

Creating Embedded Systems with Cadmium

This guide will explain how to create an embedded real time system with Cadmium from scratch. To start, this guide will explain how to install the dependencies for running the Cadmium simulator using the MSP432 microcontroller. The guide will then explain how to download a sample github repository, and show the format of one of the example projects. This guide also shows how to run an example as both an embedded execution and simulation.

Installing the Cadmium Simulator and its Dependencies

Use the manual linked below to download and install Cadmium, and its appropriate dependencies. This guide will not use the manual's instructions to run an example, so the sections called "Compiling and Running a Cadmium DEVS Model" should be ignored. Use the instructions on pages 4 - 14 to download and install Cadmium if your operating system is **Windows**, pages 17 - 22 if it's **Ubuntu or Linux**.

Manual Link: <https://www.sce.carleton.ca/courses/sysc-5104/lib/exe/fetch.php?media=cadmiumusermanual.pdf>

Installing the Embedded Compiler (Windows)

Next, we will download the embedded compiler (GNU Arm Embedded Toolchain). For **Windows** installation, enter the link: <https://developer.arm.com/downloads/-/gnu-rm> Within the link, scroll down and select the windows executable to begin the download shown in Figure 1.

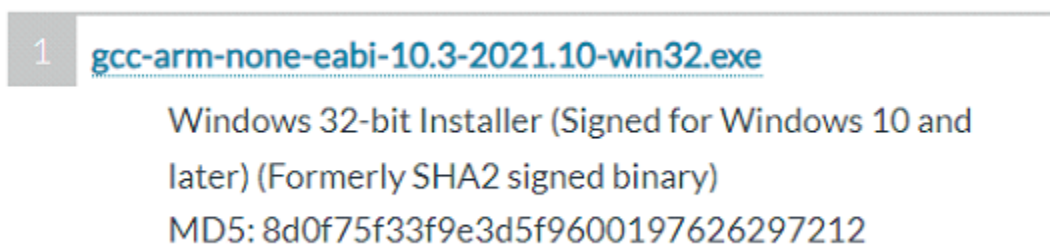


Figure 1. Windows Executable Download

Once downloaded, open the executable file. You will be greeted with several install settings windows, click agree and next with all the default settings as shown in Figure 2.

