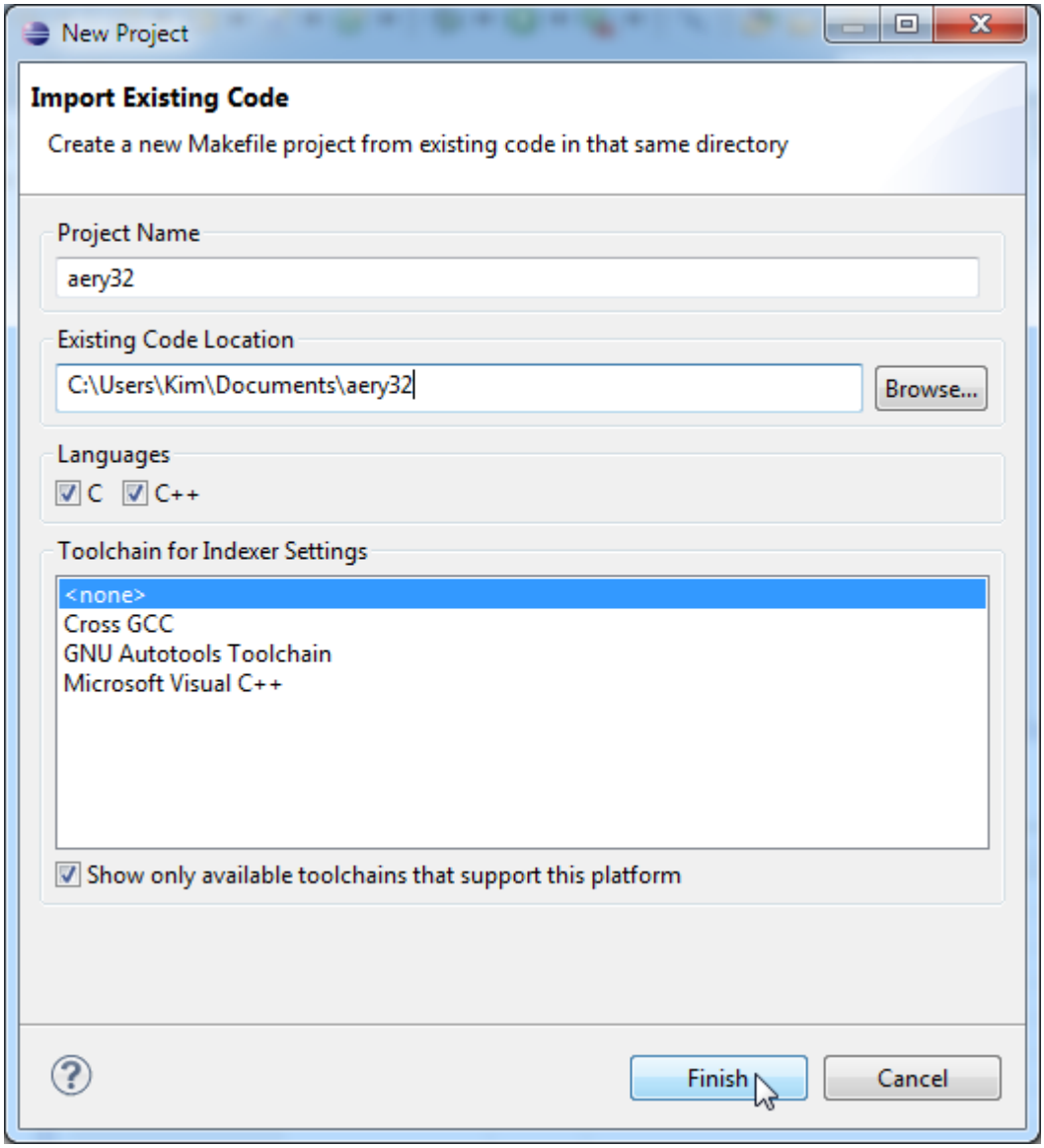


11.1.2 Import Aery32 Framework as a Makefile project

Aery32 Framework can be imported into Eclipse as a Makefile project. Start by creating a new project **File / New / Makefile Project ...**, or press **Alt-Shift-N**.

