Creating IEC 61499 Function Block Applications in 4diac

This tutorial demonstrates how to design your first function block application in 4diac. 4diac is an acronym for *For Distributed Industrial Control,* the open source framework to develop, deploy and visualize industrial applications for distributed automation and control. It is based on the IEC 61499 Function Block Reference Software Architecture standards. The home for 4diac is at https://www.eclipse.org/4diac/

The 4diac framework provides:

- *The 4diac-ide Integrated Development Environment (IDE)* that is built upon the Eclipse development framework. 4diac-ide provides a set of design surfaces that allow you to design new function block applications by wiring together pre-existing function block from a library. You can also create your own functions blocks and wire them into your applications.

- *The FORTE runtime environment* is a small portable implementation of the standard IEC 61499 runtime environment specified in the IEC standards. It targets 16-bit and 32-bit embedded control devices implemented in C++. It supports online-reconfiguration of its applications and the real-time execution of all function block types provided by the IEC 61499 standard. FORTE provides a scalable architecture which allows 4diac FORTE to adapt to the needs of your application. Applications can consist of any IEC 61499 standard element including Basic Function Blocks (BFBs), Composite Function Blocks (CFBs), and Service Interface Function Blocks (SIFBs). It also supports custom adapters and IEC 61499 subapplications.

What this tutorial covers

This tutorial complements the 4diac tutorials available at https://www.eclipse.org/4diac/en_help.php. By the end of this tutorial you should be able to:

- Create a function block application using the 4diac IDE. This tutorial creates a small Alarm Clock function block application to illustrate the design and development steps.
- Identify and use the most useful pre-built function block types available from the 4diac Type Library.
- Create a new function block, based on one of the primitive IEC 61499 function block types.
- Export your function block types as C++ classes and add them to the 4diac Type Library.
- Use CMake-gui to create the C++ Makefile required to compile your application.
- Use C++ commands to test and debug your application.

How to get the best out of this tutorial

The 4diac on-line documentation is a great place to learn how to use 4diac This tutorial uses the first parts of that documentation which provide a valuable introduction to IEC 61499, 4diac, and FORTE. It is important to understand the fundamental concepts first and the 4diac site provides an up-to-date reference to the latest 4diac IDE features.

This tutorial is divided into a number of sections that you should complete in the suggested order. However, in the AUT EMSOFT laboratory, we have developed an alternative 4diac C++ Tool Chain that makes it easier to manage multiple, simultaneous projects on the same development machine. This requires you to configure the C++ Makefiles differently. By the time you get to that part of the tutorial, the reasons for using this alternative tool chain will be a lot clearer.

Step 1 – Introducing IEC 61499

This section is an introduction to programming Programmable Logic Controllers using IEC 61499. It explains what a function block is and how you can recognise the Event and Data interfaces on a function block type.

https://www.eclipse.org/4diac/en_help.php?helppage=html/before4DIAC/iec61499.html

Step 2 – Introducing 4diac and FORTE

This section explains the relationship between the 4diac IDE and the FORTE runtime.

https://www.eclipse.org/4diac/en_help.php?helppage=html/before4DIAC/4diacFramework.html

Step 3 – Installing a 4diac development environment

The 4diac documentation explains how to install 4diac and FORTE on a development computer. That can be complex and time-consuming if you are unfamiliar with the CMake tools and the GCC C++ complier. An easier way to start developing more quickly is to install the pre-built development environment image that was created by AUT EMSOFT.

Most embedded system projects run under Linux. The pre-built image contains an Oracle Virtual Box virtual machine that runs Ubuntu 20.04.

to be completed …

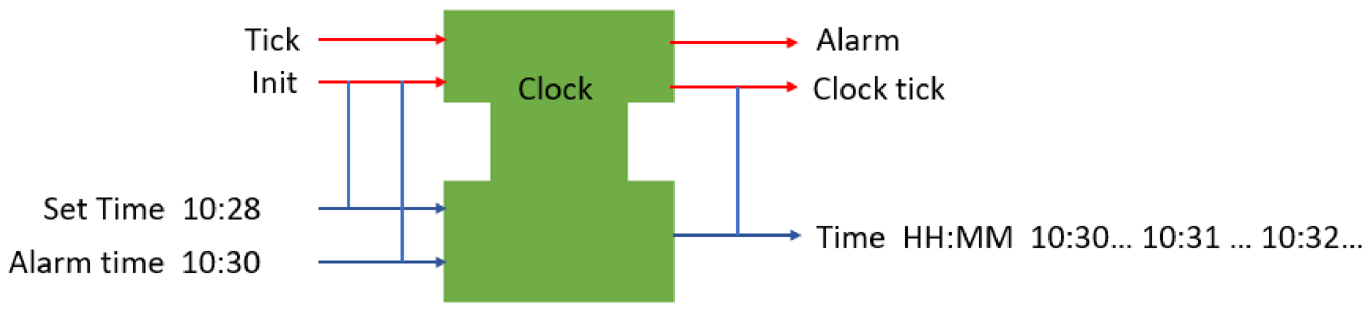
Step 4 – Creating the Alarm Clock function block application

This section explains how to create a new function block application project in 4diac. The new blocks needed to implement a simple Alarm Clock application are then created. Some blocks are already available in the Type Library and can be reused. While creating the alarm clock, you will learn how connect blocks together and how to navigate around the 4diac IDE.

This section concentrates only on *design*. The next section, Step 5, demonstrates how to export the function blocks as C++ classes and compile them so the application can be run.

Thinking about the functional requirements for the Alarm Clock

We are software engineers; we create requirements before we do anything else 😊. A simple sketch of what we want the alarm clock to do is useful. Software requirements are supposed to focus on the *what*, not the *how* of the application, but it is hard not to think about *how* the function block would meet the requirements when you are writing requirements. That is a common characteristic of embedded system design; it's just how engineer's think.



- The time is set on a data input.
- The time we want the alarm to ring is set on another data input
- There is an event called Tick that fires every second to make the clock tick.
- The time is output from a data output port.

At this stage, the names of the data inputs, outputs and events have not been assigned. That will come in the next design stage. You will probably have some other questions after thinking about this:

- What other requirements do we need?

- What other function blocks do we need to add to make this block work?

- What is a good naming convention for the data inputs, outputs and events?

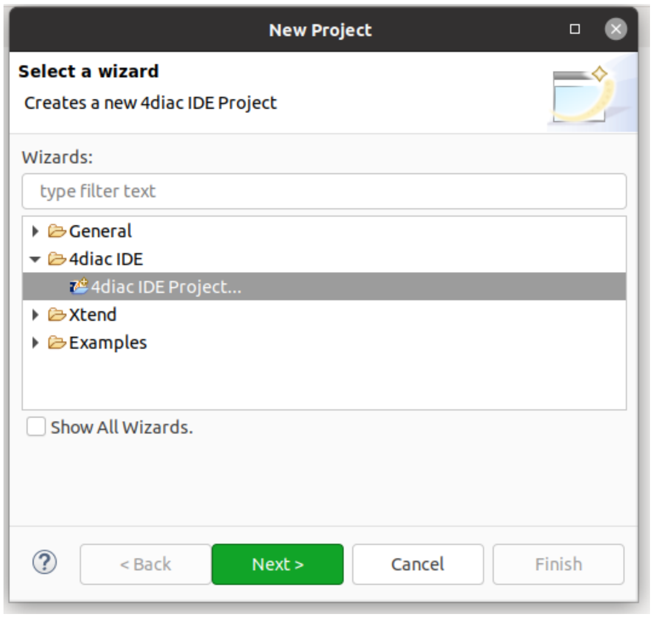
Sketching a draft state or *Execution Control Chart* will help you to think about the states and algorithms you need to make the clock tick properly. We will also have to check the time and the alarm time regularly.

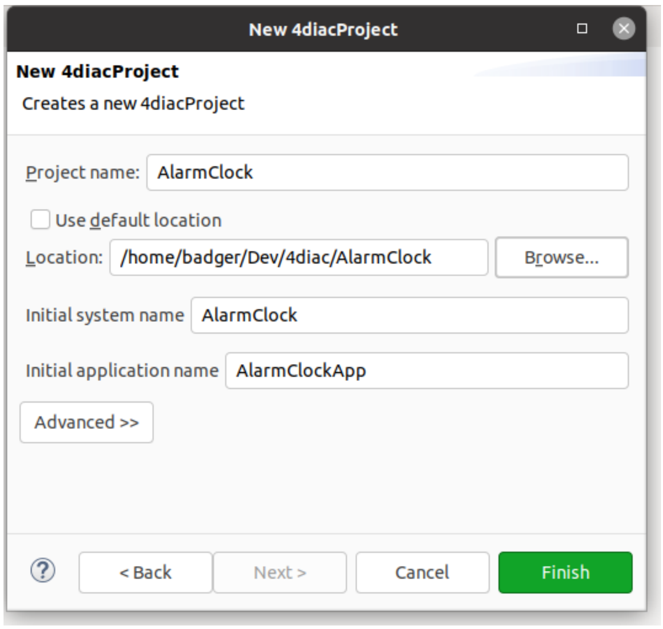
One important point about this example is that it is not a REAL clock. Specifying a static Set Time means that while you are coding and testing it, you do not need to reset the time to make it work each time you run it. That makes it easier since the runs are repeatable. In the sketch above, you can see that it will always be 10:28 am in the morning every time you start the function block application.

Creating a new AlarmClock project

- Start 4diac by single-clicking the icon on the 4diac toolbar:

- Start a new 4diac IDE Project with the menu option **File | New |Project** and click **Next.** Name the project **AlarmClock**. Choose a Location for the project if it does not suggest a default location then click **Finish.**





New 4diacProject

Creates a new 4diacProject

Project name: AlarmClock

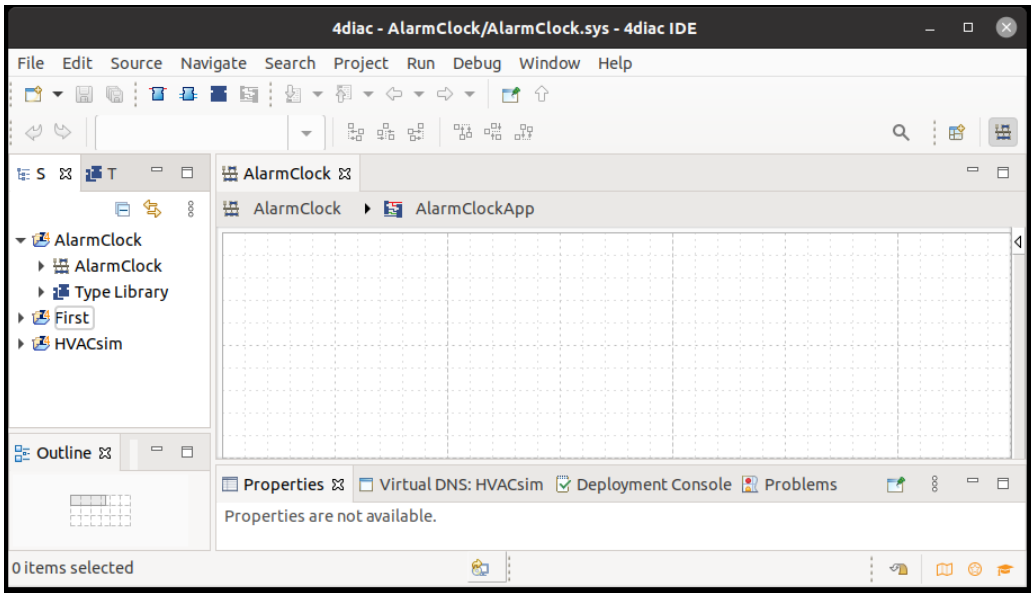
Use default location [ ]

Location: /home/badger/Dev/4diac/AlarmClock Browse...