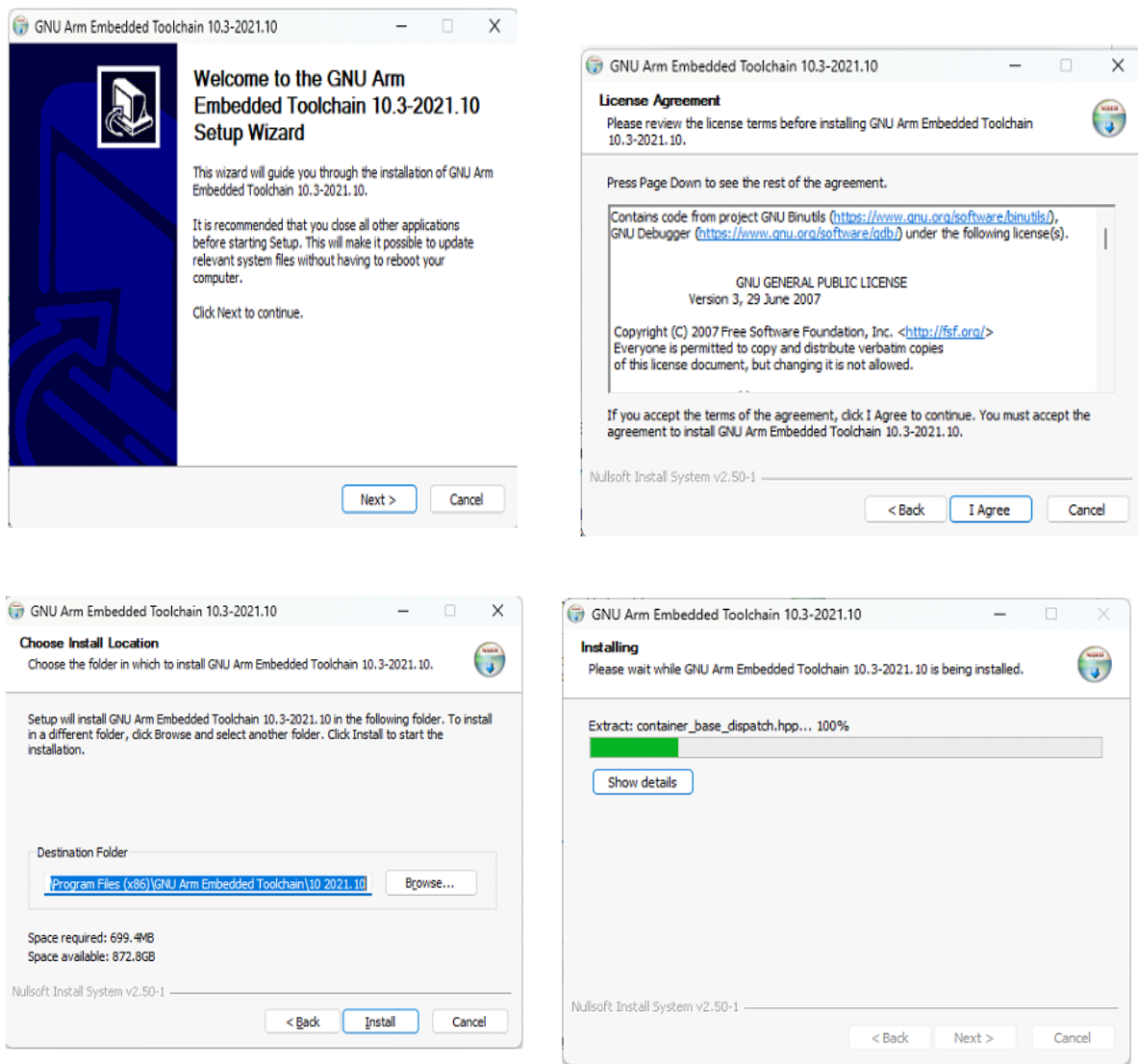


Figure 2. GNU Arm Installation Windows

In the 'Completing the GNU Arm Embedded' window, deselect 'Show Readme' and click 'Finish'. A terminal will open, which can be closed, shown in Figure 3.

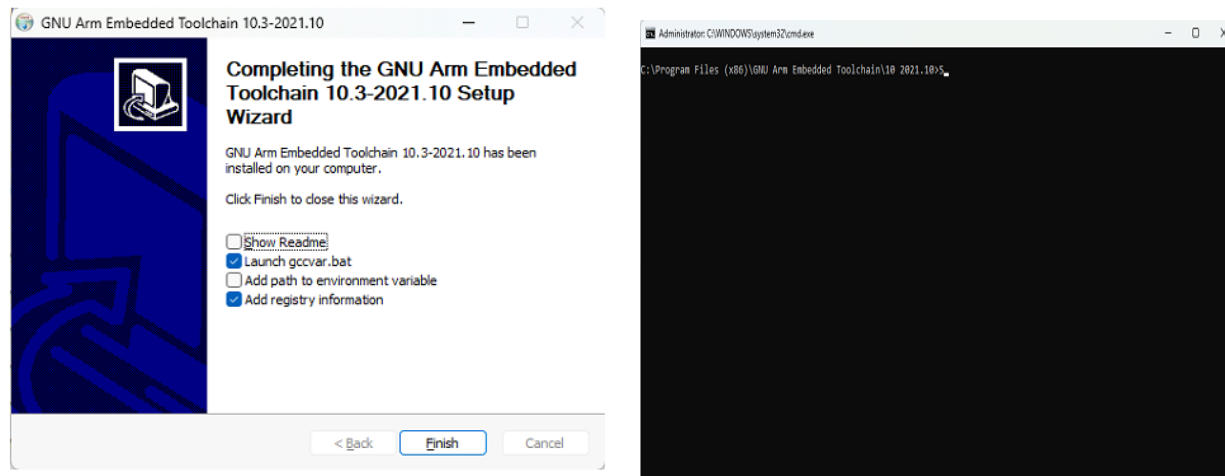


Figure 3. Completing Windows

Next, search in the Windows menu for 'Environment Variables', and open 'Edit the system environment variables' in the control panel. A window will open titled 'System Properties', and it will be on the Advanced tab. Click on the 'Environment Variables...' button, another window will open titled 'Environment Variables'. Under 'System Variables' find and double click on the variable called 'Path'. Select 'New', enter the following into the text box:

C:\Program Files(x86)\GNU Arm Embedded Toolchain\10 2021.10\bin

This environment variable should be now listed in the dialog box. See Figure 4 for all the windows that should be viewed during this segment. Installation of the embedded compiler is now complete.