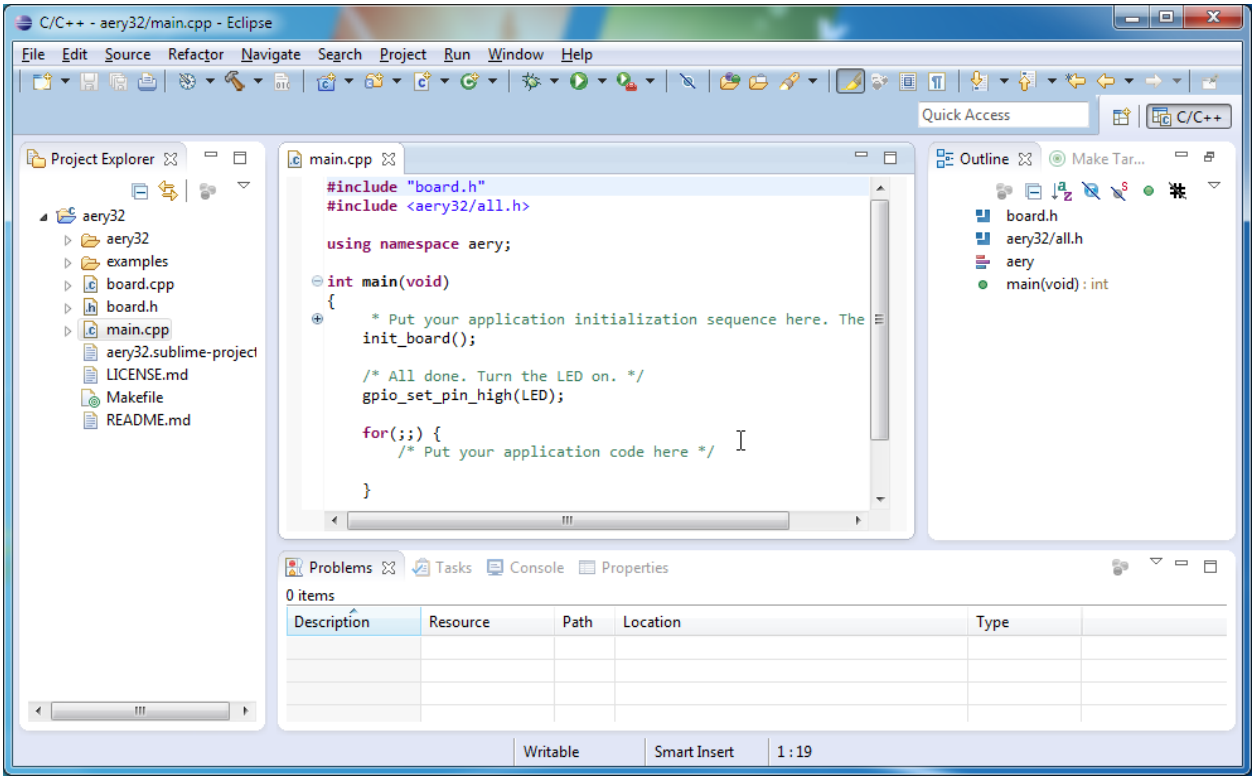


Browse to the location of Aery32 Framework by using **Browse...** button. Eclipse will name the project according to the directory, but you can change it whatever you like. It is important to choose `<none>` for the **Toolchain**



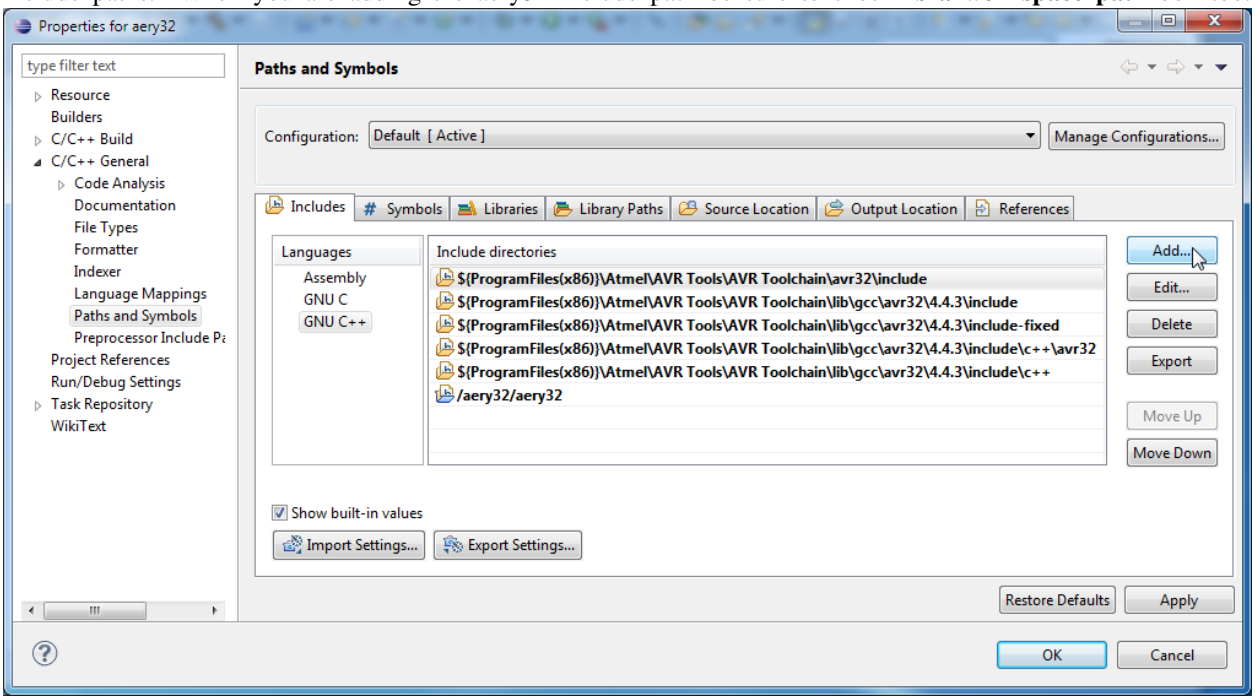
for Indexer.