Initial system name AlarmClock

Initial application name AlarmClockApp

- 4diac will then create the project and open the two IDE design surfaces you need:



Properties are not available.

0 items selected

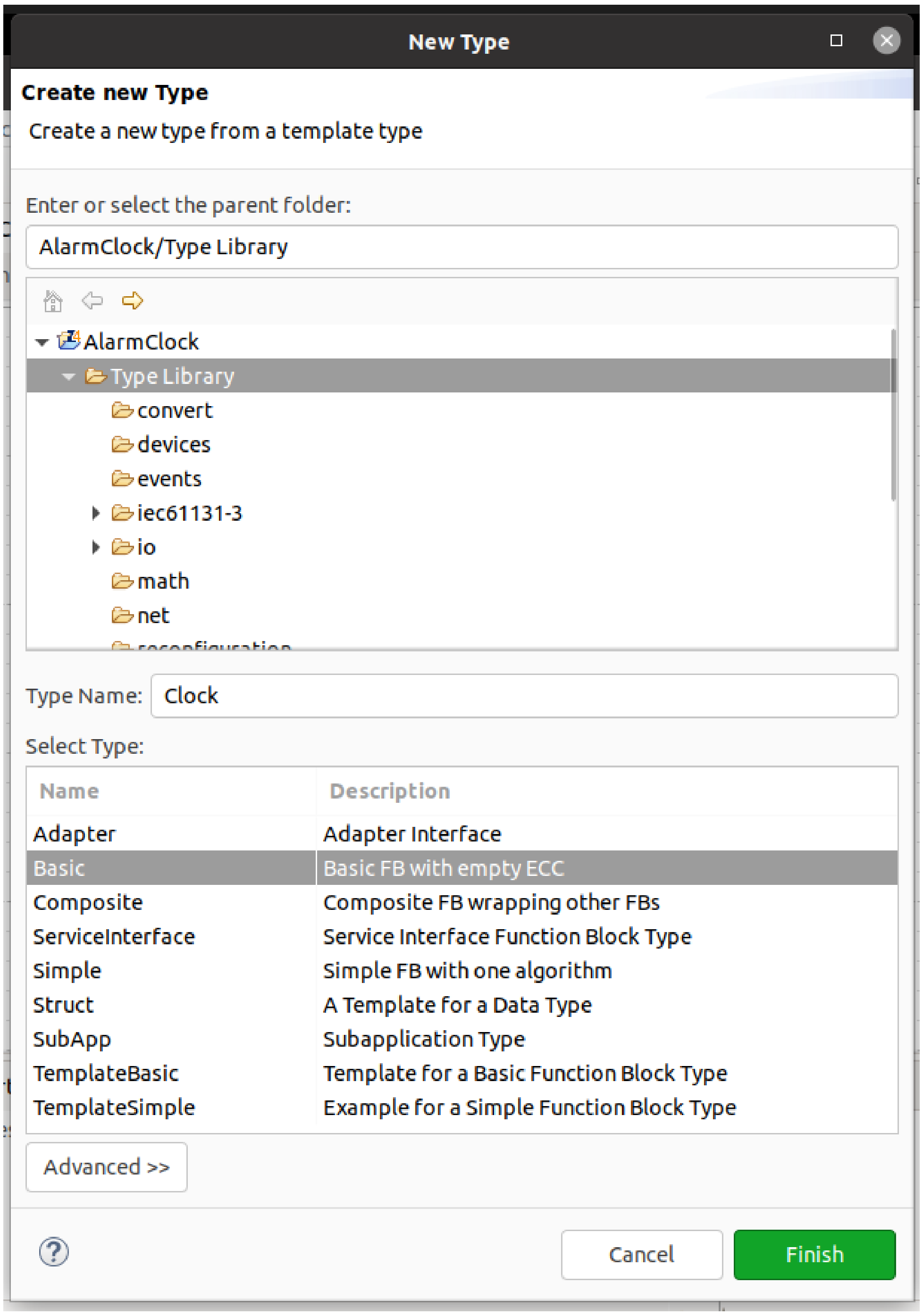
The **AlarmClock** design surface is where you select, arrange, and connect function blocks together. The second design surface is called **AlarmClockApp**. This one is needed later when you are ready to deploy your application before you run it.

Creating the Clock Function Block

Most of the clock and alarm functionality we need can be provided by a single, new function block. If you look in the **Type Library** branch of the tree in the left-hand **System** panel, you will see a library of existing function blocks.

However, none of them are exactly what we want for our clock. So, we need to create a new function block type called **Clock** using **File | New |Type:**

- Select the location of the new function block type as the root of the **Type Library.**
- The Type Name is **Clock**
- Select the Function Block template type as **Basic with empty ECC**. This will create a function block skeleton with a simple Execution Control Chart (ECC) that we can modify ourselves.
- Click **Finish**.



New Type

Create new Type

Create a new type from a template type

Enter or select the parent folder:

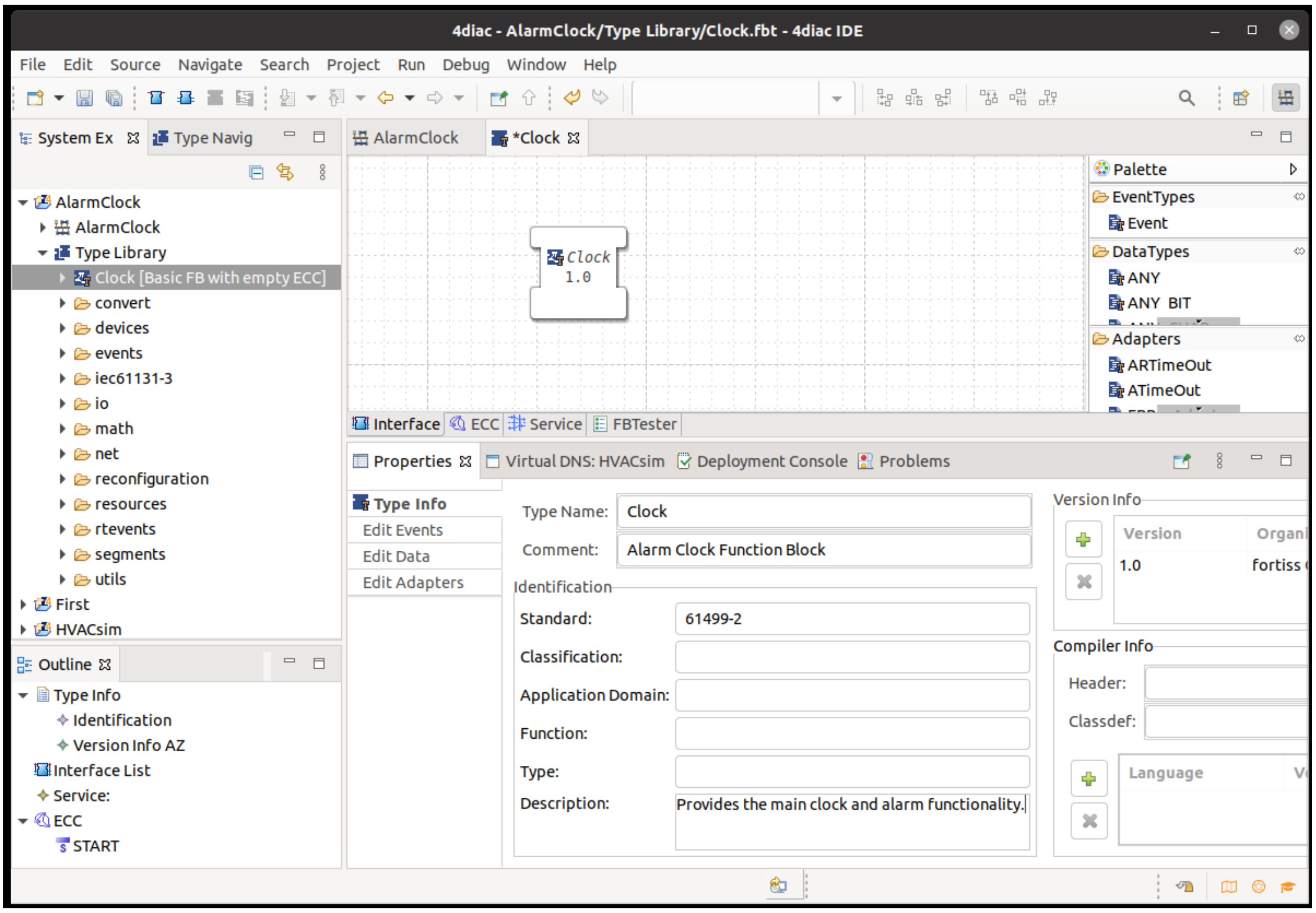
AlarmClock/Type Library

- AlarmClock
  - Type Library
    - convert
    - devices
    - events
    - iec61131-3
    - io
    - math
    - net

Type Name: Clock

Select Type:

Name	Description
Adapter	Adapter Interface
Basic	Basic FB with empty ECC
Composite	Composite FB wrapping other FBs
ServiceInterface	Service Interface Function Block Type
Simple	Simple FB with one algorithm
Struct	A Template for a Data Type
SubApp	Subapplication Type
TemplateBasic	Template for a Basic Function Block Type
TemplateSimple	Example for a Simple Function Block Type



Type Name:	Clock
Comment:	Alarm Clock Function Block

Identification

Standard:	61499-2
Classification:	
Application Domain:	
Function:	
Type:	
Description:	Provides the main clock and alarm functionality.