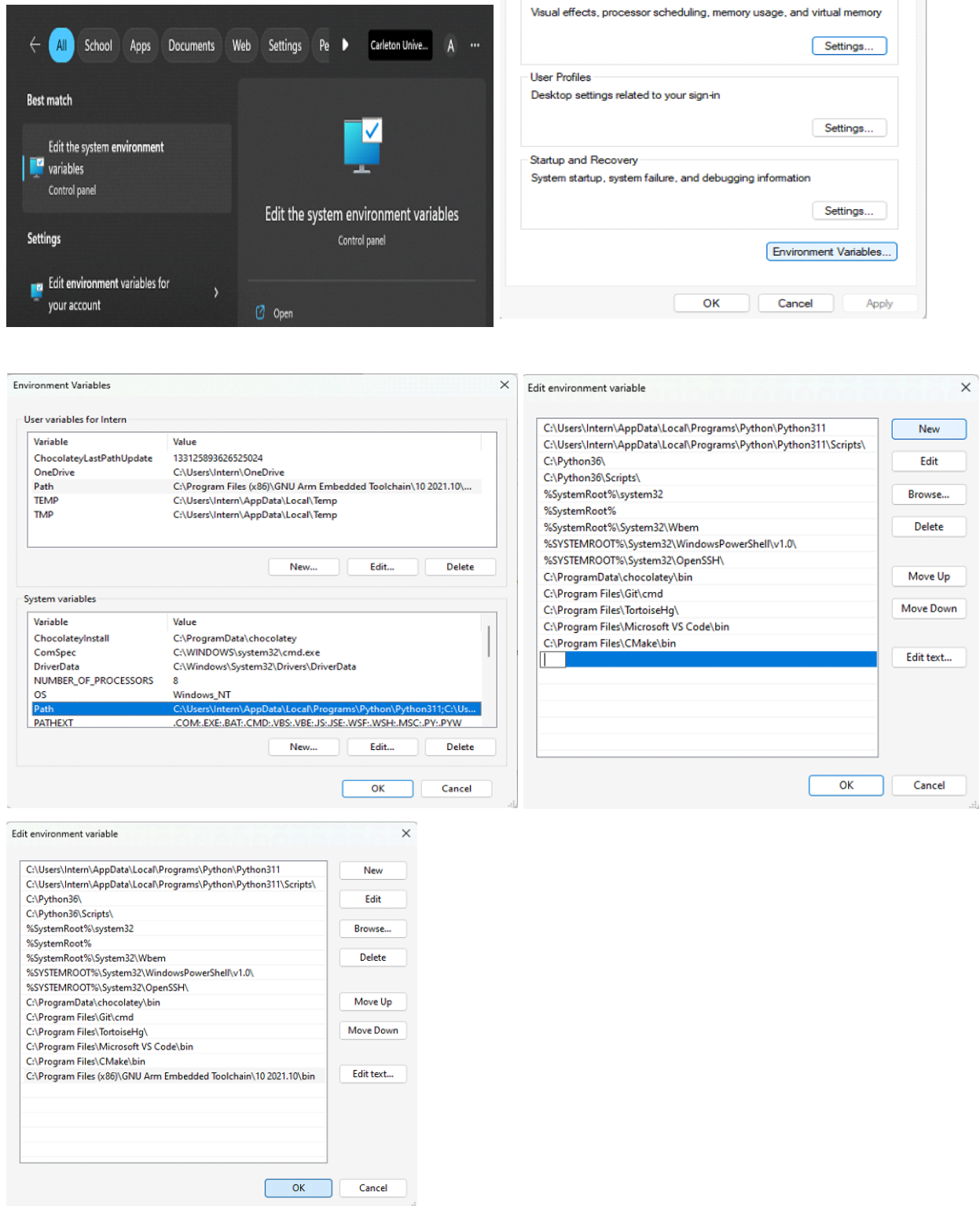


Figure 4. Environment Variable Windows

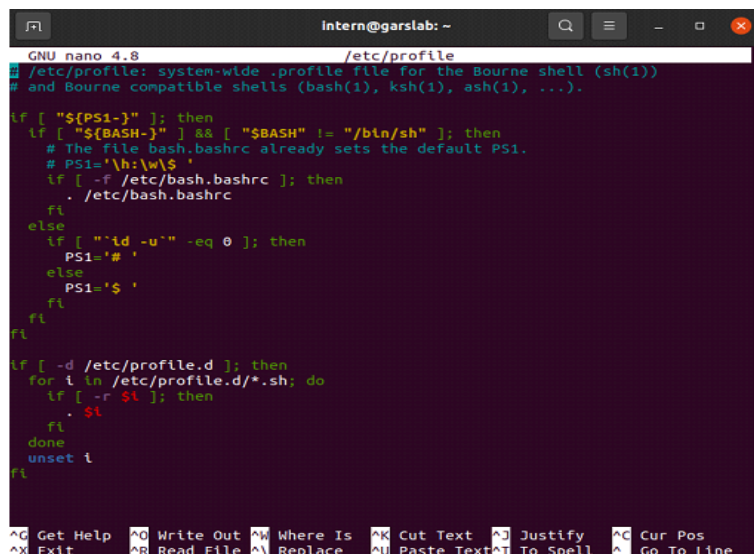
Installing the Embedded Compiler (Ubuntu / Linux)