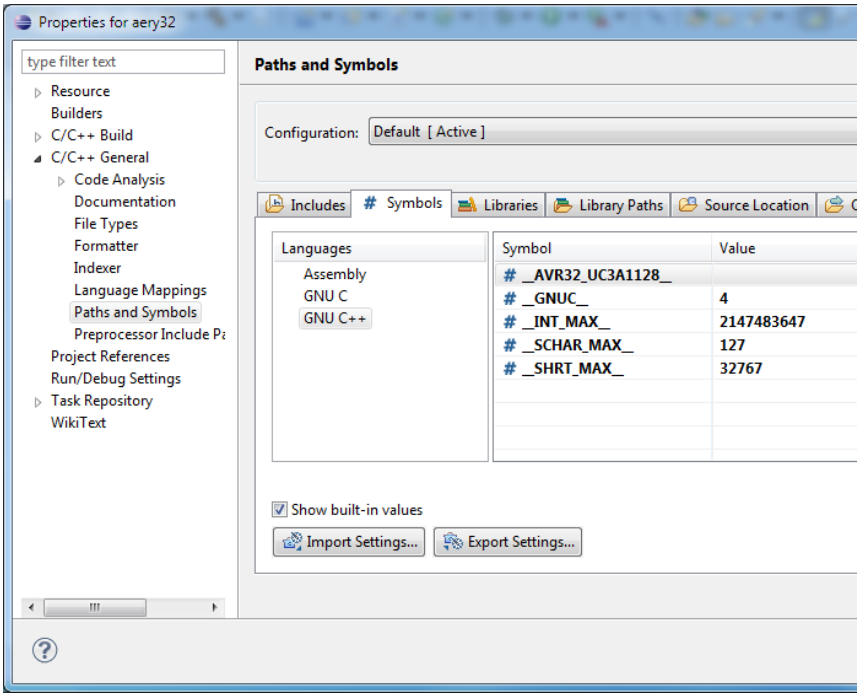
Now you have a project created and you can open e.g. the `main.cpp` from the **Project Explorer**. Right-hand side you have panels for the source file **Outline** and the project **Make Targets**.



11.1.3 Setting Paths and Symbols

To get Eclipse indexer working with the AVR32 Toolchain and Aery32 include files, you have to set up paths and symbols. Open **Project / Properties** and then **C/C++ General / Paths and Symbols**. Add the following include paths. When you are adding the aery32 include path be sure to check **Is a workspace path** box too.

