COMPSYS303
Lab 2 – Application Programming with Altera HAL

Creating Application Software with HAL provided by Altera

Last Updated: 5th July 2022

1. Memory mapped registers and Platform Designer

Peripherals/components provided by Platform Designer follow a specification that is compatible with the Avalon bus. Avalon bus is the glue-logic that interconnects all components with numerous arbitrators and multiplexers. These components can be further divided into two groups – master or slave components. The Nios II processor is an example of a master component. LEDs, switches, push buttons, timers, and UART are examples of a slave component. One of the major differences between a master and a slave is that a master can both send and receive commands to/from other masters or slaves; while slaves can only receive and respond to the commands issued by the master.

Slaves are often designed with internal registers. These registers allow the slave to communicate, store configurations, coefficients, or debugging information. The behavior of a slave device could be controlled by modifying the value in the configuration registers. Coefficient registers store data which are accessed regularly. Caching this information saves time as it reduces the number of accesses to the external memory. Debugging messages can also be stored in the registers. This can help the designer of the slave components (or even the system) to understand the internal status of the slave. In this lab, registers in salves are used for configurations (enable/disable registers).

Even we use the term registers, these are different to the registers (defined in the register file) within the processor. Processor registers can be accessed directly. In contrast, registers of slave components are **memory mapped** to the processor (Nios II in this case). The organization of the Nios II processor and peripherals is show in Figure 1. Accessing of slave registers is through memory accessing operations. Memory mappings of slave registers are according to the addresses specified in Platform Designer. For instance in Figure 1, two slave components are added with memory addresses 0x00001000 and 0x00002000 respectively. Registers within the slave components can have variable length of bits. However, in practice, they are designed to match the width of processor data bus/internal register – 32 bits for Nios II. The address of the system is **byte**-based, which allows the minimum size of data being accessed to be 8 bits wide. Hence each register will occupy 4 bytes of memory space (32 bits / 8 bits-byte = 4 bytes). Normally the first accessible register will be addressed as the component's address defined in Platform Designer, and the second accessible register will be four byte-address after that.

Besides writing in assembly language, another way to access these registers is using **IORD** and **IOWR** functions for reading and writing respectively. These are defined in the "**io.h**" header file. IORD and IOWR functions are macros (a macro is a mechanism provided by C) that wrap around built-in assembly functions to achieve memory mapped register operations. There are 3 variants for each function (i.e., 6 functions in total), which are as follows IORD_8DIRECT, IORD_16DIRECT, IORD_32DIRECT, IOWR_8DIRECT, IOWR_16DIRECT, and IOWR_32DIRECT, where the numbers (8, 16, and 32) indicate the width of the data for a particular IO operation. **However, using IORD and IOWR will just do the job**. The following section of the code will read the value of the register-1 of peripheral-1, and then writes this value to the register-2 of peripheral-2.

```
1  #include <io.h>
2  int main(void)
3  {
4      // syntax of IORD and IOWR :
5      // IORD(address_of_the_peripheral, nth_register)
6      // IOWR(address_of_the_peripheral, nth_register, value_to_write)
7      int value = 0;
8      value = IORD(0x00001000, 0); // The first register with offset 0
9      IOWR(0x00002000, 1, value); // The second register with offset 1
10 }
```

Listing 1 - Uses of IORD and IOWR functions

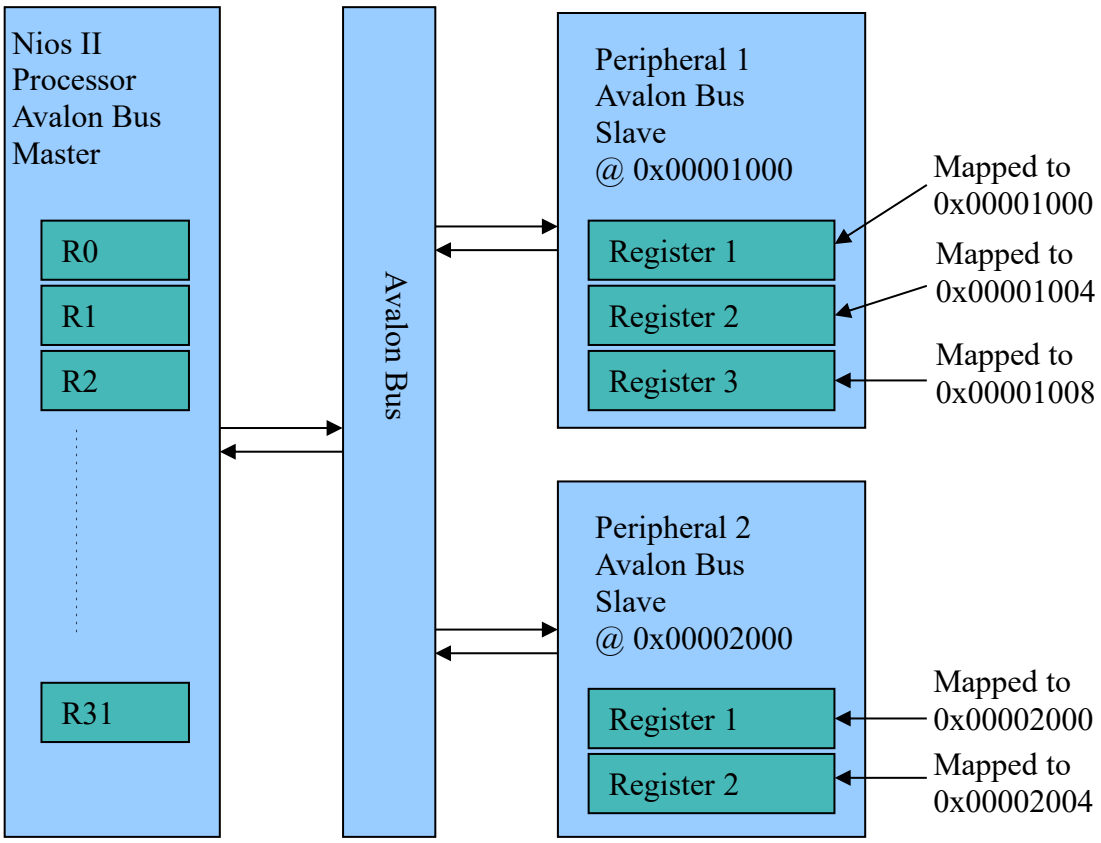

Figure 1 - Nios II and peripherals and memory mapped I/O registers

2. System.h and HAL APIs

For an application programmer, it is not convenient to program everything with IORD and IOWR. It will be hard to refer to the exact address of the register. Also, a program filled with IORD and IOWR instructions will be hard to understand and maintain. Hence, Altera provides a set of HAL (Hardware Abstraction Layer) APIs (Application Programming Interface) to access memory mapped components for an easier and more sensible reference to the components and their registers.

Before introducing the use of the HAL APIs, a file named **system.h** should be explained. A configuration of the nios2_system is defined in the previous lab while the nios2_system is generated in Platform Designer. The file containing this information is "**nios2_system.sopcinfo**" located in the project directory. This file includes important information of the system and its components, such as the processor type, the available hardware components and their interconnections. When building the project (from lab1, or your own projects later), especially when building the system library, a file named system.h is generated. Once finished building the system library, you should be able to find the system.h file by expanding the project tree of the *projectName*_bsp. Entries within the system.h are in **#define** fashion, for instance, the base address of push buttons is 0x01912080 (might be different for your case), and is assigned with a symbolic name as "**KEYS_BASE**". You can use KEYS_BASE in the application code instead of hard-coding the based address. This approach increases the portability of the program across different hardware configuration (Note that the names of modules are the same but the assigned addresses could be different). You should use system.h macros at all times.