For Ubuntu / Linux installation, enter the link: <https://developer.arm.com/downloads/-/gnu-rm>
Scroll down and select the correct executable for your operating system to begin the download:

- 3 [gcc-arm-none-eabi-10.3-2021.10-x86_64-linux.tar.bz2](#)
Linux x86_64 Tarball
MD5: 2383e4eb4ea23f248d33adc70dc3227e
- 4 [gcc-arm-none-eabi-10.3-2021.10-aarch64-linux.tar.bz2](#)
Linux AArch64 Tarball
MD5: 3fe3d8bb693bd0a6e4615b6569443d0d

Figure 5. Ubuntu / Linux Executable

Next, extract the downloaded folder to your desired location. Open a terminal, by pressing CTRL + ALT + T. To add a permanent directory to the PATH environment variable we need to edit the /etc/profile/ file. Enter the following command into the terminal: 'sudo nano /etc/profile'. The following screen will appear in the terminal:



```
GNU nano 4.8 /etc/profile
# /etc/profile: system-wide .profile file for the Bourne shell (sh(1))
# and Bourne compatible shells (bash(1), ksh(1), ash(1), ...).

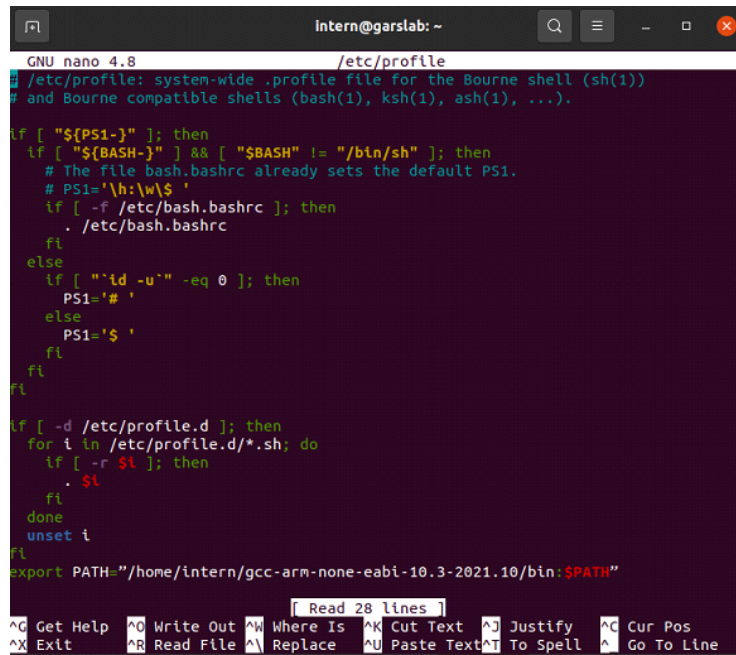
if [ "${PS1-}" ]; then
  if [ "${BASH-}" ] && [ "$BASH" != "/bin/sh" ]; then
    # The file bash.bashrc already sets the default PS1.
    # PS1='\h:\w\$ '
    if [ -f /etc/bash.bashrc ]; then
      . /etc/bash.bashrc
    fi
  else
    if [ "$(id -u)" -eq 0 ]; then
      PS1='# '
    else
      PS1='$ '
    fi
  fi
fi

if [ -d /etc/profile.d ]; then
  for i in /etc/profile.d/*.sh; do
    if [ -r $i ]; then
      . $i
    fi
  done
unset i
fi
```

Figure 6. Terminal View After Command

We will edit this file to add a directory. This can be done by adding the following line and exchanging the filepath with the correct one for your computer to gcc-arm-none-eabi-10.3-2021.10/bin.