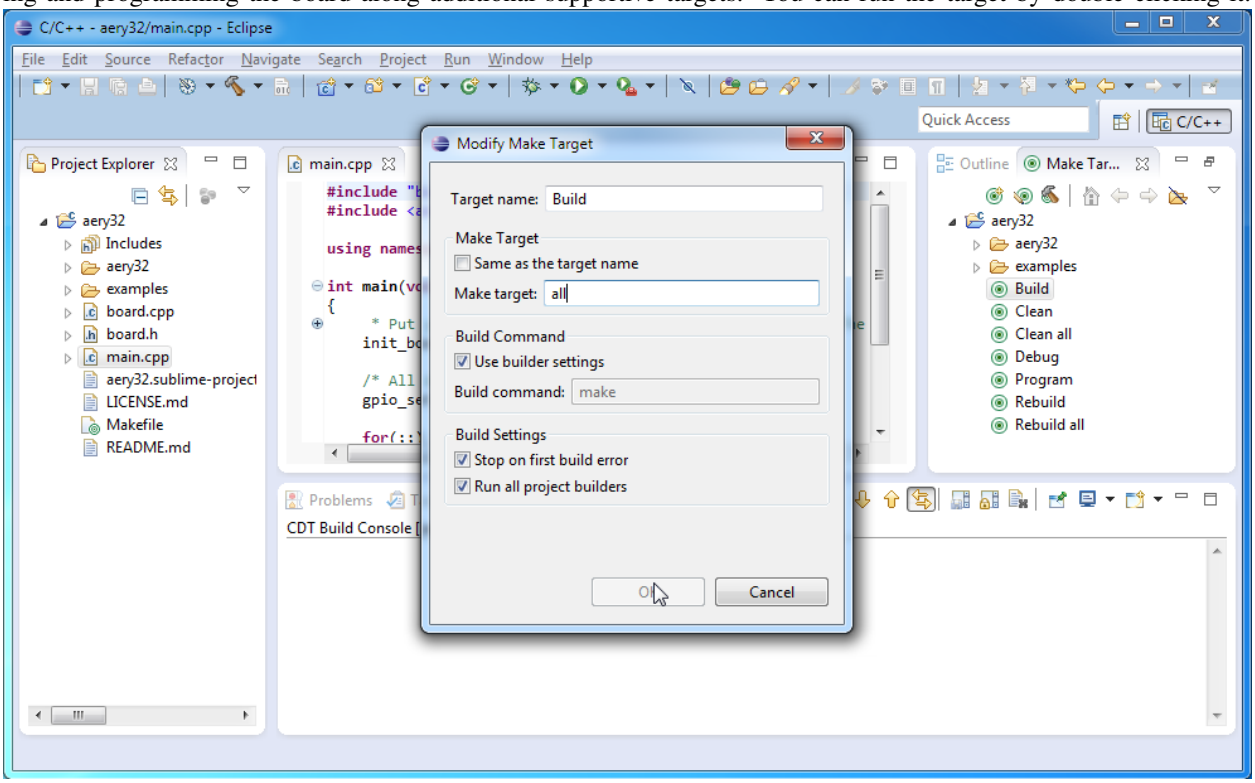




Next open **Symbols** tab and add the following symbols.

11.1.4 Setting Makefile targets

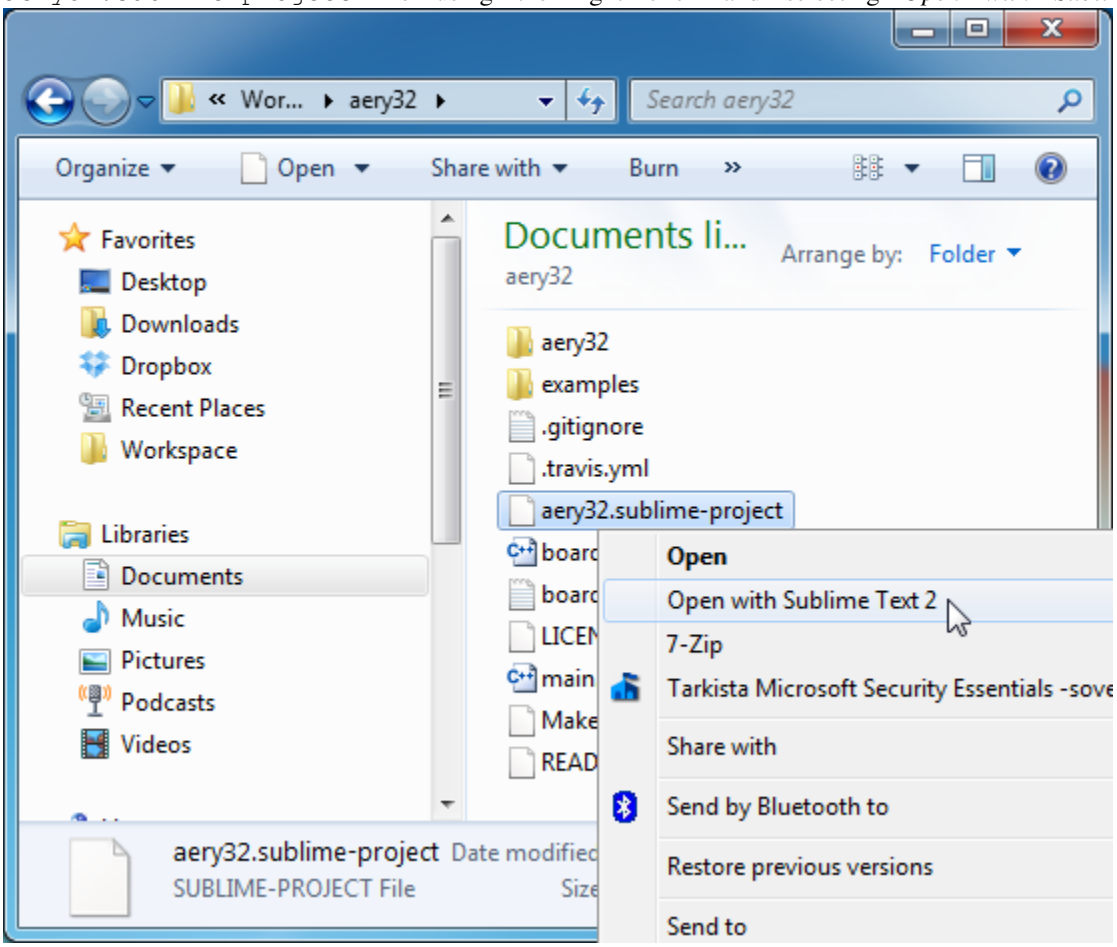
Makefile targets can be added from the right-hand side panel. Here I have added all targets needed for compiling and programming the board along additional supportive targets. You can run the target by double clicking it.