Version Info

Version	Organi
1.0	fortiss

Compiler Info

Header:	
Classdef:	

Language	V

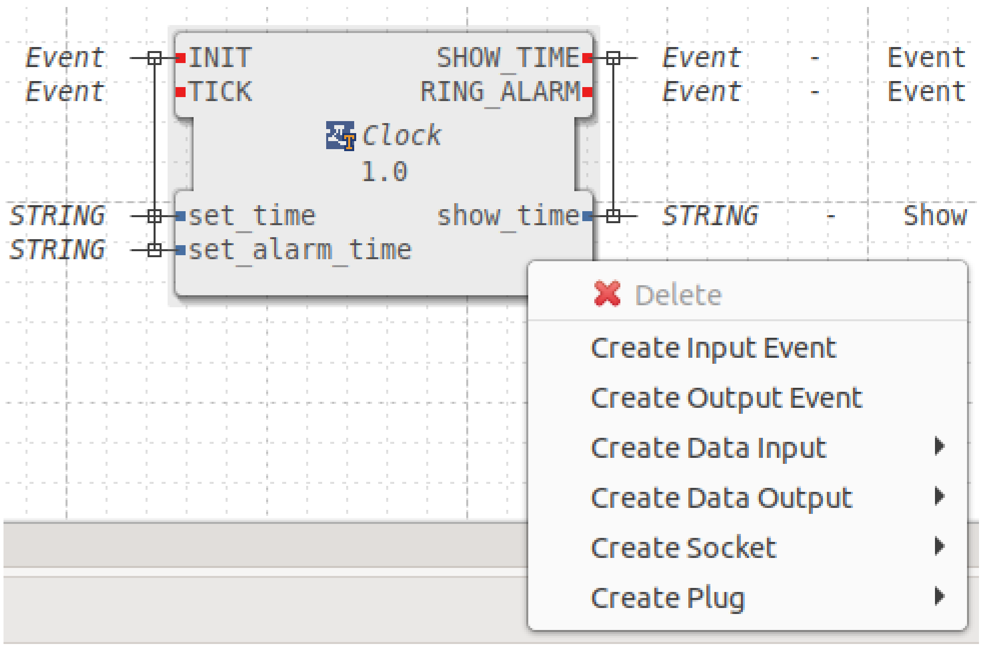
4diac opens a new design surface that allows you to customise the **Clock** function block. You can add documentation such as a **Comment** and a **Description** which will be saved with the block in the library. That makes the **Clock** block easier to reuse in other projects.

If the Clock block does not appear, then locate in the **System Explorer Type Library** branch and click on it to open it on a new design surface.

Adding data inputs, data outputs and events to the Clock Function Block

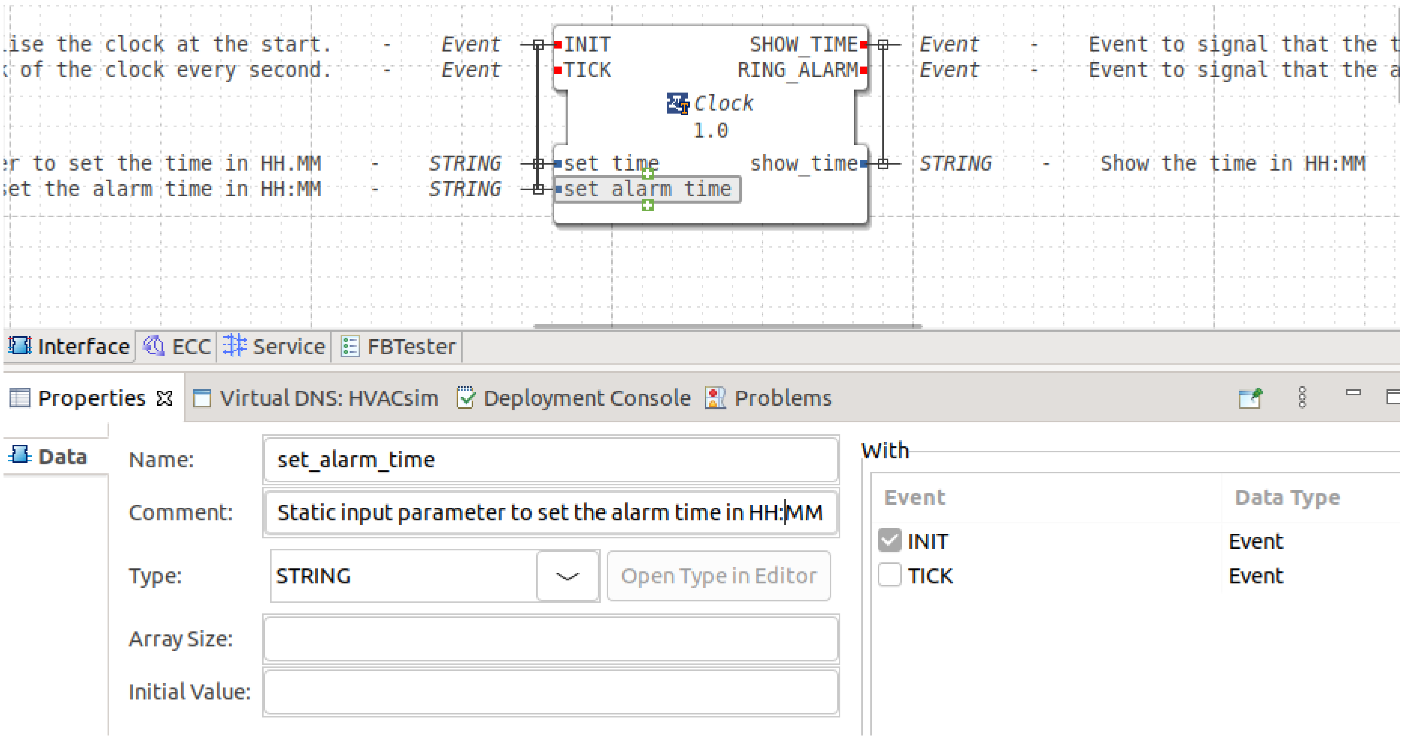
Data inputs, outputs, and events are created by right-clicking the corner of the block.

Specify the data type using the submenu.



When you highlight an input or event, the property box Interface tab will allow you to change the properties.

In the example below, the set_alarm_time data input is read when the input event INIT is triggered.



```
ise the clock at the start.    -   Event  INIT          SHOW_TIME  Event    -   Event to signal that the t
 of the clock every second.    -   Event  TICK          RING_ALARM Event    -   Event to signal that the a
                                               Clock
                                                1.0
r to set the time in HH.MM    -   STRING  set_time     show_time   STRING   -   Show the time in HH:MM
et the alarm time in HH:MM    -   STRING  set_alarm_time
```

Name: set_alarm_time

Comment: Static input parameter to set the alarm time in HH:MM

Type: STRING

Array Size:

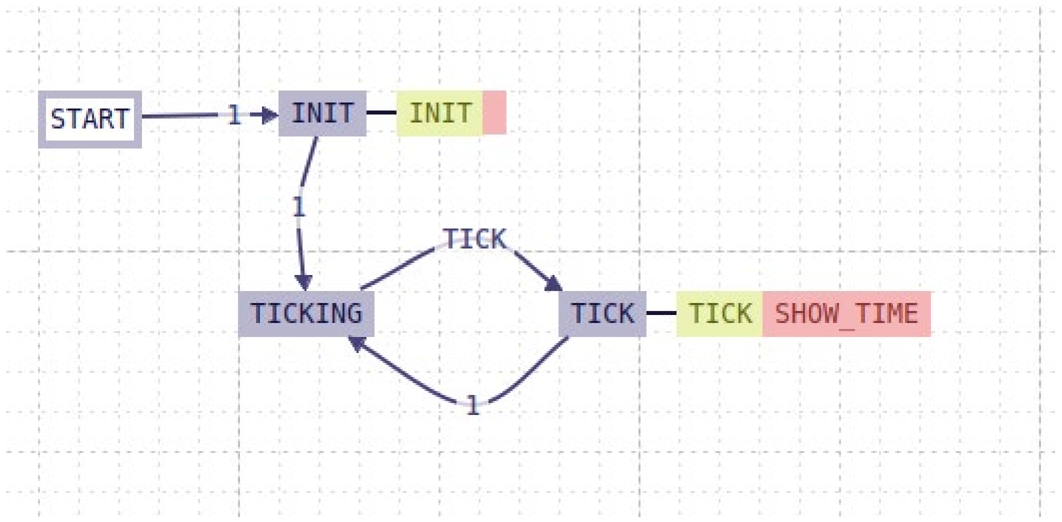
Initial Value:

With

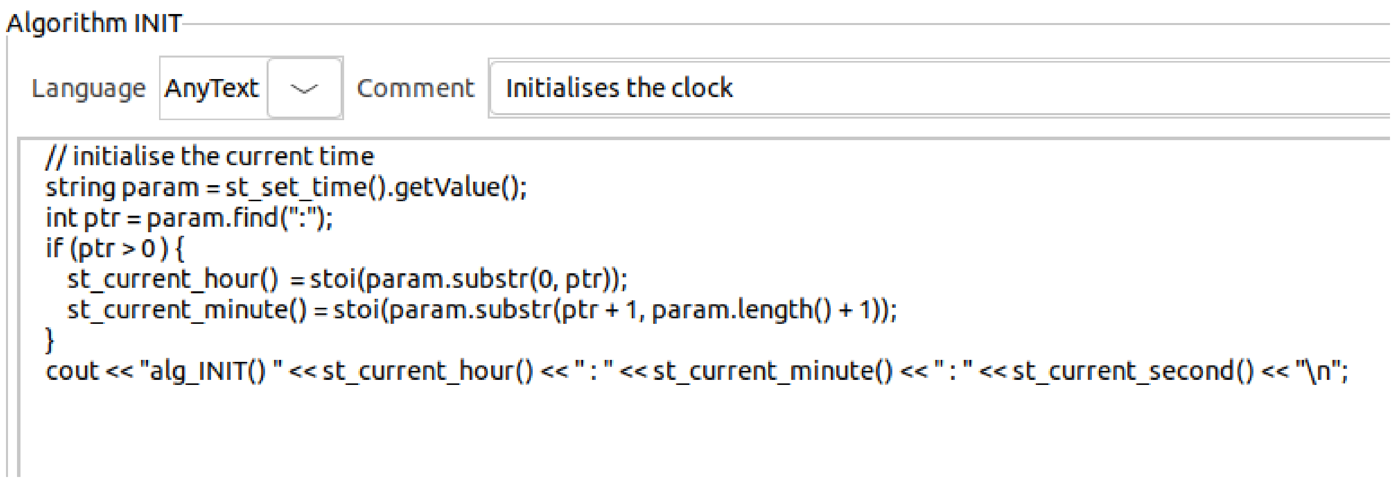
Event	Data Type
✓ INIT	Event
TICK	Event

Writing the Execution Control Chart and the Clock algorithms

The Execution Control Chart (ECC) controls what the function block does when each event is triggered