'export PATH = "/home/intern/gcc-arm-none-eabi-10.3-2021.10/bin:\$PATH"



```
GNU nano 4.8 /etc/profile
# /etc/profile: system-wide .profile file for the Bourne shell (sh(1))
# and Bourne compatible shells (bash(1), ksh(1), ash(1), ...).

if [ "${PS1-}" ]; then
  if [ "${BASH-}" ] && [ "$BASH" != "/bin/sh" ]; then
    # The file bash.bashrc already sets the default PS1.
    # PS1='\h:\w\$ '
    if [ -f /etc/bash.bashrc ]; then
      . /etc/bash.bashrc
    fi
  else
    if [ "`id -u`" -eq 0 ]; then
      PS1='# '
    else
      PS1='$ '
    fi
  fi
fi

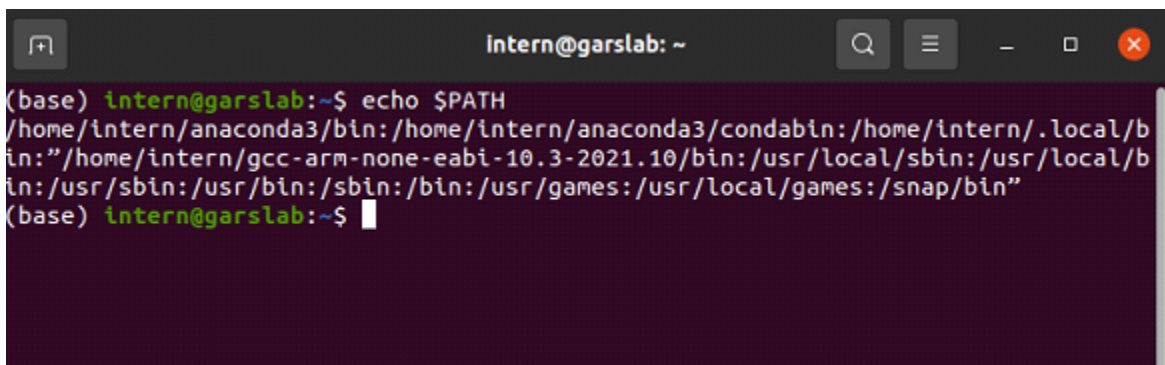
if [ -d /etc/profile.d ]; then
  for i in /etc/profile.d/*.sh; do
    if [ -r $i ]; then
      . $i
    fi
  done
unset i
fi

export PATH="/home/intern/gcc-arm-none-eabi-10.3-2021.10/bin:$PATH"
```

Figure 7. Terminal with export PATH

Save and exit from the file by entering 'CTRL + X', then 'Y', and then 'ENTER'.

Restarting the system is required for the changes to take effect. After restarting you can verify the changes were successful by entering 'echo \$PATH' and looking for the filepath you added.



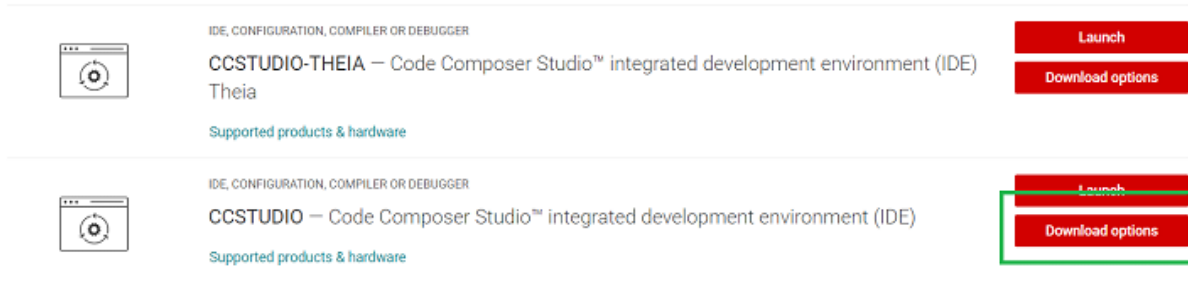
```
(base) intern@garslab:~$ echo $PATH
/home/intern/anaconda3/bin:/home/intern/anaconda3/condabin:/home/intern/.local/bin:/home/intern/gcc-arm-none-eabi-10.3-2021.10/bin:/usr/local/sbin:/usr/local/bin:/usr/sbin:/usr/bin:/sbin:/bin:/usr/games:/usr/local/games:/snap/bin
(base) intern@garslab:~$
```

Figure 8. Terminal with echo \$PATH

Installing the Code Composer IDE

Next, we will download an IDE that can be used to compile our program on an embedded system. Go to the following link: <https://www.ti.com/tool/CCSTUDIO> and select 'Download Options' for 'CCSTUDIO'.

Downloads



The screenshot shows the 'Downloads' section of the TI.com website. It lists two IDE options: 'CCSTUDIO-THEIA' and 'CCSTUDIO'. Both options include a 'Launch' button and a 'Download options' button. The 'Download options' button for 'CCSTUDIO' is highlighted with a green rectangular box.

Icon	Category	Product Name	Buttons
	IDE, CONFIGURATION, COMPILER OR DEBUGGER	CCSTUDIO-THEIA – Code Composer Studio™ integrated development environment (IDE) Theia	Launch, Download options
	IDE, CONFIGURATION, COMPILER OR DEBUGGER	CCSTUDIO – Code Composer Studio™ integrated development environment (IDE)	Launch, Download options (highlighted)

Figure 9. CCSTUDIO Download options

For Windows, select the 'Windows single file (offline) installer'. Then extract the folder to your desired location and run the ccs_setup executable file.