3. Accessing file-based peripherals

The following text is a segment of the system.h, which is of the component UART added in the previous lab. The values might vary from yours and the underlying example which is fine.

```
#define UART_NAME "/dev/uart"
#define UART_TYPE "altera_avalon_uart"
#define UART_BASE 0x00512000
#define UART_SPAN 32
#define UART_IRQ 8
#define UART_BAUD 115200\
```

Listing 2 – Definitions for the UART

3.1 Using File Pointers

The Altera HAL provides **file-pointer** mechanism for the application programmer to use peripherals as files. A file pointer allows standard write and read methods to be used to interact with this device. To create the file pointer you just need to open the peripheral as if it were a file through the `fopen()` command. This command takes two parameters, the file name to be opened (in our case UART_NAME) and the mode to open the file as, and returns the file pointer. The mode can be values such as “r” for read-only access, “w” for write-only access, “r+” for read-and-write access, etc. In this case, outputting a string to the UART port can be done by the following code:

```
#include <stdio.h>
#include <string.h>
#include "system.h"
int main(void)
{
	char* stringToOutput = "example1";
	FILE* fp; // To operate on a file descriptor, a file pointer is required

	// a file descriptor can be open by using fopen function
	// the syntax of fopen, which will return a file pointer
	// fopen("the_string_of_the_name_of_file_descriptor",
"the_string_of_the_attribute")
	// if success, it will return the pointer to the device
	// otherwise, a NULL pointer will be returned
	fp = fopen(UART_NAME, "w"); // open up UART with write accesss
	if (fp != NULL) // check if the UART is open successfully
	{
		fprintf(fp, "%s", stringToOutput); // use fprintf to write things
		fclose(fp); // remember to close the file
	}
	return 0;
}
```

Listing 3 – Using file descriptors to control peripherals

To write to this file pointer you can use standard C file-writing calls such as `fprintf()` which takes the first argument as the file pointer you wish to write to, and the second argument as the string you want to write.

Similarly, any standard C file-reading calls can be used to read from the device, such as `fgetc()` which will read a single character from the file pointer which is provided as an argument. This function returns the next character in the receive buffer and, if there is no pending character, will

block until the next character arrives.

3.2 Using Status Registers

Similarly to how UART was handled on the ATMega boards from 209, you can use the registers of the Nios II system to interact with the UART peripheral. In Nios II, each of these registers are accessed through associated helper functions with the base address of the UART peripheral (UART_BASE) passed as an argument. The following should be all you need to use these registers.

- **Status register**: `IORD_ALTERA_AVALON_UART_STATUS(base)` – Returns the status register. A pre-defined mask to check for waiting data is `ALTERA_AVALON_UART_STATUS_RRDY_MASK`.
- **UART read**: `IORD_ALTERA_AVALON_UART_RXDATA(base)` – Returns the current received value.
- **UART write**: `IOWR_ALTERA_AVALON_UART_TXDATA(base, data)` – Writes a value to the UART.

If desired, an Interrupt Service Routine (ISR) can be created which will be called each time a new piece of data arrives on the UART. This is initialised in a similar way to other interrupts (e.g. buttons) using an `alt_irq_register()` call to register the ISR, an `IOWR_ALTERA_AVALON_PIO_IRQ_MASK()` call to enable the interrupt handler, and associated `IOWR_ALTERA_AVALON_PIO_EDGE_CAP()` calls whenever the interrupts should be cleared.

3.3 Using File Descriptors

The final way to use the UART peripheral is through file descriptors. These function similarly to file pointers, with some small differences.

Writing to file descriptors is achieved through the `write()` method, which takes three arguments. The first argument being the file descriptor to write to, the second being a pointer to where the data to be written is stored, and the third being the number of bytes to write from that location. For example, if you have an array of 10 characters named “a”, you could provide “a” as the second argument and “10” as the third argument in order to write all characters, or just “5” to write just the first five characters.

Reading from a file descriptor is through the `read()` method, which again takes three arguments. The first argument is the file descriptor to read from, the second argument is the memory location (i.e. pointer) where data should be stored once it is read, and the third argument is the maximum number of bytes to read. One useful aspect of this method is that it will return the number of bytes that it actually managed to read from the file and will not block if there is no awaiting data. Instead, if no data is waiting to be read from the file then the method will simply return that it read zero bytes.

4 Accessing other peripherals with HAL APIs

For embedded applications, their characteristics can be categorized into two kinds: data dominated or control dominated behaviors. For the former type of applications, such as sampling for example, the sensor-data is obtained at fixed time intervals. The application software must know “when” it should start the sampling process. It is achieved by using timers, and when the timer timeouts, an interrupt is generated, which is then used to initiate sampling. For control-dominated applications,

the interactions between external world and internal application are also achieved by interrupts generated by different devices, such as push buttons. Hence, it is important to understand how interrupts work, and how to setup an Interrupt Service Routine (**ISR**).

4.1 Requirements of writing ISRs

To successfully implement an ISR, following requirements must be ensured:

1. As you are programming a device, which will generate an interrupt it is essential to understand how to initialize and handle interrupts. This device is assigned an IRQ number, so its interrupts can be distinguished compared to interrupts from other devices. You are not required to memorize the IRQ numbers nor hard code them in your applications. These are defined in the system.h file.
2. You have to provide an ISR function. This function will be executed when an interrupt occurs (asserted).
3. Within the ISR function, you might need to do some housekeeping work such as clearing the flag that caused the interrupt, or re-enable interrupt if required (sometimes the interrupt is disabled once the control enters the ISR).
4. The interrupt of the device must be enabled to make the ISR take the affect
5. The ISR is needs to be registered so that the NIOS interrupt handling mechanism knows where to transfer control once this interrupt is recognized.

To write an ISR for buttons, we can check the above requirements. The buttons will generate interrupts as mentioned in the previous lab.

From the system.h we can find:

```
1  #define KEYS_NAME "/dev/keys"
2  #define KEYS_TYPE "altera_avalon_pio"
3  #define KEYS_BASE 0x01912080
4  #define KEYS_SPAN 16
5  #define KEYS_IRQ 4
```

Listing 4 – Definitions for the Buttons

KEYS_IRQ represents the interrupt number assigned by Platform Designer. We are also going to use the **KEYS_BASE** as the peripheral address instead of hard coding the address (0x01912080) of the buttons. Before writing the ISR, for PIO (parallel input/outputs, such as buttons, switches) functions defined in “altera_avalon_pio_regs.h” must be included.

4.1.1 IORD_ALTERA_AVALON_PIO_DATA(BASE_NAME)

This function was introduced in the previous lab. It is used to read the current value of the PIO component, which is very similar to IORD. For instance, IORD_ALTERA_AVALON_PIO_DATA(KEYS_BASE) will return an integer value which represents the or-values the buttons.

4.1.2 IOWR_ALTERA_AVALON_PIO_DATA(BASE_NAME, value)

This will write the value to the device with BASE_NAME. This function does not work over input-only PIO devices.

4.1.3 IORD_ALTERA_AVALON_PIO_IRQ_MASK(BASE_ADDR)

This will return the value representing the current masking status of the input ports. For instance, for 3 buttons (3 input bits), the decimal value of 7 (binary string **111**, which "masks" all 3 bits) means all of the buttons are enabled with interrupts.

4.1.4 IOWR_ALTERA_AVALON_PIO_IRQ_MASK(BASE_ADDR, value)

Similar to the previous function, but write the value to set the mask. This function is used to enable or disable interrupts.