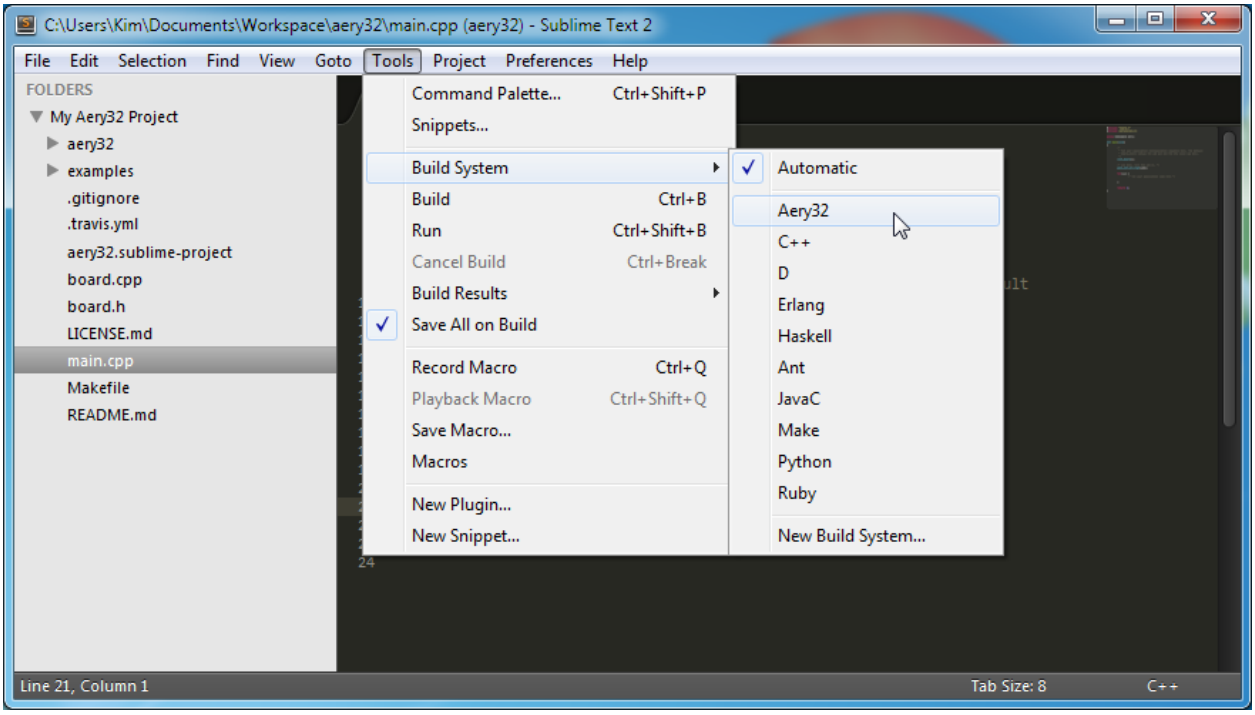
11.2 Sublime Text 2

Note: These instructions are written for Windows but should work similarly in Linux and Mac OS X. And don't miss the Aery32 plug-in for Sublime Text 2.

Aery32 Framework comes with the default Sublime Text 2 project file which you can use straight away. The quickest way to start the project is to browse to the project directory and open `aery32.sublime-project` file using the right click and selecting *Open with Sublime Text 2*.

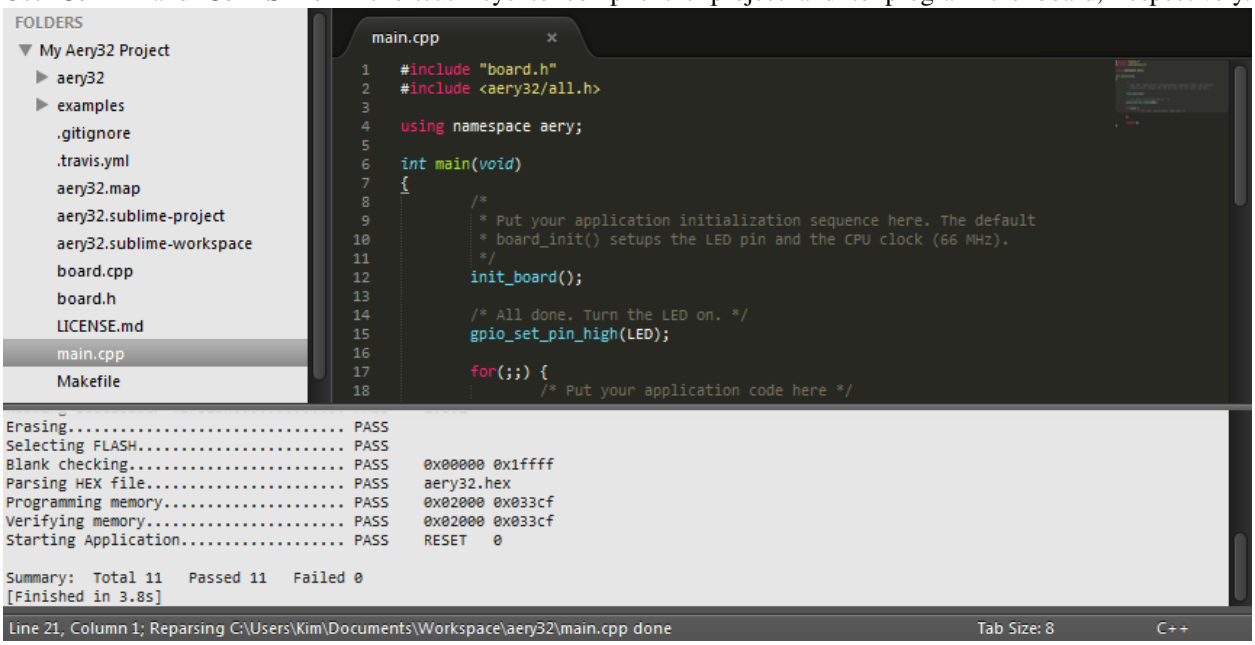


After then change the default build system to *Aery32* from *Tools/Build System*.