↓ [Windows single file \(offline\) installer for Code Composer Studio IDE \(all features, devices\) – 1227793 K](#)

Name	Type	Size
binary	File folder	
components	File folder	
features	File folder	
artifacts	Executable Jar File	1 KB
ccs_setup_12.5.0.00007	Application	38,688 KB
content	Executable Jar File	4 KB
README_FIRST_win64	Text Document	1 KB
timestamp	Text Document	1 KB

Figure 10. Windows Installer and Extracted Executable

For Ubuntu/Linux, select the 'Linux single file (offline) installer' below where the windows offline installer is located. Then extract the folder to your desired location and run the executable file: 'css_setup_12.4.00007.run'.

↓ [Linux single file \(offline\) installer for Code Composer Studio IDE \(all features, devices\) – 1209458 K](#)

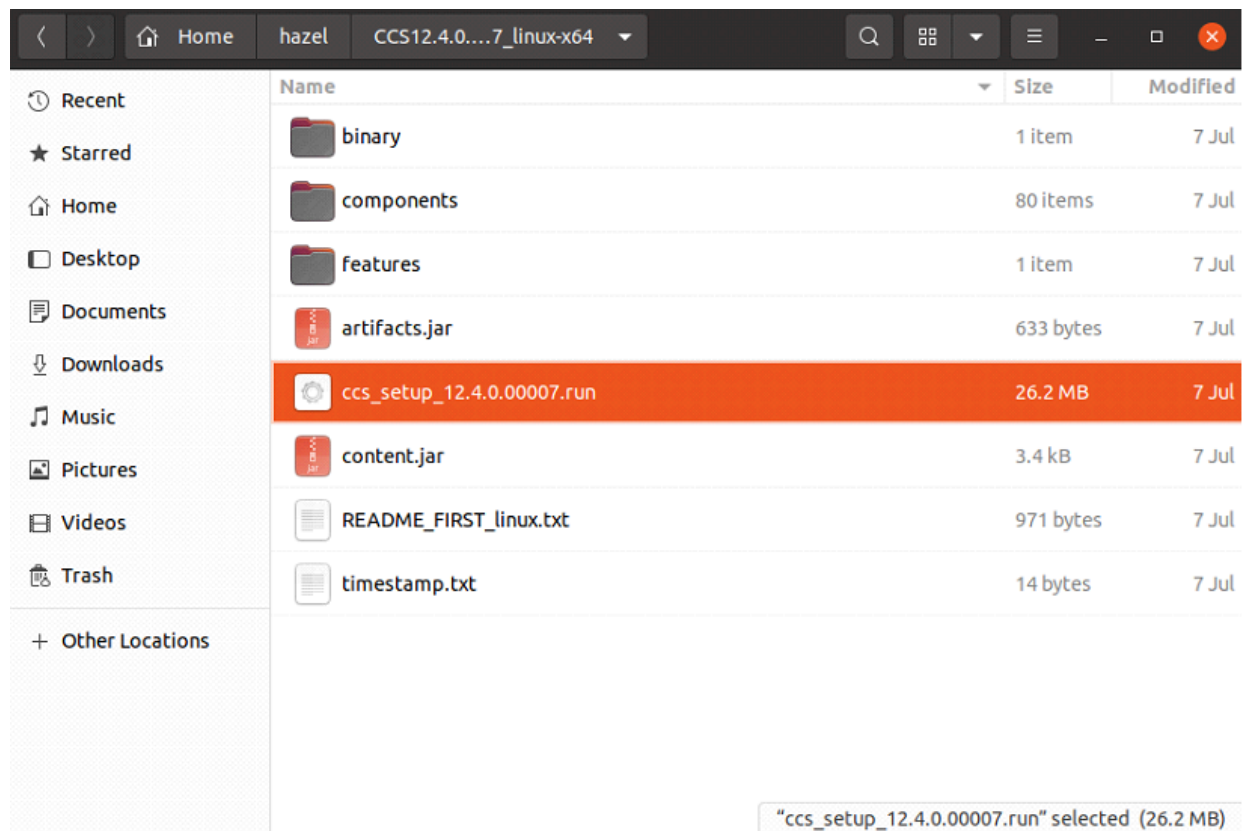


Figure 11. Ubuntu/Linux Installer and Run File

This step is exclusive for Ubuntu/Linux. Select 'Forward' when prompted. You will need to enter the installation directory.