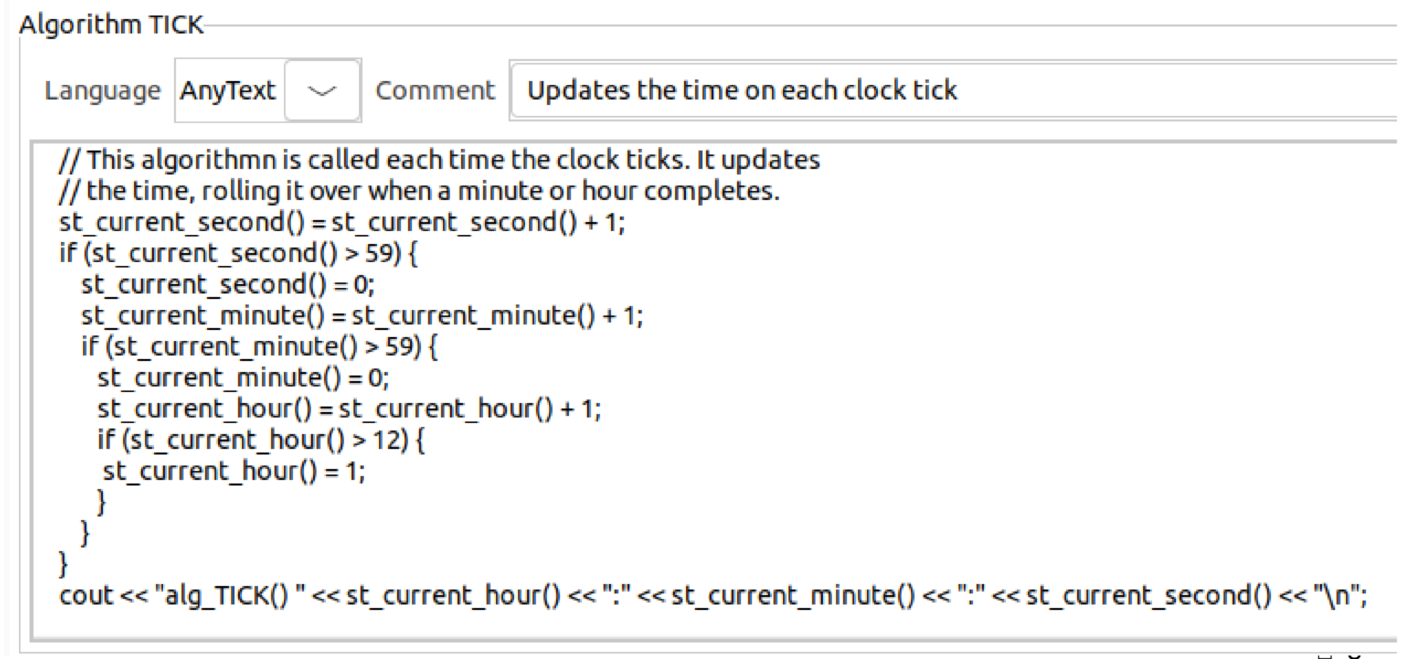


Two algorithms are used:



Initialises the clock

```
// initialise the current time
string param = st_set_time().getValue();
int ptr = param.find(":");
if (ptr > 0 ) {
   st_current_hour()  = stoi(param.substr(0, ptr));
   st_current_minute() = stoi(param.substr(ptr + 1, param.length() + 1));
}
cout << "alg_INIT() " << st_current_hour() << " : " << st_current_minute() << " : " << st_current_second() << "\n";
```

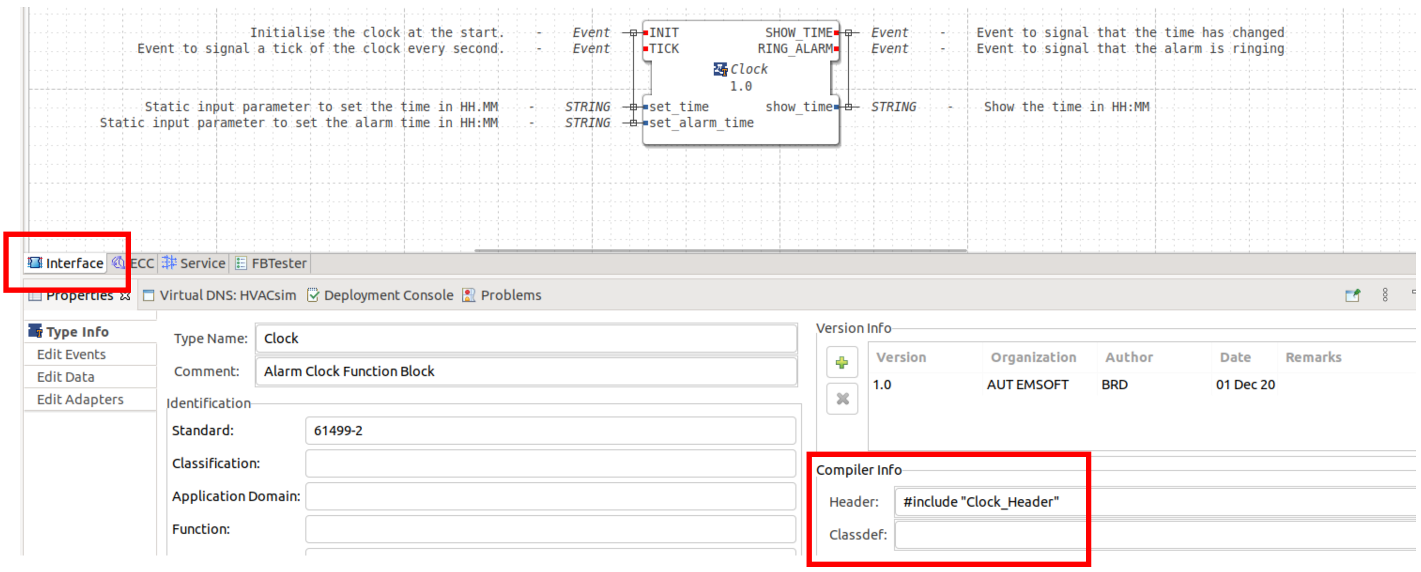


Updates the time on each clock tick

```
// This algorithmn is called each time the clock ticks. It updates
// the time, rolling it over when a minute or hour completes.
st_current_second() = st_current_second() + 1;
if (st_current_second() > 59) {
   st_current_second() = 0;
   st_current_minute() = st_current_minute() + 1;
   if (st_current_minute() > 59) {
     st_current_minute() = 0;
     st_current_hour() = st_current_hour() + 1;
     if (st_current_hour() > 12) {
      st_current_hour() = 1;
     }
   }
}
cout << "alg_TICK() " << st_current_hour() << ":" << st_current_minute() << ":" << st_current_second() << "\n";
```

These algorithms are written in C++. Set the **Language** box to be **AnyText** so that 4diac knows you are using C++ code and not IEC 61499 Structured Text.

If you need to include other C++ libraries, specify the library name in the box **Header** in the **Interface** tab.



```
    Initialise the clock at the start.  -  Event  INIT          SHOW_TIME  Event  -  Event to signal that the time has changed
Event to signal a tick of the clock every second.  -  Event  TICK     RING_ALARM  Event  -  Event to signal that the alarm is ringing
                                                        Clock
                                                         1.0
    Static input parameter to set the time in HH.MM  -  STRING  set_time   show_time  STRING  -  Show the time in HH:MM
Static input parameter to set the alarm time in HH:MM  -  STRING  set_alarm_time
```

Type Name: Clock

Comment: Alarm Clock Function Block

Identification

Standard: 61499-2

Classification:

Application Domain:

Function:

Version Info

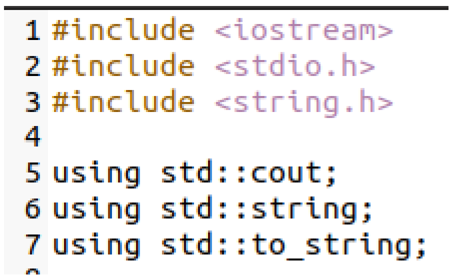
Version	Organization	Author	Date	Remarks
1.0	AUT EMSOFT	BRD	01 Dec 20	

Compiler Info

Header: #include "Clock_Header"

Classdef:

The **Clock_Header** file contains these **include** statements and **using** declarations.

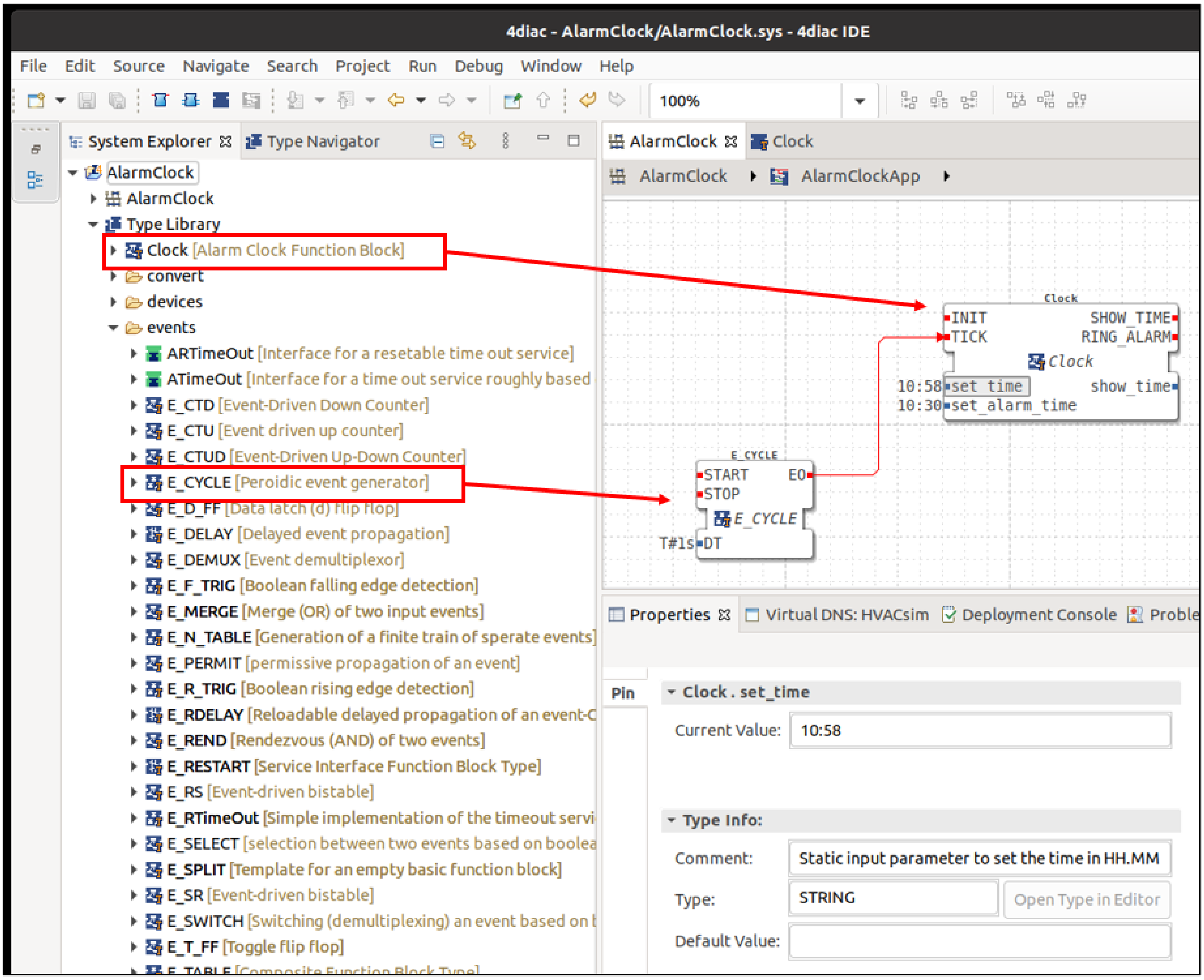


```
#include <iostream>
#include <stdio.h>
#include <string.h>

using std::cout;
using std::string;
using std::to_string;
```

Wiring the application together with function blocks.