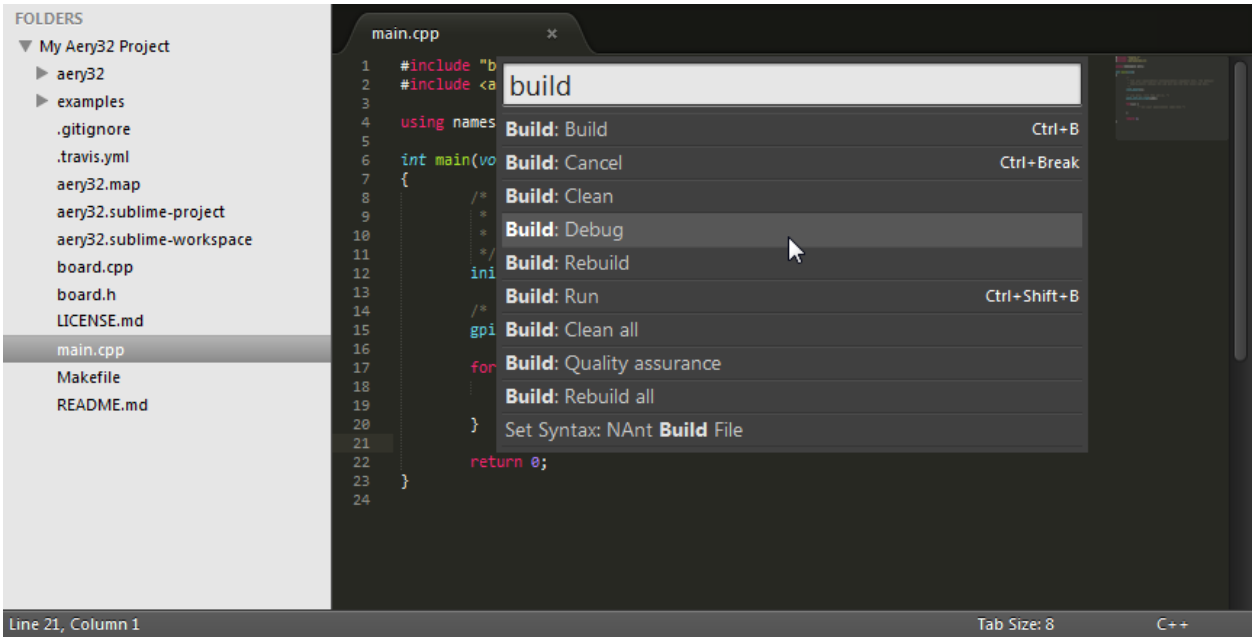


11.2.1 Shortcut keys

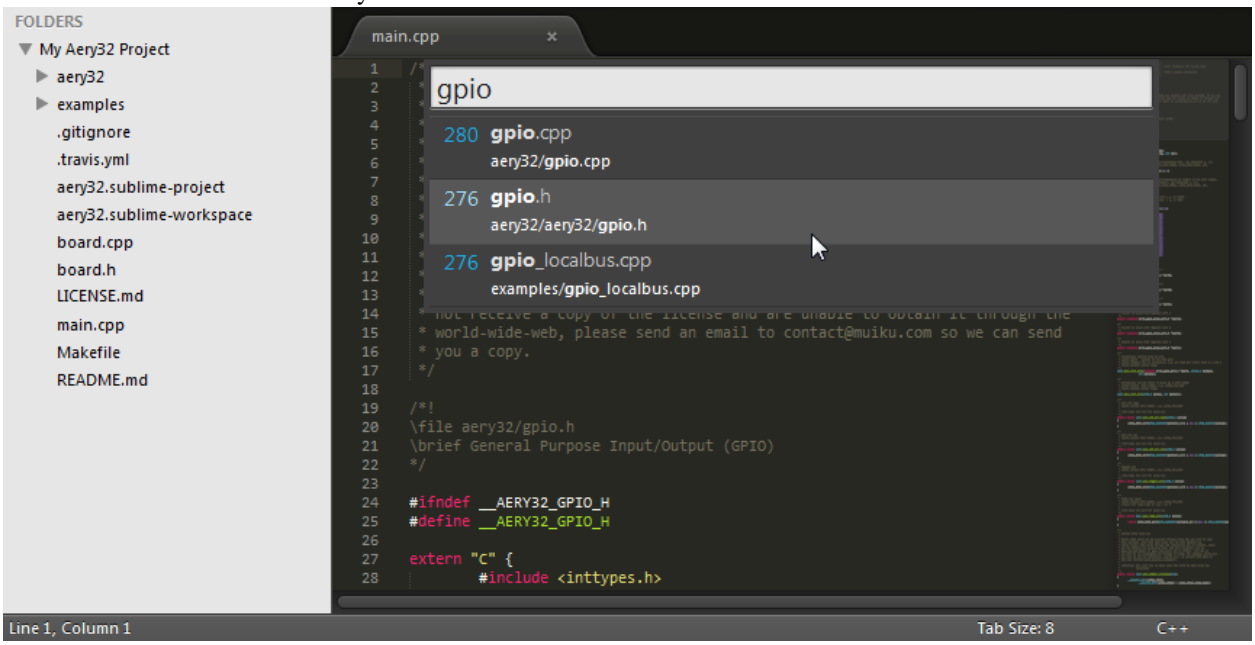
Use **Ctrl+B** and **Ctrl+Shift+B** shortcut keys to compile the project and to program the board, respectively.