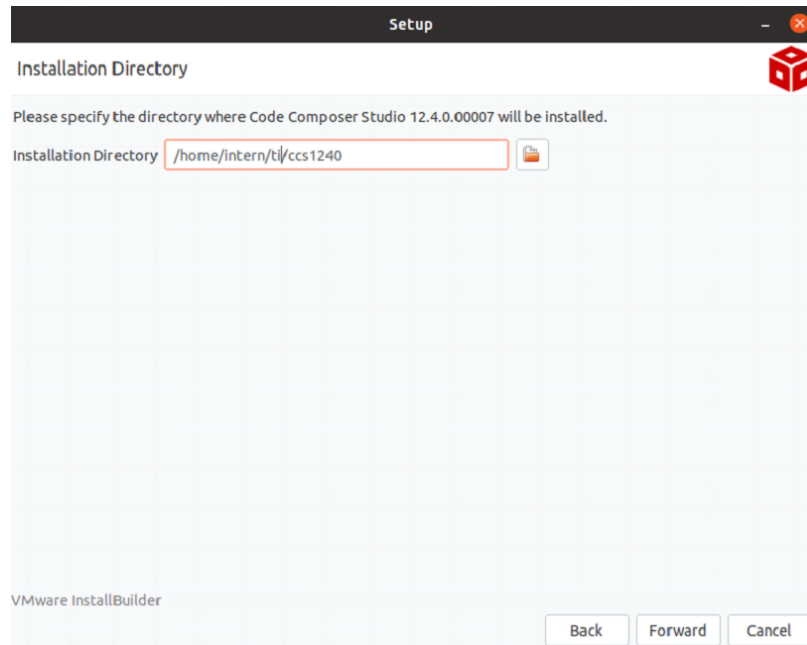


Figure 12. Installation Directory Window for Ubuntu/Linux

Continue the installation steps on the following pages for all operating systems.

A setup window will open, select 'I accept the agreement', followed by clicking the 'Next' button. Another setup window will open, reboot your PC if indicated, otherwise select the 'Next' button.

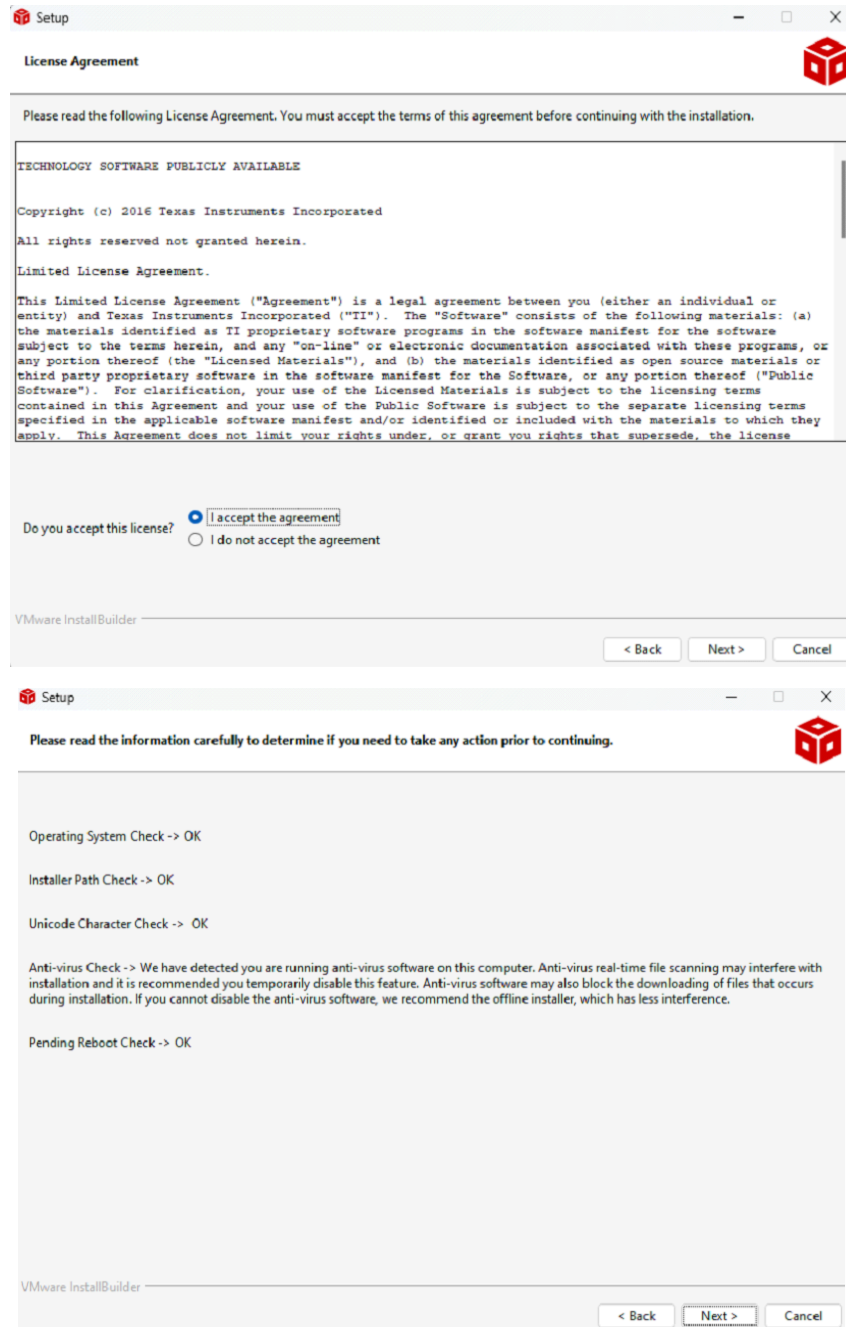


Figure 13. License Agreement and Setup Windows

Select 'Custom Installation' and 'Next'. Select the box for 'SimpleLink MSP432 low power + performance MCUs' then 'Next'.