4.1.5 IORD_ALTERA_AVALON_PIO_EDGE_CAP(BASE_NAME)

A button will interrupt the CPU, when its respective bit in the edge-capture register becomes active. The current status of this register can be obtained by calling this function.

4.1.6 IOWR_ALTERA_AVALON_PIO_EDGE_CAP(BASE_NAME, value)

This function is similar to the previous one, but it writes the value to edge capture register. After an interrupt one could use this function to unmask the bit related to the specific interrupt. So, it's ready to accept next interrupt.

4.2 The body of the ISR

The implementation of the ISR must follow the exact function prototype, defined by Altera. An example of an ISR function is as follows:

void name_of_the_isr_function (void* context, alt_u32 ID)

The first argument is of type void pointer. It means you can pass any kind of pointer to the ISR function (that pointer will be passed in the ISR register function which will be discussed later). The second argument is the ID of the interrupt, for buttons, it is **KEYS_IRQ** as mentioned previously. The ID will be passed by the ISR register function as well. This process is further explained in the following section.

4.3 Registering the ISR

The ISR registering function is:
int alt_irq_register (alt_u32 id, void* context, void (*isr)(void* , alt_u32))

The first argument is the IRQ number of the interrupt. The second argument is the pointer to anything, which the programmer wishes to pass to the interrupt (if you are working on the global variables, then you can just pass NULL to it). The third argument is a **function pointer.** This is the ISR function name, and is executed when the interrupt occurs. Also, to register ISR, you need to include "**sys/alt_irq.h**"

The following code segment is an example of button ISR that will print out which button is pushed. Note that a value 4 means the third button (key2 - zero based index) is pushed. This translates to "100" in binary.

```
1  #include <system.h> // to use the symbolic names
2  #include <altera_avalon_pio_regs.h> // to use PIO functions
3  #include <alt_types.h> // alt_u32 is a kind of alt_types
4  #include <sys/alt_irq.h> // to register interrupts
5
6  // first we write our interrupt function
7  void button_interrupts_function(void* context, alt_u32 id)
8  {
9          int* temp = (int*) context; // need to cast the context first before using it
10         (*temp) = IORD_ALTERA_AVALON_PIO_EDGE_CAP(KEYS_BASE);
11         // clear the edge capture register
12         IOWR_ALTERA_AVALON_PIO_EDGE_CAP(KEYS_BASE, 0);
13         printf("button: %i\n", *temp);
14 }
15
16 int main(void)
17 {
18         int buttonValue = 1;
19         void* context_going_to_be_passed = (void*) &buttonValue; // cast before
   passing to ISR
20
21         // clears the edge capture register
22         IOWR_ALTERA_AVALON_PIO_EDGE_CAP(KEYS_BASE, 0);
23
24         // enable interrupts for all buttons
25         IOWR_ALTERA_AVALON_PIO_IRQ_MASK(KEYS_BASE, 0x7);
26
27         // register the ISR
28         alt_irq_register(KEYS_IRQ,context_going_to_be_passed,
   button_interrupts_function);
29         while(1); // need this to keep the program alive
30         return 0;
31 }
```

Listing 5 – Updating the buttons using an ISR

The circles in yellow show how the context is being initiated in the main function (line 18 and 19). It is then registered/passed to the interrupt handler (line 28 and line 7) and when interrupt happens, it changes to the current button value as shown in line 9 and 10. The circles in green shows the function name is used to passed as a function pointer to the register function in lines 7 and 28.

4.4 Timer ISR

The timer ISR is quite different from the PIO ISRs. The prototype of the timer ISR is:

alt_u32 timer_isr_function(void* context)

The return value of the timer ISR is the timeout value (will be described in the next function) for the next timer iteration. When it is 0 then it implies that the timer will stop once the timeout has happened. The timer ISR does not need to be registered, unlike a general ISR. However, a timer must be started (which on its timeout generates an interrupt) using the following function:

int alt_alarm_start(alt_alarm* timer_pointer, alt_u32 time_to_run,
alt_u32 (*function_pointer_to_the_timer_isr) (void* context),
void* context);

The first argument is the timer pointer specifying/pointing to a timer. The second argument is the

time to run before the timeout happens (remember these are countdown timers). The third argument is a function pointer to the ISR. The last argument is the context, which is passed to the timer ISR. The function will return a negative value if the timer is failed to start. To use these timer functions, you need to include “sys/alt_alarm.h”

To count how many seconds since the timer start can be achieved by the following code:

```
1  #include <stdio.h>
2  #include "sys/alt_alarm.h"
3
4  alt_u32 timer_isr_function(void* context)
5  {
6        int *timeCount = (int*) context;
7        (*timeCount)++;
8        printf("time:%d\n", *timeCount);
9        return 1000; // the next time out will be 1000 milli-seconds
10 }
11
12 int main()
13 {
14       alt_alarm timer;
15       int timeCountMain = 0;
16       void* timerContext = (void*) &timeCountMain;
17             // start the timer, with timeout of 1000 milli-seconds
18             alt_alarm_start(&timer, 1000, timer_isr_function, timerContext);
19
20       while(1)
21       {
22             if ( timeCountMain == 10)
23             {
24                   alt_alarm_stop(&timer);
25                   usleep(1000000);
26                   alt_alarm_start(&timer, 1000, timer_isr_function, timerContext);
27                   usleep(1500000);
28             }
29       }
30       return 0;
31 }
```

Listing 6 - Example of a timer ISR

Line 14 first creates a timer instance, which will be used to start the timer. Line 4 to line 10 defines the timer ISR, which increments the timeCount passed from main (it is called timeCountMain in the main function). The timer is started on line 18. A while loop is required to keep the program alive. When the time count reaches 10, the timer stops (at line 24). Then, using **usleep** function, the processor will sleep for 1000000 micro-seconds (1 second). Then the timer will start again. **QUESTION: Why there is another usleep after the timer starts?**

5. Programming technique

5.1 Polling

In contrast to section 4.3, where a button value is checked only when interrupt happens, there is another programming technique called **polling**. It can achieve the almost same effect. The equivalent code to Listing 5 is as follows:

```
1  #include "system.h" // to use the symbolic names
2  #include "altera_avalon_pio_regs.h" // to use PIO functions
3  int main(void)
4  {
5         int buttonValue = 1;
6         while(1)
7         {
8               buttonValue = IORD_ALTERA_AVALON_PIO_DATA(KEYS_BASE);
9         }
10        return 0;
11 }
```

Listing 7 – Update buttons with polling

However, with the polling technique, the processor will be busy checking the value of button at all time. It is fine if the processor only does this simple job.

QUESTION: Even though the system in Listing 7 is functionally correct, what will be the draw back compared to Listing 5?

5.2 Pointers

Pointers are values that store the memory locations of another value in memory. For example, Figure 2 shows an example of a pointer. The integer variable "b" is declared with an initial value of 1234 (line 3). To store the address to which the variable b is located, the integer pointer variable "a" is declared with the initial value being the address of the variable "b" (line 4). When the pointer is dereferenced (to obtain the value stored on the address in memory) and printed out on to the console, the value of "b" (1234) is displayed (line 5).

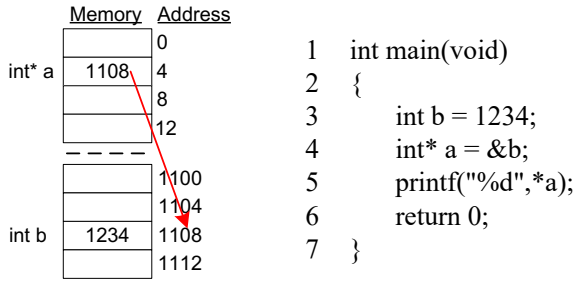

Figure 2 - Example of a pointer

6. Exercise

In this exercise, you will use a timer, interrupts, and buttons to increment a counter. Following are the specifications:

1. Whenever the button Key0 is pushed, the counter should increment.
2. Whenever the button Key1 is pressed and held down, the counter should increment every 500ms.
3. The current value of the counter should at all times be displayed on the LCD (and any other output if you wish).