A function block application is created by dragging the function blocks you created onto the design surface. The E_CYCLE block is a standard 4diac library block. It outputs an event at the frequency specified on its DT data input. The setting T#1s causes the E0 event to trigger once every second.



AlarmClock
- AlarmClock
- Type Library
  - Clock [Alarm Clock Function Block]
  - convert
  - devices
  - events
    - ARTimeOut [Interface for a resetable time out service]
    - ATimeOut [Interface for a time out service roughly based
    - E_CTD [Event-Driven Down Counter]
    - E_CTU [Event driven up counter]
    - E_CTUD [Event-Driven Up-Down Counter]
    - E_CYCLE [Peroidic event generator]
    - E_D_FF [Data latch (d) flip flop]
    - E_DELAY [Delayed event propagation]
    - E_DEMUX [Event demultiplexor]
    - E_F_TRIG [Boolean falling edge detection]
    - E_MERGE [Merge (OR) of two input events]
    - E_N_TABLE [Generation of a finite train of sperate events]
    - E_PERMIT [permissive propagation of an event]
    - E_R_TRIG [Boolean rising edge detection]
    - E_RDELAY [Reloadable delayed propagation of an event-C
    - E_REND [Rendezvous (AND) of two events]
    - E_RESTART [Service Interface Function Block Type]
    - E_RS [Event-driven bistable]
    - E_RTimeOut [Simple implementation of the timeout servi
    - E_SELECT [selection between two events based on boolea
    - E_SPLIT [Template for an empty basic function block]
    - E_SR [Event-driven bistable]
    - E_SWITCH [Switching (demultiplexing) an event based on t
    - E_T_FF [Toggle flip flop]
    - E_TABLE [Composite Function Block Type]

Clock . set_time

Current Value: 10:58

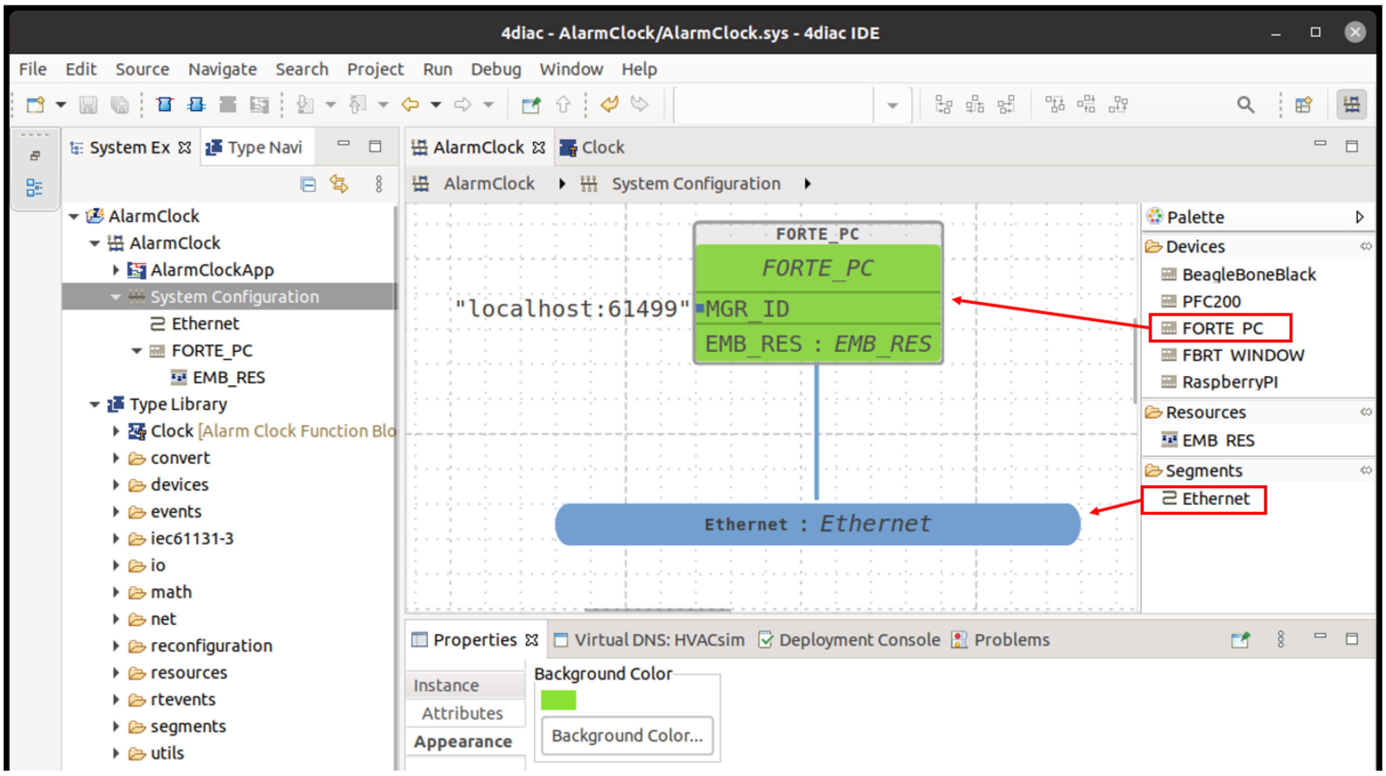
Type Info:

Comment:	Static input parameter to set the time in HH.MM
Type:	STRING — Open Type in Editor
Default Value:	

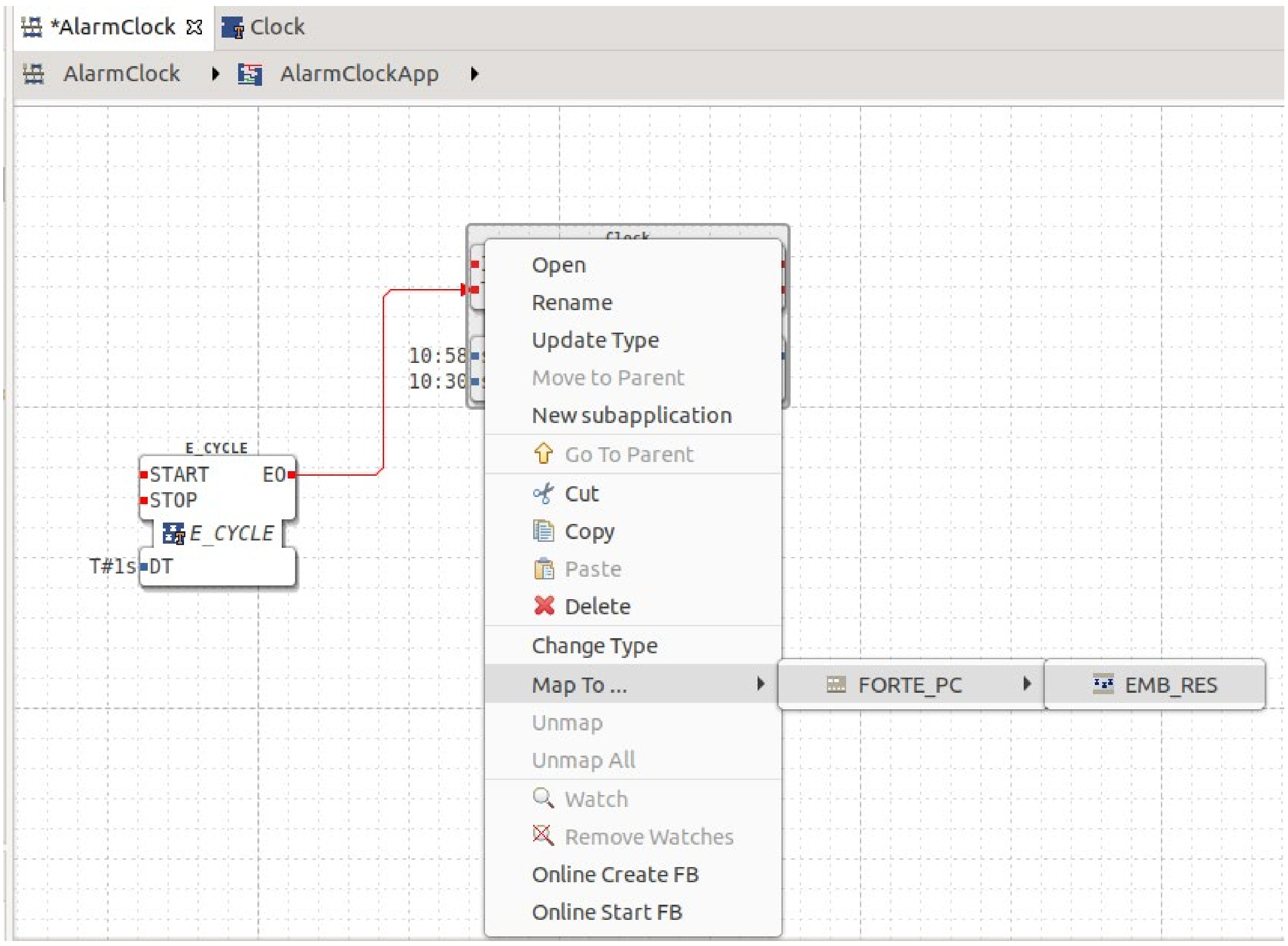
Mapping the function block application to an embedded system

Create and embedded controller to run the function block application on. To run it within Linux, drag a **FORTE PC** controller to the **System Configuration** design surface. You can change its colour in the Properties window.