For other make targets press **Ctrl+Shift+P** and write *build*. Then select the desired target from the list.

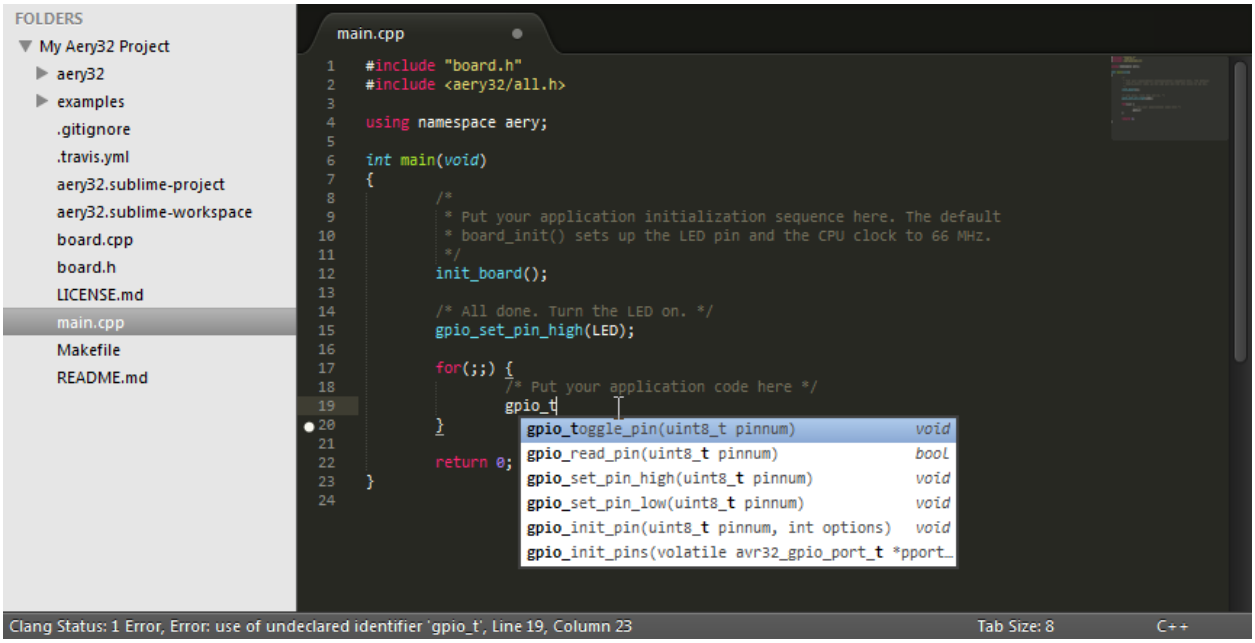


When you want to open a file, let's say `aery32/aery32/gpio.h`, press **Ctrl+P** and write *gpio*. Then select the file from the list. Of course you can also use the folder tree from the left-hand side if you like. Btw. it can be switched on and off with **Ctrl+K+B**.



11.2.2 Autocomplete with SublimeClang

SublimeClang provides a neat autocompletion for Aery32 project. With this plugin you don't have to remember all the function names and their parameters completely. Just write the beginning of the function, for example *gpio_t*, and you get a list of functions. Press **Ctrl+Space** to move on the list and **Tab** to select the function. When you have set the first param, you can press **Tab** again to jump to the next one.