7. SCCharts

SCCharts is a tool that can be used to model synchronous systems, especially those which are highly reactive to their environment, and safety-critical systems. It uses a text-based syntax to describe the model, but is able to create a graphical representation of the model, and convert the model into executable code in a range of languages through model-to-model transformations.

7.1 Opening SCCharts

To open SCCharts, simply open the start menu and search for "**sccharts**". An option should appear for "**Kieler - SCCharts**" should appear. Clicking on this item should open up a splash screen as shown in Figure 3.

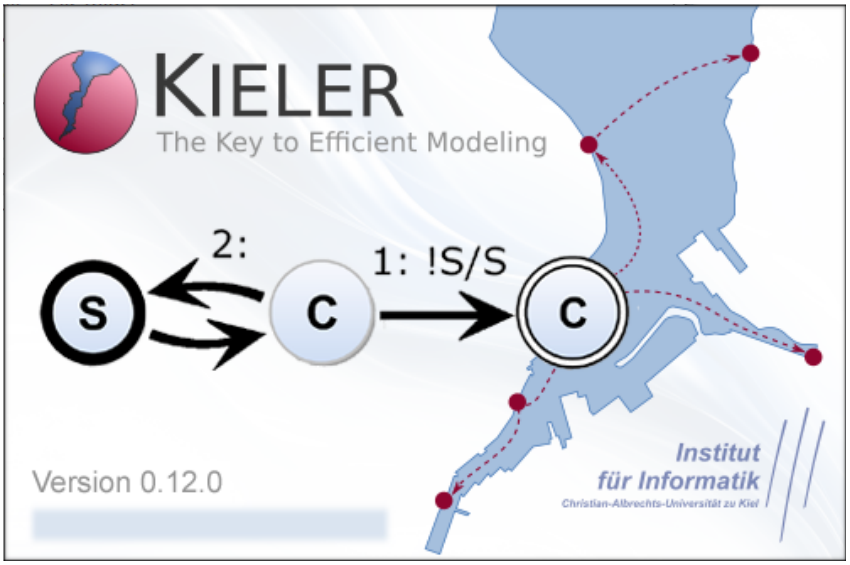

Figure 3 – Kieler Splash Screen

After the splash screen has finished loading, you may be asked to set the workspace location, as is typical in any Eclipse-based IDE. For example, set the workspace to be "C:\Users\{your_upi}\kieler-workspace" as shown in Figure 4 and click "OK".

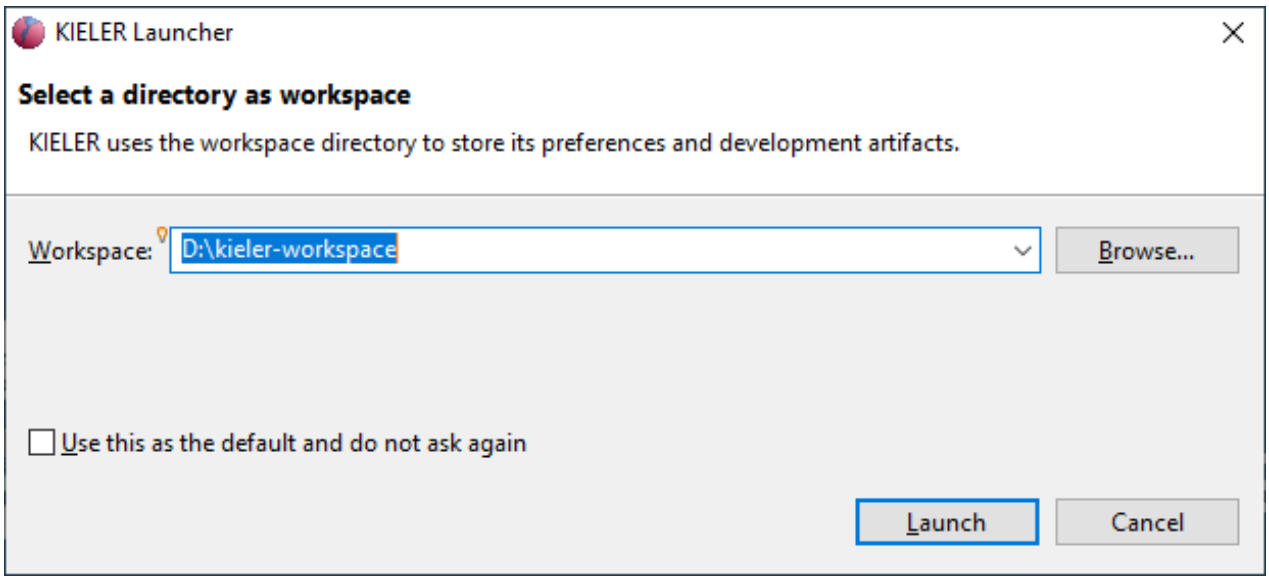

Figure 4 - Setting the Kieler Workspace

7.2 Creating a Project

After you've set the workspace and Kieler has successfully started, you will be shown a default welcome screen. We now need to create our SCCharts project, which is accomplished by going to "File" -> "New" -> "Project" as shown in Figure 5. This will bring up a dialog box as shown in Figure 6 where you need to click "General" -> "Project" before clicking "Next" to bring up the final box of Figure 7. Here, you need to set the name of the project want to create, let's call it "cs303_lab2". Note that SCCharts is unhappy if you have project names with hyphens, so refrain from using them. You can also optionally select where you want your project to be stored if you wish for it to be stored somewhere other than your workspace (e.g. a git repo). Simply click "Finish" and your project will be created.