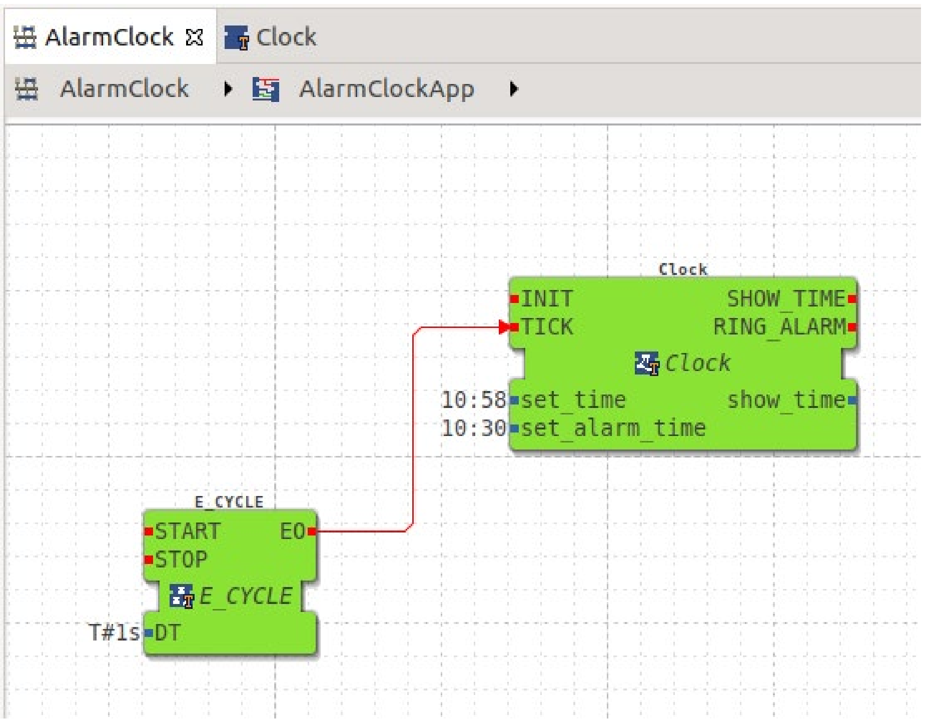
You need to add some Ethernet resources to give your embedded controller network capabilities. Drag an **Ethernet** component to the System Configuration. Then connect them by clicking on the Ethernet component and dragging a connection to the FORTE PC.



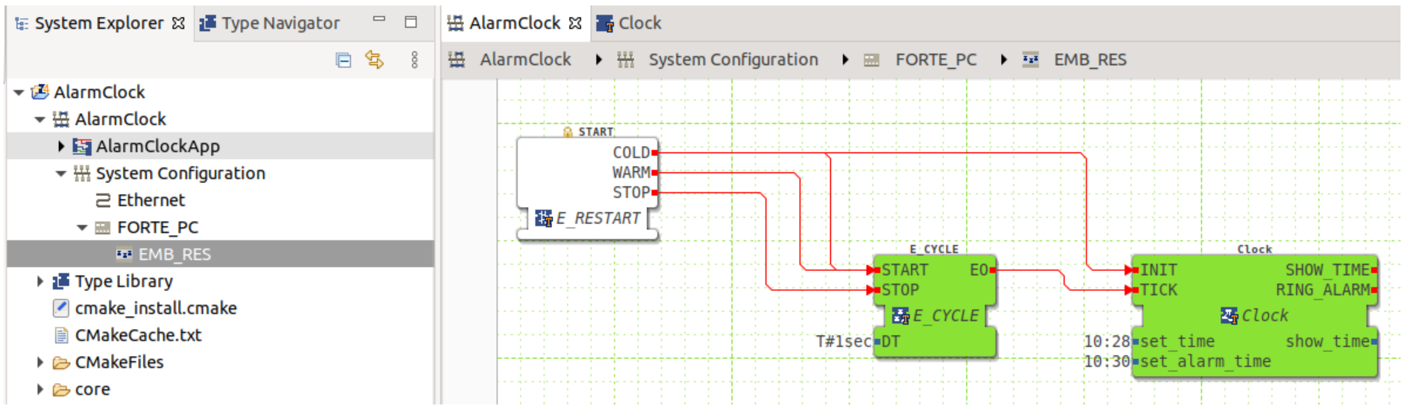
Then map each function block in the AlarmClock app to the FORTE PC EMB_RES



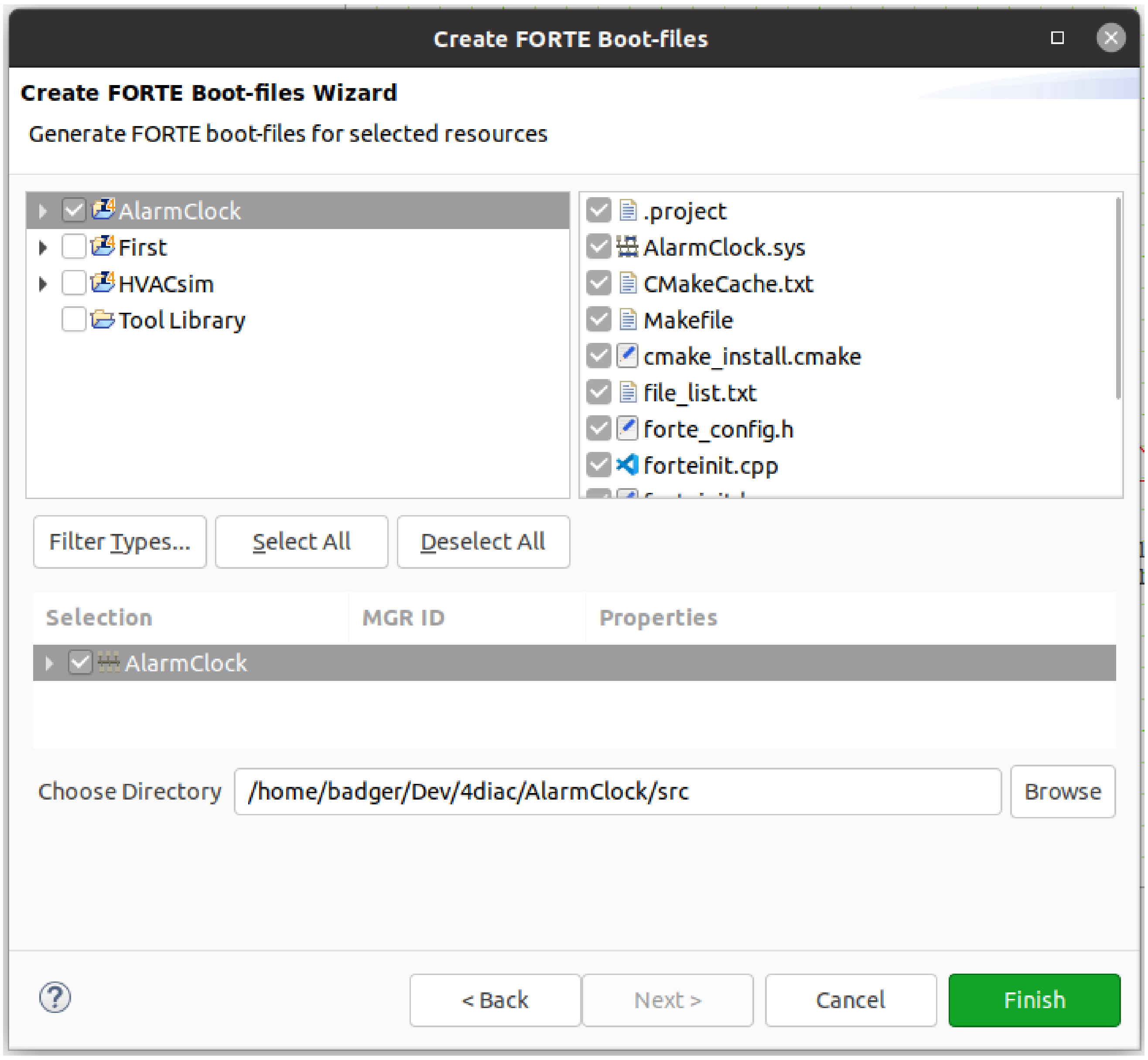
The app is now just about ready to be deployed and run:



Wire the Start function block into the Embedded Resource



Create the FORTE forte.fboot configuration file



Create FORTE Boot-files

Create FORTE Boot-files Wizard

Generate FORTE boot-files for selected resources

- [✓] AlarmClock
- [ ] First
- [ ] HVACsim
- [ ] Tool Library

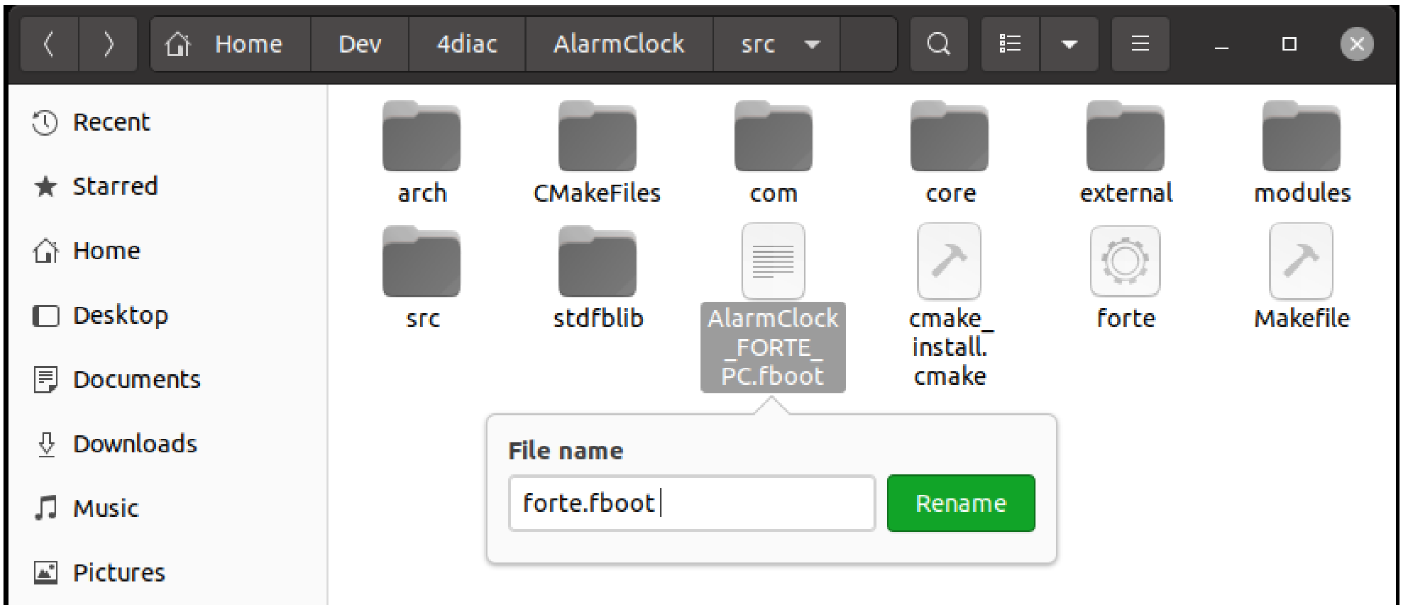
- [✓] .project
- [✓] AlarmClock.sys
- [✓] CMakeCache.txt
- [✓] Makefile
- [✓] cmake_install.cmake
- [✓] file_list.txt
- [✓] forte_config.h
- [✓] forteinit.cpp