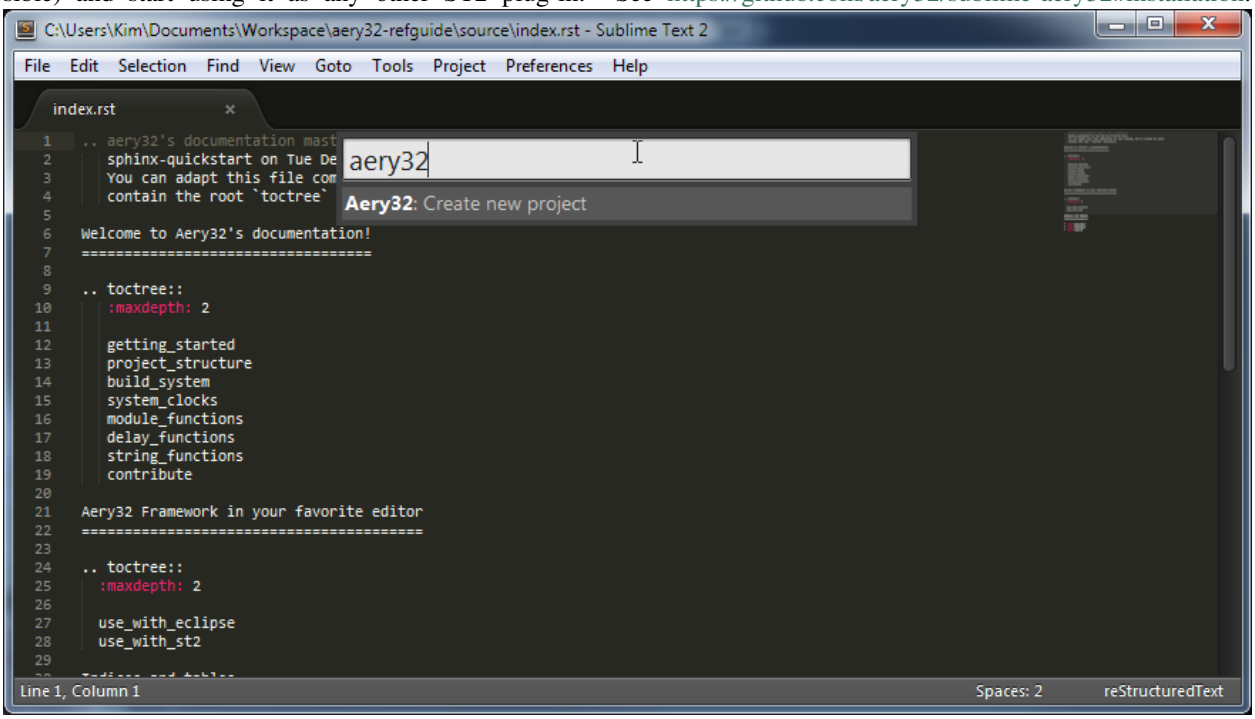
SublimeClang prerequisites for the clang static analyzer is to have clang installed and set in your path. The other functionality works without having the clang binaries installed. Here we only use the "other functionalities" and thus do not need clang. So install SublimeClang as instructed in its README without paying any attention toward clang. In Linux you can also skip the additional prerequisities.

Now open `aery32.sublime-project` and add the following settings. If you are on Linux or Mac OS X, alter the AVR Toolchain installation directory appropriately. The preprocessor definitions within the sublimeclang_options, like *-D__AVR32_UC3A1128__=1*, are the few that we have found to be essentials. If you use Aery32 Sublime Text 2 plug-in to create a new project, other definitions are added too so the list will be far longer.

```
"sublimeclang_enabled": True,
"sublimeclang_dont_prepend_clang_includes": True,
"sublimeclang_show_output_panel": True,
"sublimeclang_hide_output_when_empty": True,
"sublimeclang_show_status": True,
"sublimeclang_show_visual_error_marks": True,
"sublimeclang_options": [
        "-Wall", "-Wno-attributes",
        "-ccc-host-triple", "mips",
        "-I${project_path:aery32}",
        "-include", "${project_path:settings.h}",
        "-IC:/Program Files (x86)/Atmel/AVR Tools/AVR Toolchain/avr32/include",
        "-IC:/Program Files (x86)/Atmel/AVR Tools/AVR Toolchain/lib/gcc/avr32/4.4.3/include",
        "-IC:/Program Files (x86)/Atmel/AVR Tools/AVR Toolchain/lib/gcc/avr32/4.4.3/include-fixed",
        "-IC:/Program Files (x86)/Atmel/AVR Tools/AVR Toolchain/lib/gcc/avr32/4.4.3/include/c++",
        "-IC:/Program Files (x86)/Atmel/AVR Tools/AVR Toolchain/lib/gcc/avr32/4.4.3/include/c++/avr32
        "-D__GNUC__=4",
        "-D__AVR32_UC3A1128__=1",
        "-D__INT_MAX__=2147483647",
        "-D__SHRT_MAX__=32767",
        "-D__SCHAR_MAX__=127"
]
```

11.2.3 Aery32 plug-in

With Aery32 plug-in for Sublime Text 2 you can easily create new projects. All the above setup process is automated for you. Just install the plugin via the Package Control plug-in (manual install also possible) and start using it as any other ST2 plug-in. See https://github.com/aery32/sublime-aery32#installation.



CHAPTER 12

Indices and tables

- *genindex*
- *modindex*
- *search*