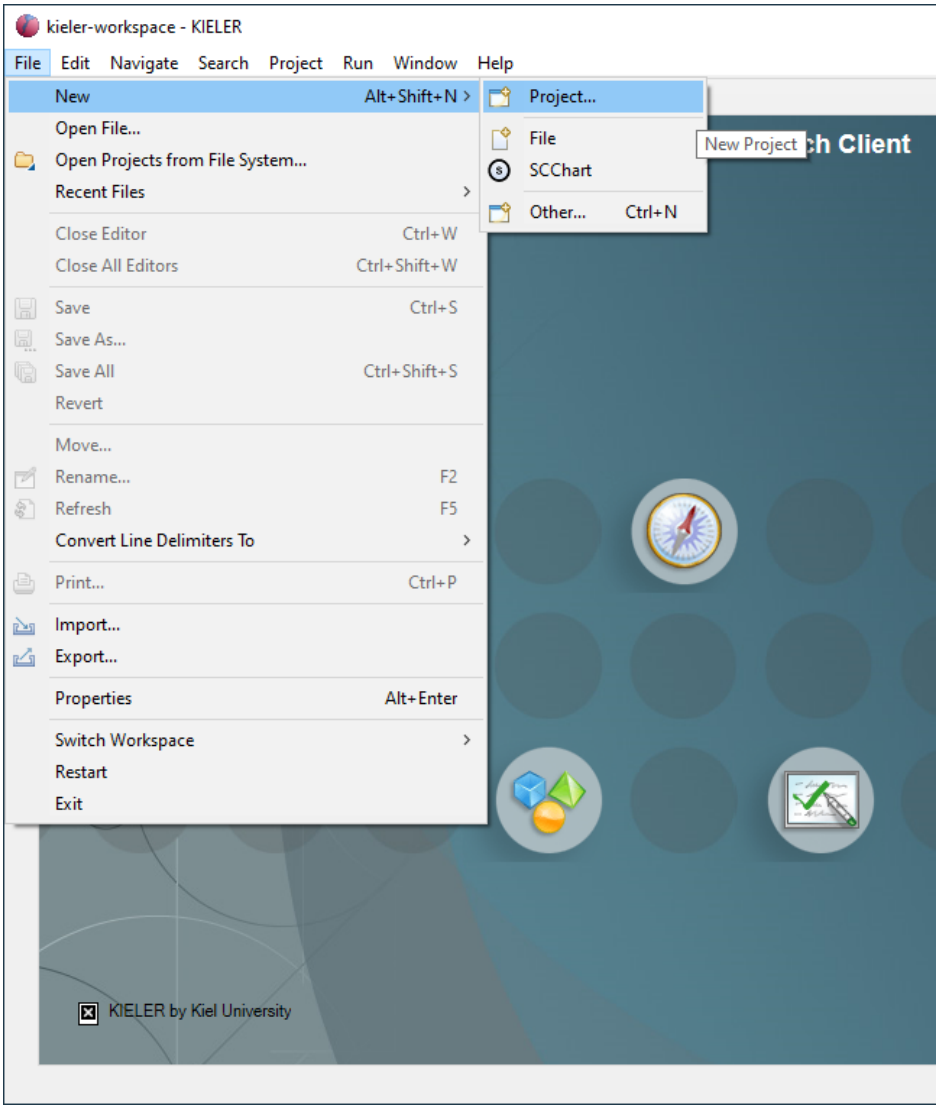

Figure 5 - Creating a new SCCharts project

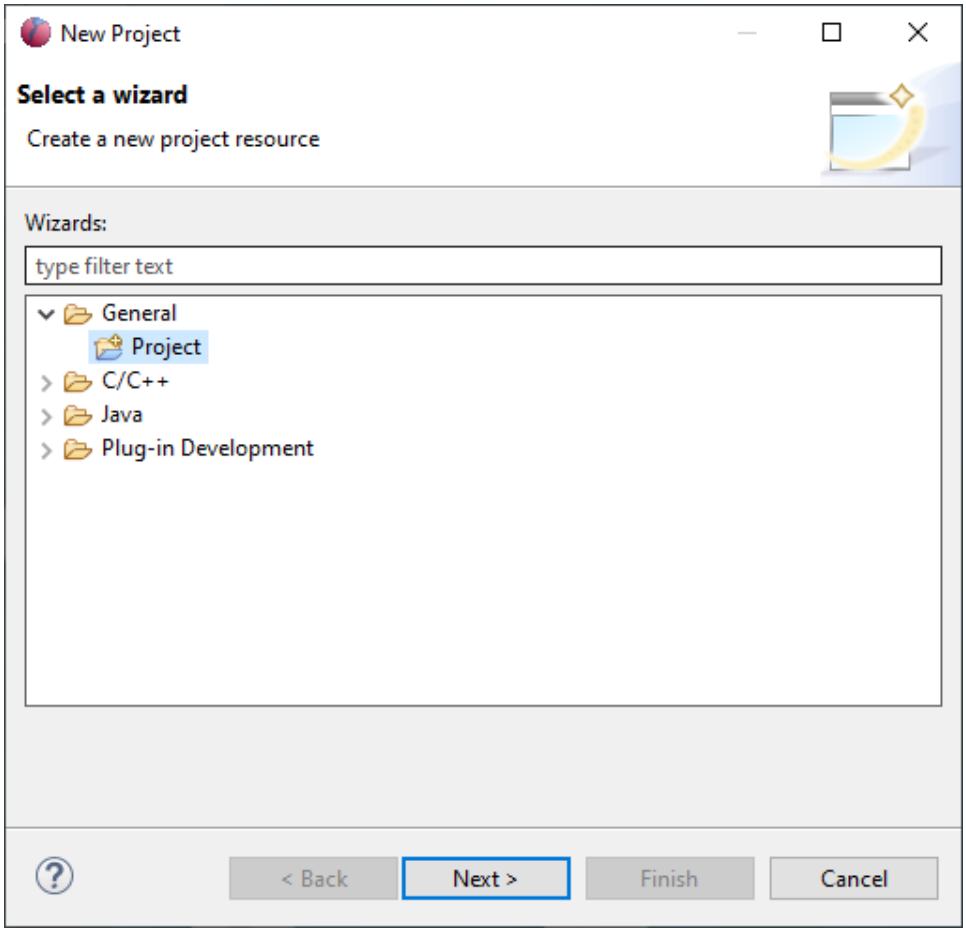

Figure 6 - Creating a new SCCharts project

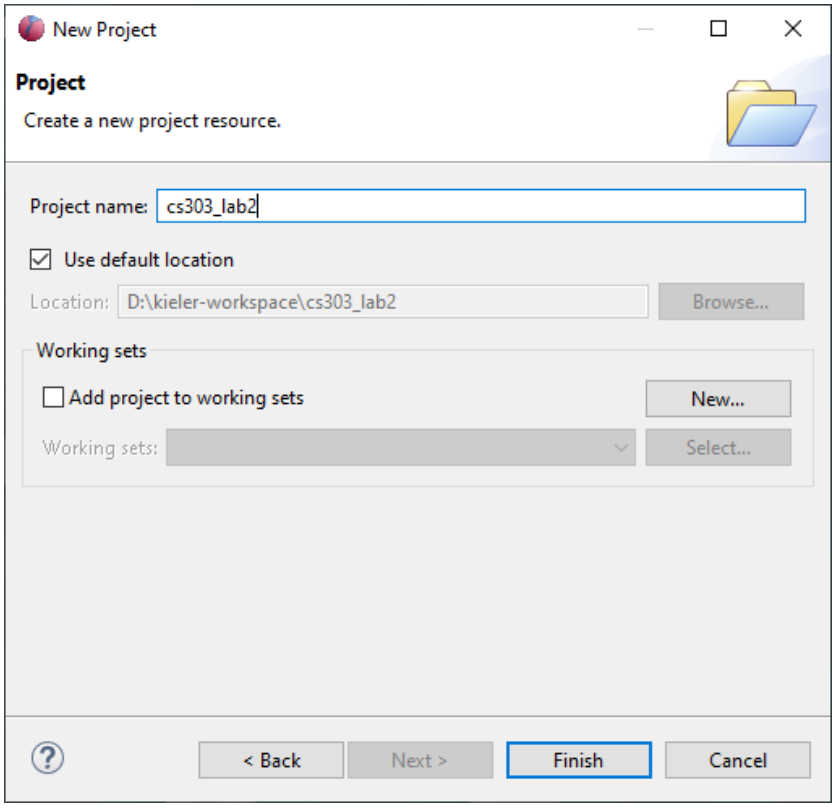

Figure 7 - Setting the Project name and location

After the project has been created, we need to make the file containing our SCChart. Go to “File” -> “New” -> “SCChart” as shown in Figure 8. If this option is not visible, you need to click on

"Other…" and look under the "Other" category. This will bring up a dialog for you to select the project that this file will be added to along with the file name, as shown in Figure 8. Select our "cs303_lab2" project, and enter a name for the file such as "cs303_lab2.sctx" (the .sctx extension will be added automatically), before clicking "Finish". This will then bring up a view similar to that shown in Figure 9. In SCCharts, your automata are described in a textual format on the left (known as an SCTX file), while the diagram on the right is automatically generated from the textual file.