Filter Types... | Select All | Deselect All

Selection	MGR ID	Properties
[✓] AlarmClock		

Choose Directory /home/badger/Dev/4diac/AlarmClock/src Browse

< Back | Next > | Cancel | Finish

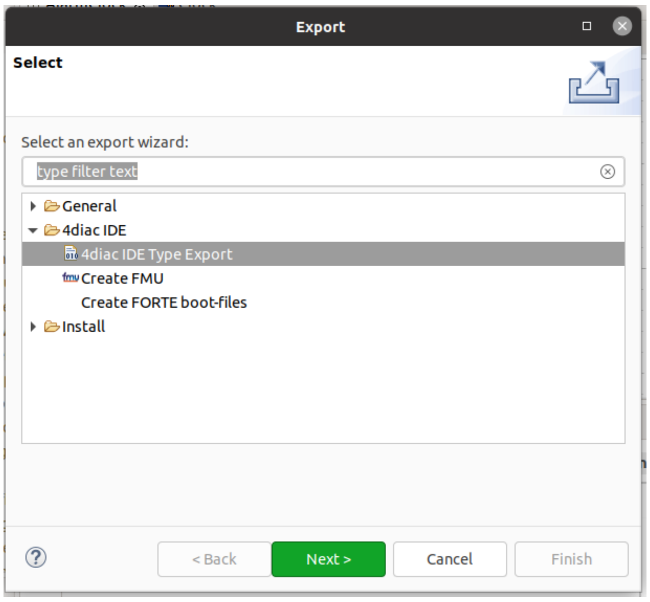

Home
Dev
4diac
AlarmClock
src
Recent
Starred
Home
Desktop
Documents
Downloads
Music
Pictures
arch
CMakeFiles
com
core
external
modules
src
stdfblib
AlarmClock_FORTE_PC.fboot
cmake_install.cmake
forte
Makefile
File name
forte.fboot
Rename

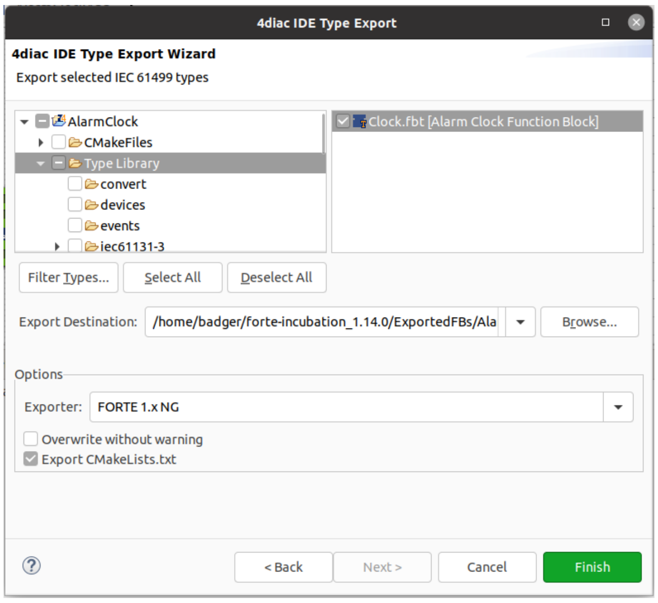
Step 5 – Deploying the completed Alarm Clock function block application

When all the function blocks have been designed and wired together, the completed application can be deployed and run. Deployment includes these tasks:

Exporting your function block types

The **File | Export** menu starts the 4diac Export wizard to convert the function blocks you have written into C++ classes. These will be compiled and linked into the FORTE runtime in a later step.





☑ Clock.fbt [Alarm Clock Function Block]

Export Destination: /home/badger/forte-incubation_1.14.0/ExportedFBs/Ala

Exporter: FORTE 1.x NG

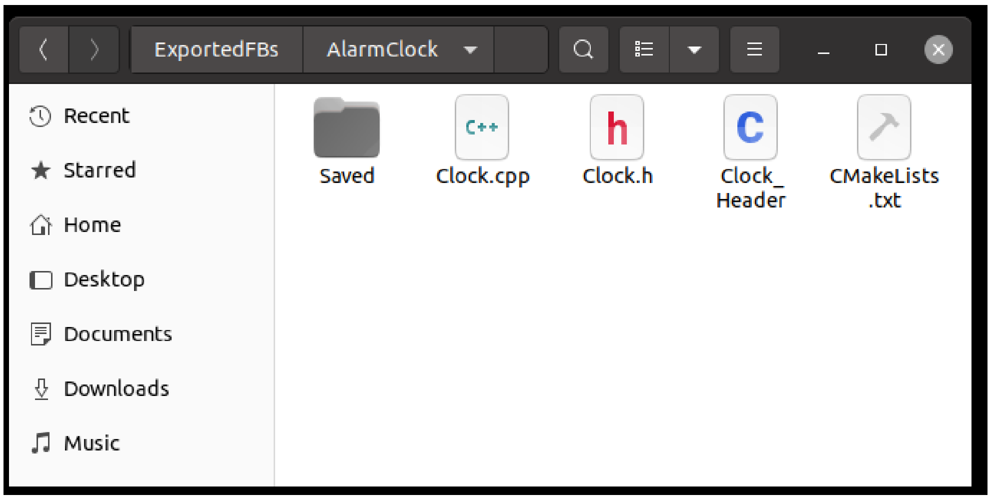
☐ Overwrite without warning

☑ Export CMakeLists.txt

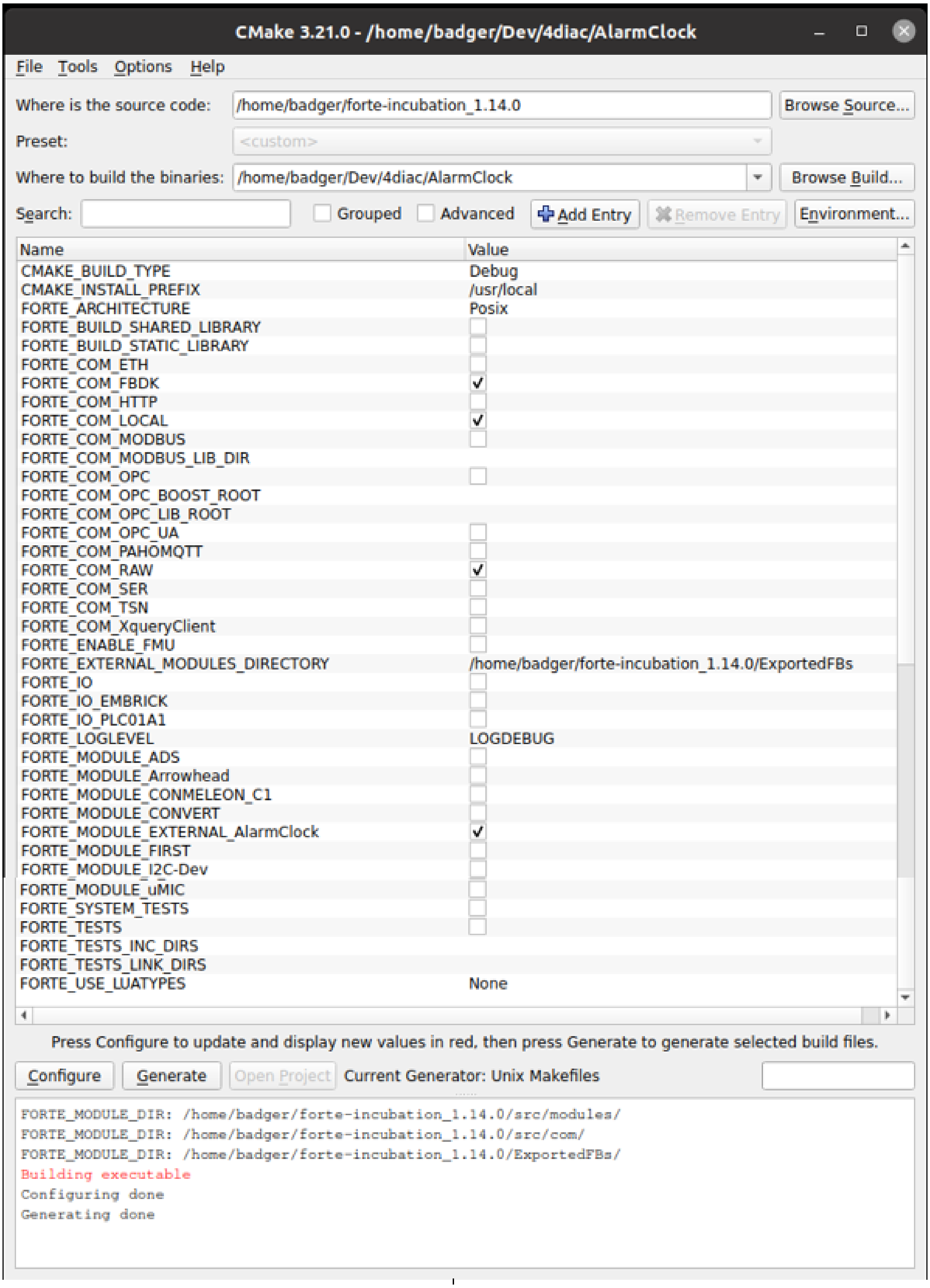
When selecting the function blocks to export, be careful not to select **all** the blocks in the Type Library by accidentally ticking the box. If you do, you will fill the export folder with many, many unnecessary blocks.

The GCC compiler and linker need to find your function blocks. It looks for custom function blocks in folders below the FORTE source code folder **/home/**<your ID>**/ forte-incubation_1.14.0/ExportedFBs.** Create a folder for your custom function blocks called **AlarmClock** to match your project name. Then select the **Export Destination** using the **Browse** button.