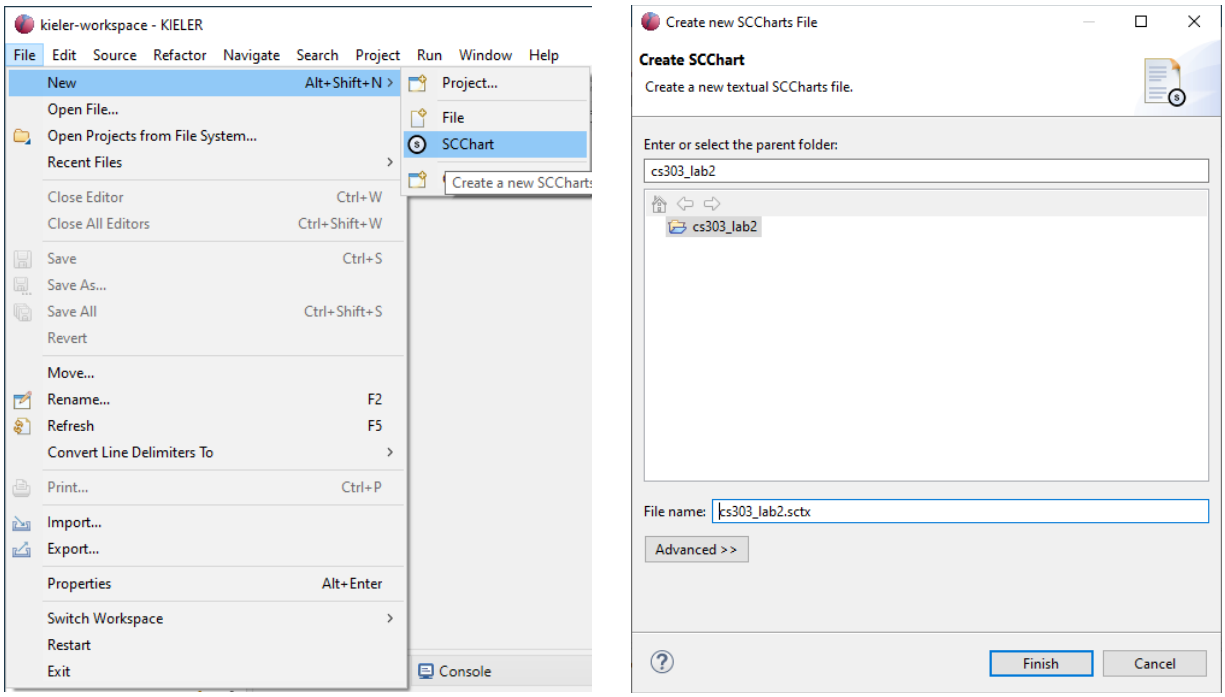

Figure 8 – Creating a SCTX file

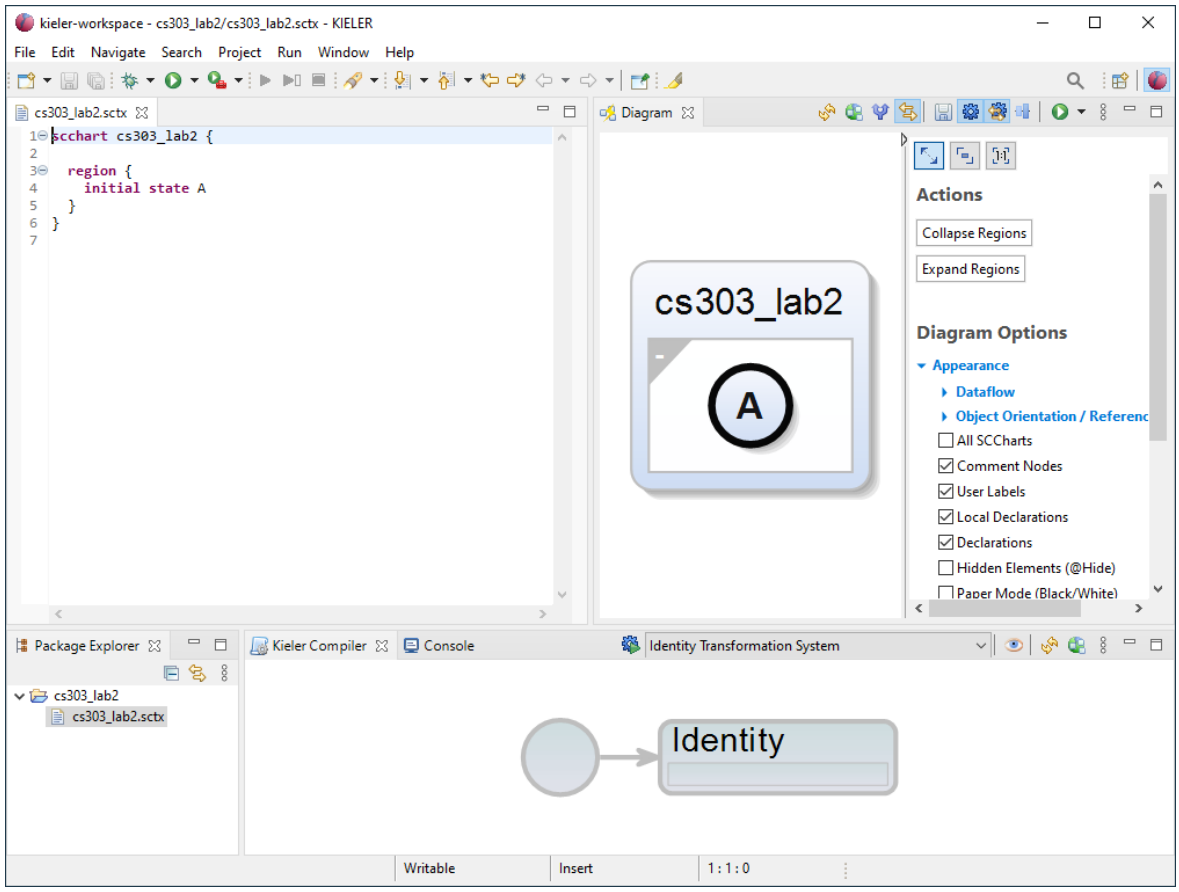

Figure 9 - Default SCT file

7.3 ABRO

We are going to create a very simple example, which is common in embedded system design, known as ABRO. Here, we want to satisfy the following statement:
"Emit output **O** each time the inputs **A** and **B** have occurred. Repeat each time that input **R** occurs."
In our case, we will make one small modification and require that **O** will be sustained until **R** occurs. Further, **R** is of higher priority than either **A** or **B**.

7.3.1 HandleA

```
1  scchart cs303_lab2 {
2    input bool A
3
4    region HandleA {
5      initial state waitA
6      if A go to doneA
7
8      final state doneA
9    }
10 }
```

Listing 8 - HandleA Code

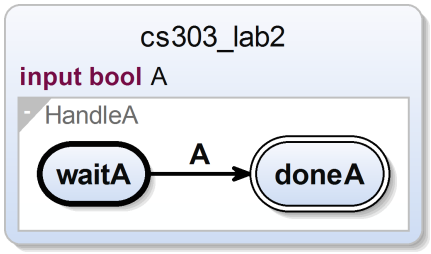

Figure 10 – HandleA Diagram

To start with, we will just model the simple operation of "waiting for input A to occur". This can be represented by a simple 2-state automata with "wait" and "done" locations. The diagram of what we want to achieve can be seen in Figure 10, while the code that creates it is shown in Listing 8. Here, we have an initial state named "waitA", which has a single transition (denoted by the "go to") to a state called "doneA" whenever an input "A" is present. We also declare the state "doneA" to be "final", meaning that execution of this region will halt when it is reached.

7.3.2 ABO

```
1  scchart cs303_lab2 {
2    input bool A, B
3    output bool O = false
4
5    initial state WaitAandB {
6      region HandleA {
7        initial state waitA
8        if A go to doneA
9
10       final state doneA
11     }
12
13     region HandleB {
14       initial state waitB
15       if B go to doneB
16
17       final state doneB
18     }
19   }
20   do O = true join to done
21
22   final state done
23 }
```

Listing 9 - ABO Code

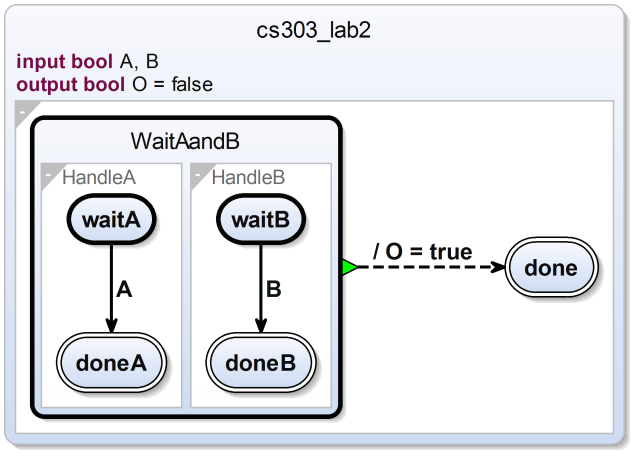

Figure 11 - ABO Diagram

Next, we will extend this to both include the waiting for input “B” and emitting the output “O” when required. HandleB is simply a clone of the previous HandleA with it just referring to a different input (“B”). These two automata are executed concurrently by declaring the two regions at the same level. Further, we will implement the outputting of “O” by having a **termination transition** (“join to”) to a final state “done” where it will output “O”. Such termination transitions are only enabled once the previous state has finished, i.e. when both HandleA and HandleB have reached their final states. Listing 9 and Figure 11 illustrate these.

7.3.3 Complete ABRO

```
1   scchart cs303_lab2 {
2     input bool A, B, R
3     output bool O = false
4
5     initial state ABO {
6       entry do O = false
7
8       initial state WaitAandB {
9         region HandleA {
10          initial state waitA
11          if A go to doneA
12
13          final state doneA
14        }
15
16        region HandleB {
17          initial state waitB
18          if B go to doneB
19
20          final state doneB
21        }
22      }
23      do O = true join to done
24
25      final state done
26    }
27    if R abort to ABO
28  }
```

Listing 10 - ABRO Code

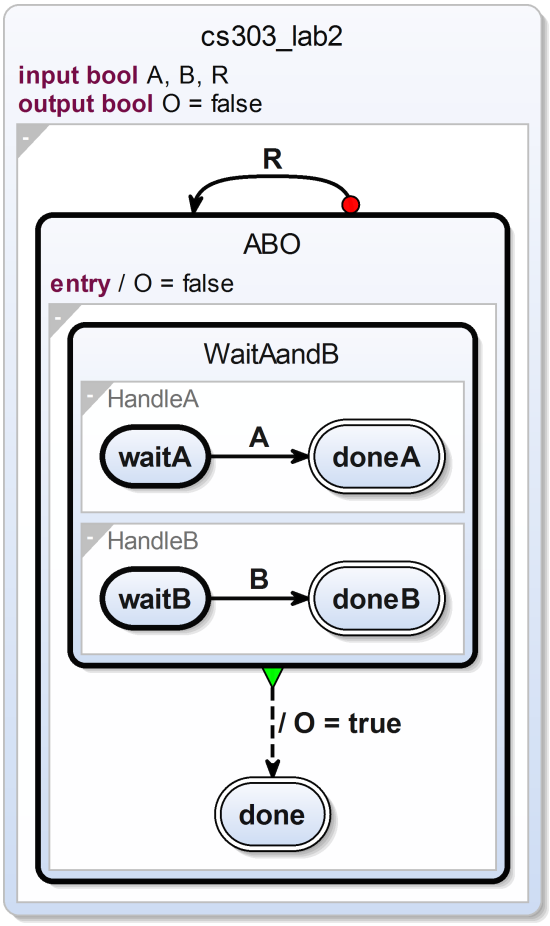

Figure 12 - ABRO Diagram

Finally, we will add support for the reset input “R”. To do this, we simply encapsulate the previous “ABO” logic inside another state (let’s call it “ABO”) with a self-transition that resets the logic. This self-transition needs to be of higher priority than any of the other internal logic and so we create what is called a strong abort (“abort to”). Strong aborts will always happen at the beginning of any execution, meaning that they have the highest priority. We also need to add an “entry” condition to ABO which resets the output “O” to false whenever it is started. The final ABRO system specification can be seen in Listing 10 and its generated diagram in Figure 12.

7.4 Generating Code

In order to generate C Code for our model, we need to use the “Kieler Compiler” box at the bottom of the screen. Here, you are able to select what displays on the right hand side of the screen where our diagram currently is. The Compiler Selection options allow you to view what the compiler is doing at each point in the process, for example checking for dependencies.

There are multiple compiler pathways supported by SCCharts, for generating C Code, Java, etc. We are interested in C code generation so in the drop-down box at the top-right of this panel select “Netlist-based Compilation (C)” (it likely defaulted to “Identity Transformation System”) and invoke a compilation process by clicking the button to the left of this. You should now end up with something that looks like Figure 13.

To view the generated C code, simply click on the box labelled “C Code” in order to generate C code for the model. Your window view should now change to look similar to Figure 14, consisting of two files: a header file which contains a struct for the model execution and associated function declarations, and a source file which provides the definitions for these functions. To see the entire generated code rather than just a preview, click the “Open in Editor” text at the top of the preview. This will allow you to view and copy the entire generated code for use wherever you need, such as in NIOS as we will now do.