Tick the **Export CMakeLists.txt** check box before you click **Finish.** This creates a list of your function blocks for the GCC compiler and linker to process. The AlarmClock folder will contain the C++ source files after the export completes:



Creating the C++ Makefile using CMake-gui



Where is the source code: /home/badger/forte-incubation_1.14.0

Preset: <custom>

Where to build the binaries: /home/badger/Dev/4diac/AlarmClock

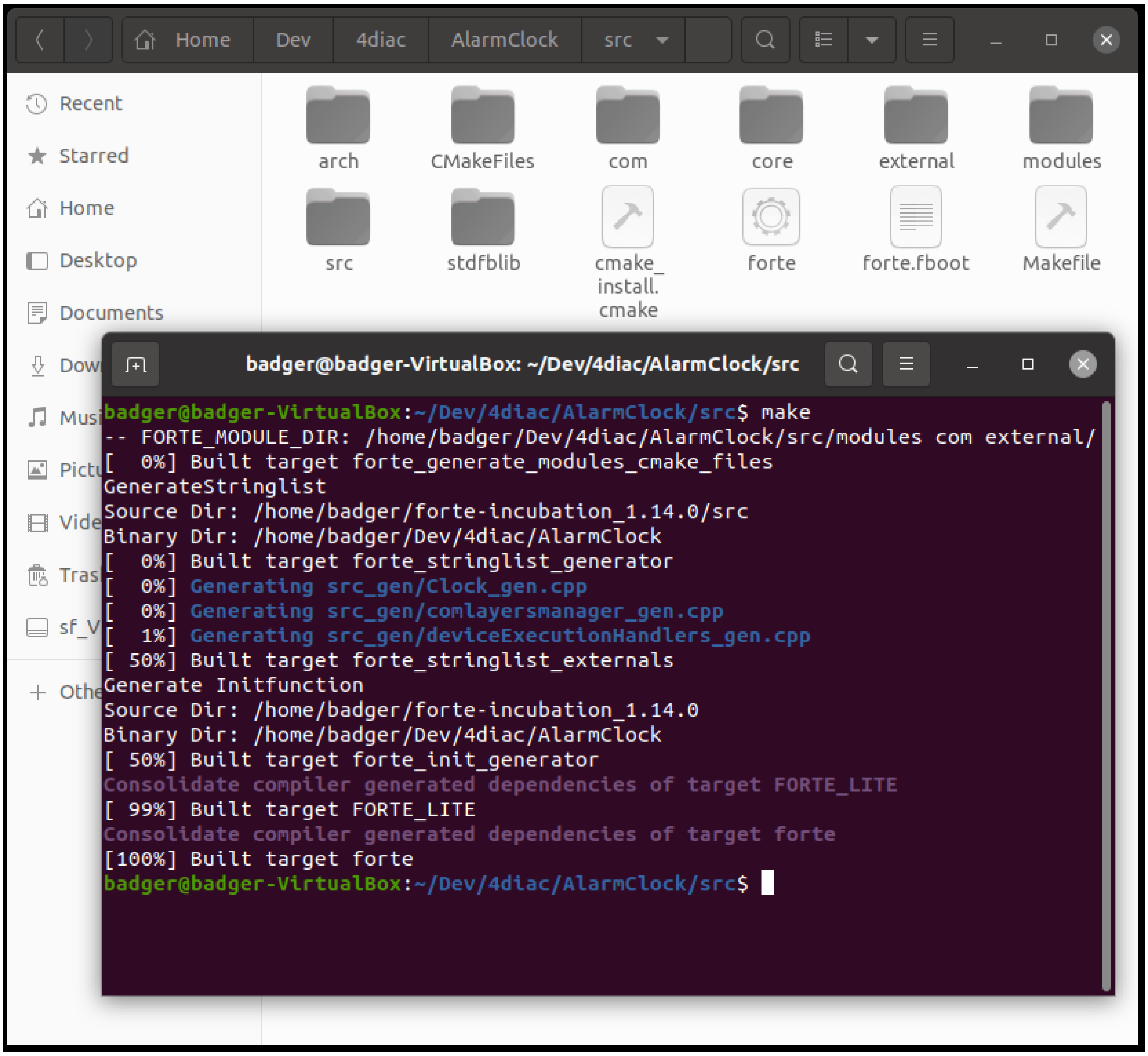
Search:

Name	Value
CMAKE_BUILD_TYPE	Debug
CMAKE_INSTALL_PREFIX	/usr/local
FORTE_ARCHITECTURE	Posix
FORTE_BUILD_SHARED_LIBRARY	☐
FORTE_BUILD_STATIC_LIBRARY	☐
FORTE_COM_ETH	☐
FORTE_COM_FBDK	✓
FORTE_COM_HTTP	☐
FORTE_COM_LOCAL	✓
FORTE_COM_MODBUS	☐
FORTE_COM_MODBUS_LIB_DIR	
FORTE_COM_OPC	☐
FORTE_COM_OPC_BOOST_ROOT	
FORTE_COM_OPC_LIB_ROOT	
FORTE_COM_OPC_UA	☐
FORTE_COM_PAHOMQTT	☐
FORTE_COM_RAW	✓
FORTE_COM_SER	☐
FORTE_COM_TSN	☐
FORTE_COM_XqueryClient	☐
FORTE_ENABLE_FMU	☐
FORTE_EXTERNAL_MODULES_DIRECTORY	/home/badger/forte-incubation_1.14.0/ExportedFBs
FORTE_IO	☐
FORTE_IO_EMBRICK	☐
FORTE_IO_PLC01A1	☐
FORTE_LOGLEVEL	LOGDEBUG
FORTE_MODULE_ADS	☐
FORTE_MODULE_Arrowhead	☐
FORTE_MODULE_CONMELEON_C1	☐
FORTE_MODULE_CONVERT	☐
FORTE_MODULE_EXTERNAL_AlarmClock	✓
FORTE_MODULE_FIRST	☐
FORTE_MODULE_I2C-Dev	☐
FORTE_MODULE_uMIC	☐
FORTE_SYSTEM_TESTS	☐
FORTE_TESTS	☐
FORTE_TESTS_INC_DIRS	
FORTE_TESTS_LINK_DIRS	
FORTE_USE_LUATYPES	None

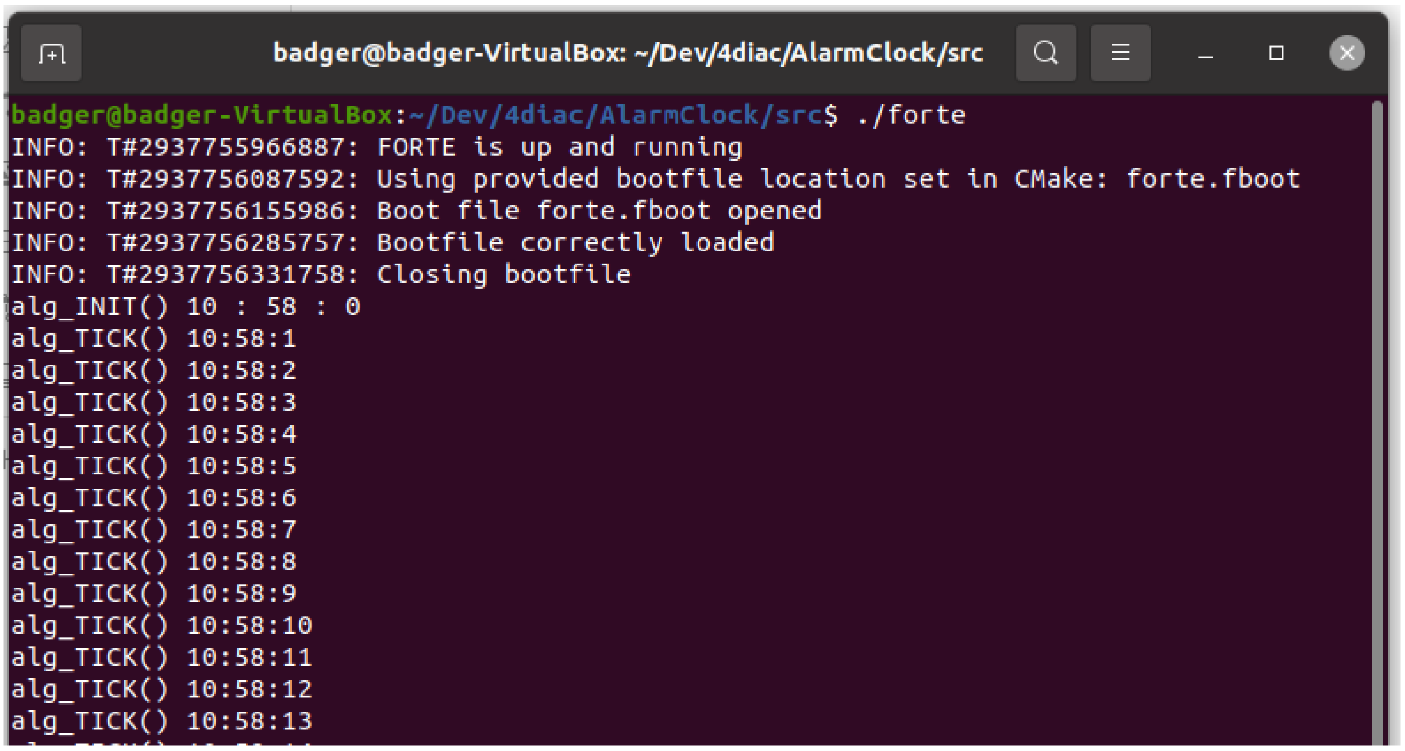
Press Configure to update and display new values in red, then press Generate to generate selected build files.

Current Generator: Unix Makefiles

```
FORTE_MODULE_DIR: /home/badger/forte-incubation_1.14.0/src/modules/
FORTE_MODULE_DIR: /home/badger/forte-incubation_1.14.0/src/com/
FORTE_MODULE_DIR: /home/badger/forte-incubation_1.14.0/ExportedFBs/
Building executable
Configuring done
Generating done
```



```
badger@badger-VirtualBox:~/Dev/4diac/AlarmClock/src$ make
-- FORTE_MODULE_DIR: /home/badger/Dev/4diac/AlarmClock/src/modules com external/
[  0%] Built target forte_generate_modules_cmake_files
GenerateStringlist
Source Dir: /home/badger/forte-incubation_1.14.0/src
Binary Dir: /home/badger/Dev/4diac/AlarmClock
[  0%] Built target forte_stringlist_generator
[  0%] Generating src_gen/Clock_gen.cpp
[  0%] Generating src_gen/comlayersmanager_gen.cpp
[  1%] Generating src_gen/deviceExecutionHandlers_gen.cpp
[ 50%] Built target forte_stringlist_externals
Generate Initfunction
Source Dir: /home/badger/forte-incubation_1.14.0
Binary Dir: /home/badger/Dev/4diac/AlarmClock
[ 50%] Built target forte_init_generator
Consolidate compiler generated dependencies of target FORTE_LITE
[ 99%] Built target FORTE_LITE
Consolidate compiler generated dependencies of target forte
[100%] Built target forte
badger@badger-VirtualBox:~/Dev/4diac/AlarmClock/src$ 
```


badger@badger-VirtualBox: ~/Dev/4diac/AlarmClock/src

```
badger@badger-VirtualBox:~/Dev/4diac/AlarmClock/src$ ./forte
INFO: T#2937755966887: FORTE is up and running
INFO: T#2937756087592: Using provided bootfile location set in CMake: forte.fboot
INFO: T#2937756155986: Boot file forte.fboot opened
INFO: T#2937756285757: Bootfile correctly loaded
INFO: T#2937756331758: Closing bootfile
alg_INIT() 10 : 58 : 0
alg_TICK() 10:58:1
alg_TICK() 10:58:2
alg_TICK() 10:58:3
alg_TICK() 10:58:4
alg_TICK() 10:58:5
alg_TICK() 10:58:6
alg_TICK() 10:58:7
alg_TICK() 10:58:8
alg_TICK() 10:58:9
alg_TICK() 10:58:10
alg_TICK() 10:58:11
alg_TICK() 10:58:12
alg_TICK() 10:58:13
```