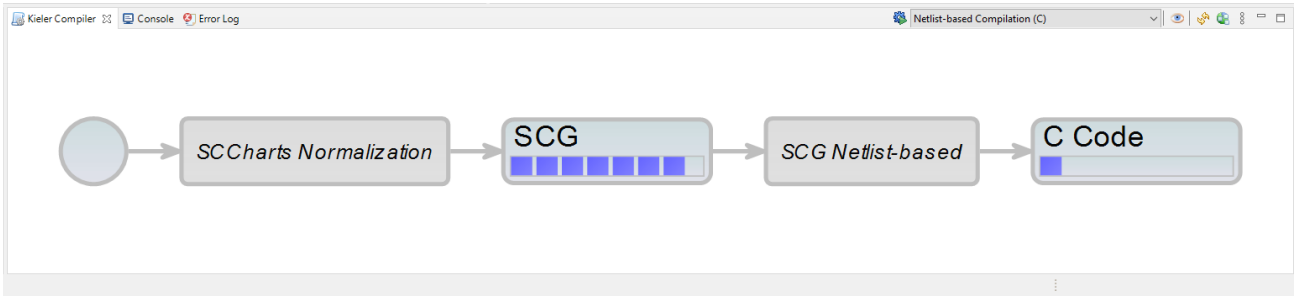

Figure 13 - KIELER Compiler Selection for C Code

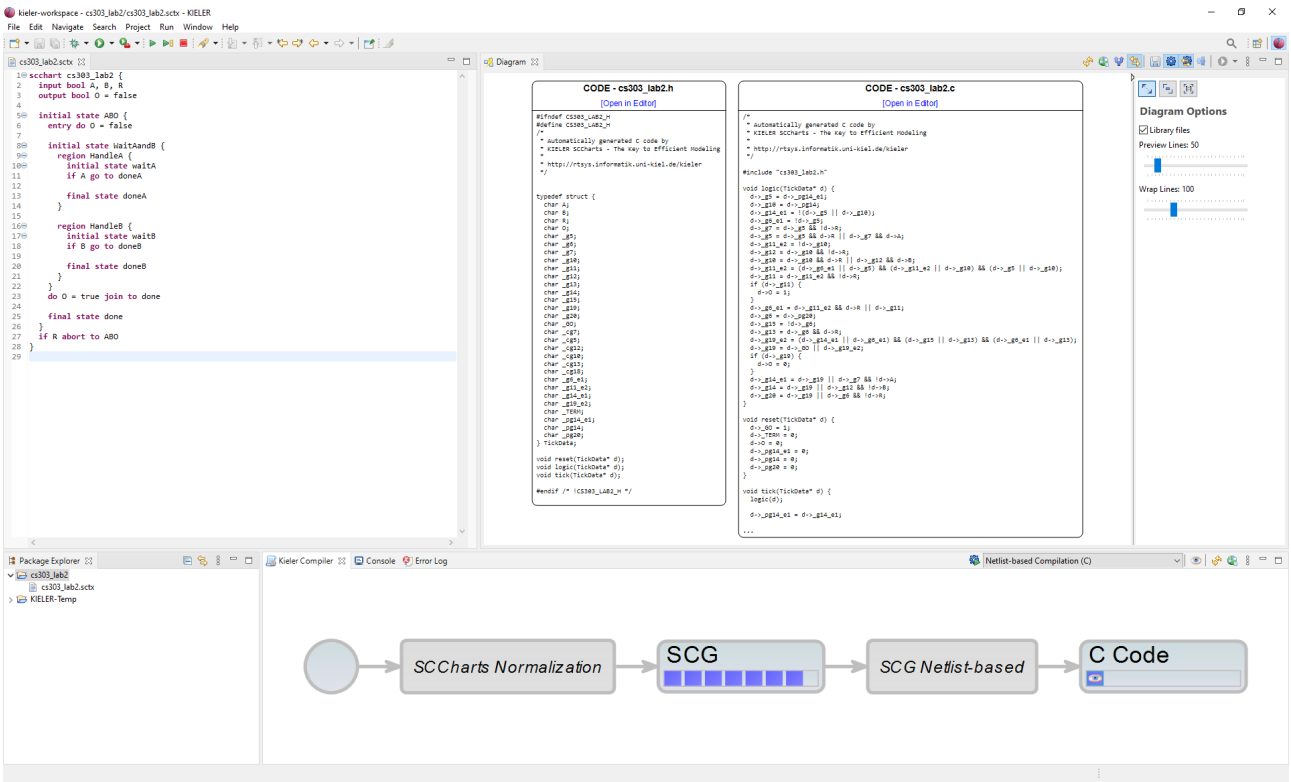

Figure 14 - KIELER after changing compilation target

8. Executing SCCharts on Nios II

SCCharts operates through what is known as “reactive” semantics. These semantics mean that the generated code consists of two functions: a “reset” function which is used to initialize the model, as well as a “tick” function which does a single step of execution. You can think of each “step” like each currently active state machine being given the option of taking a single transition. For example, when we have “HandleA” and “HandleB” operating in parallel, a single “tick” involves both “HandleA” checking if “A” is present and “HandleB” checking if “B” is present.

8.1 Implementation in Nios II Build Tools

In order to add our new C code and Header file to our Nios project, simply right click on the project, proceed to “New”, and select either “Header File” or “Source File” depending on which one you want to add. It will then ask you for the name of each file, let’s name our Header File “sccharts.h” and our Source File “sccharts.c” as shown in Figure 15 and Figure 16. Note that after adding each file the editor will refresh the Makefile to include your newly added file, which may take some time.

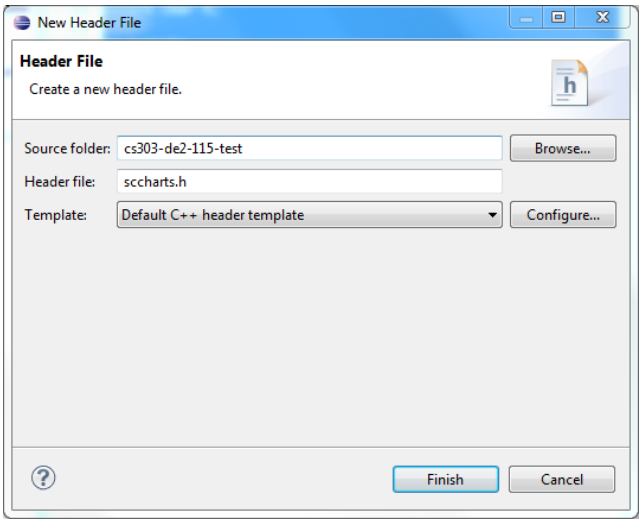

Figure 15- Adding a new Header File

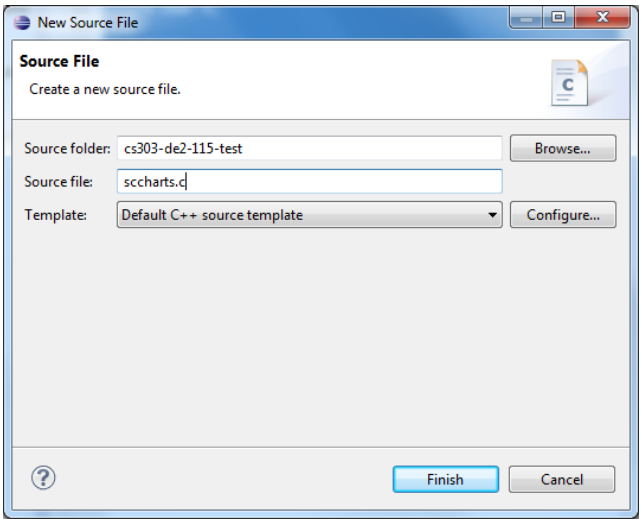

Figure 16 - Adding a new Source File

Inside the “sccharts.h” file we will copy the generated header from SCCharts, while for the “sccharts.c” file, just copy the generated C code from SCCharts.

Now, we just need to use these new functions in our main C code. All we need to do is simply provide a “#include” to the header file at the top of our main program, call the “reset” function before our while loop, and then call the “tick” function inside each iteration of the while loop.

We also need to map each of the inputs and outputs to the mechanical aspects of the board. Here, we’re going to do the following:

- A is Key 2
- B is Key 1
- R is Key 0
- O is Red LED 0 (or all of them, if you want to have more fun)

The keys can be read in through a method like the polling we did before. You’ll need to figure out how to get the value for a **single** key, rather than an integer that corresponds to all the keys. Think about bitwise manipulation and remember that the keys are **active low**.

8.2 Exercise

Implement the above main C file so that the ABRO example is able to run on your processor as described above. An incomplete file is given to you as an example in Listing 11, build off of this to complete the task.

```
1   #include <system.h>
2   #include <altera_avalon_pio_regs.h>
3   #include <stdio.h>
4   #include "sccharts.h"
5
6   int main()
7   {
8         // Create the struct
9         TickData data;
10
11        // Initialise
12        reset(&data);
13
14        while(1)
15        {
16              // Fetch button inputs
17              // A is Key 2, B is Key 1, R is Key 0
18              // Remember that keys are active low
19
20
21              // Do a tick!
22              tick(&data);
23
24              // Output O to Red LED
25
26        }

          return 0;
    }
```

Listing 11 – Incomplete Main C File for the ABRO example