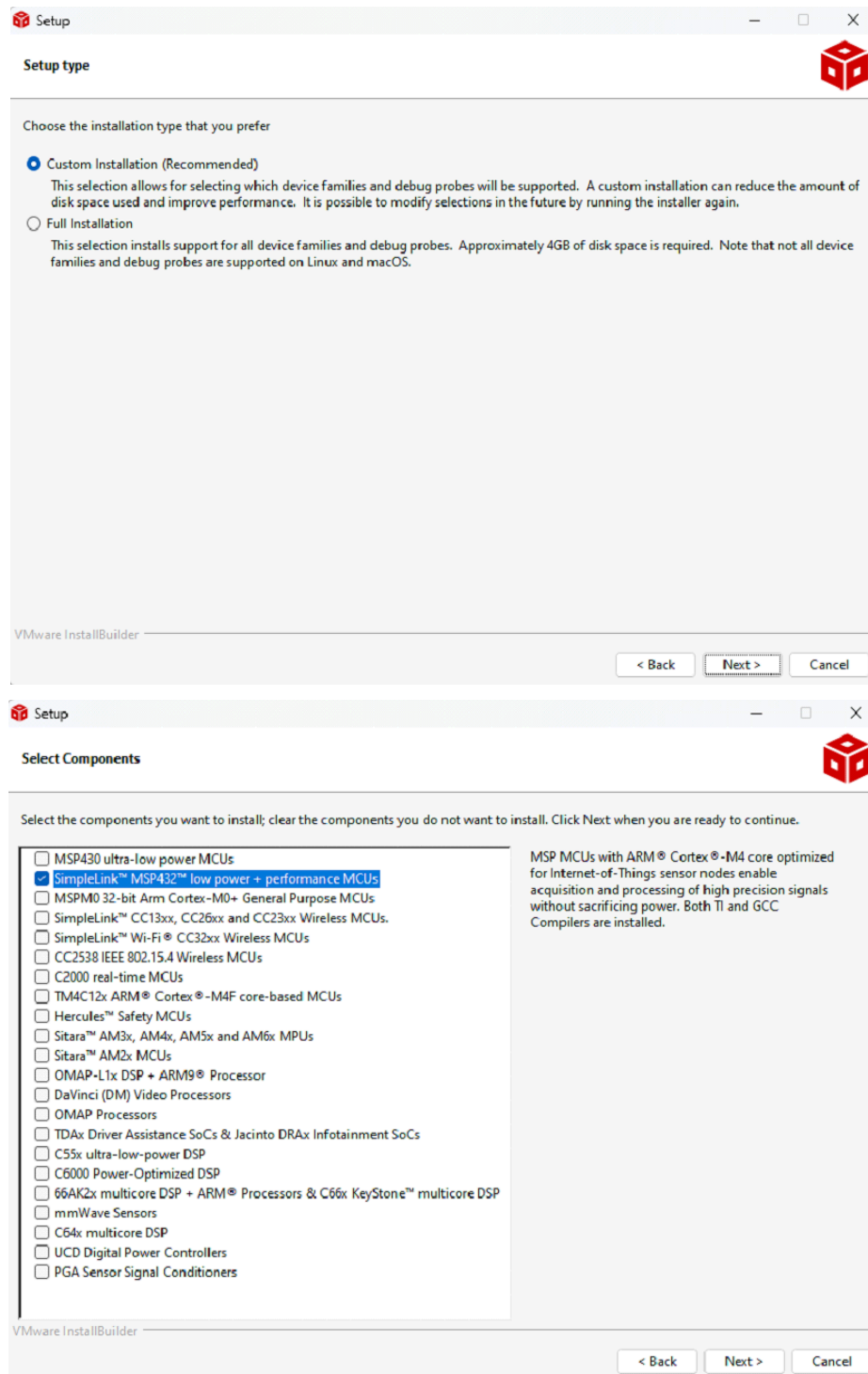


Figure 14. Custom Installation and Components Windows

When selecting which debug probes to install, ignore all the options and press 'Next'.
When presented with unsupported boards, click 'Next'.

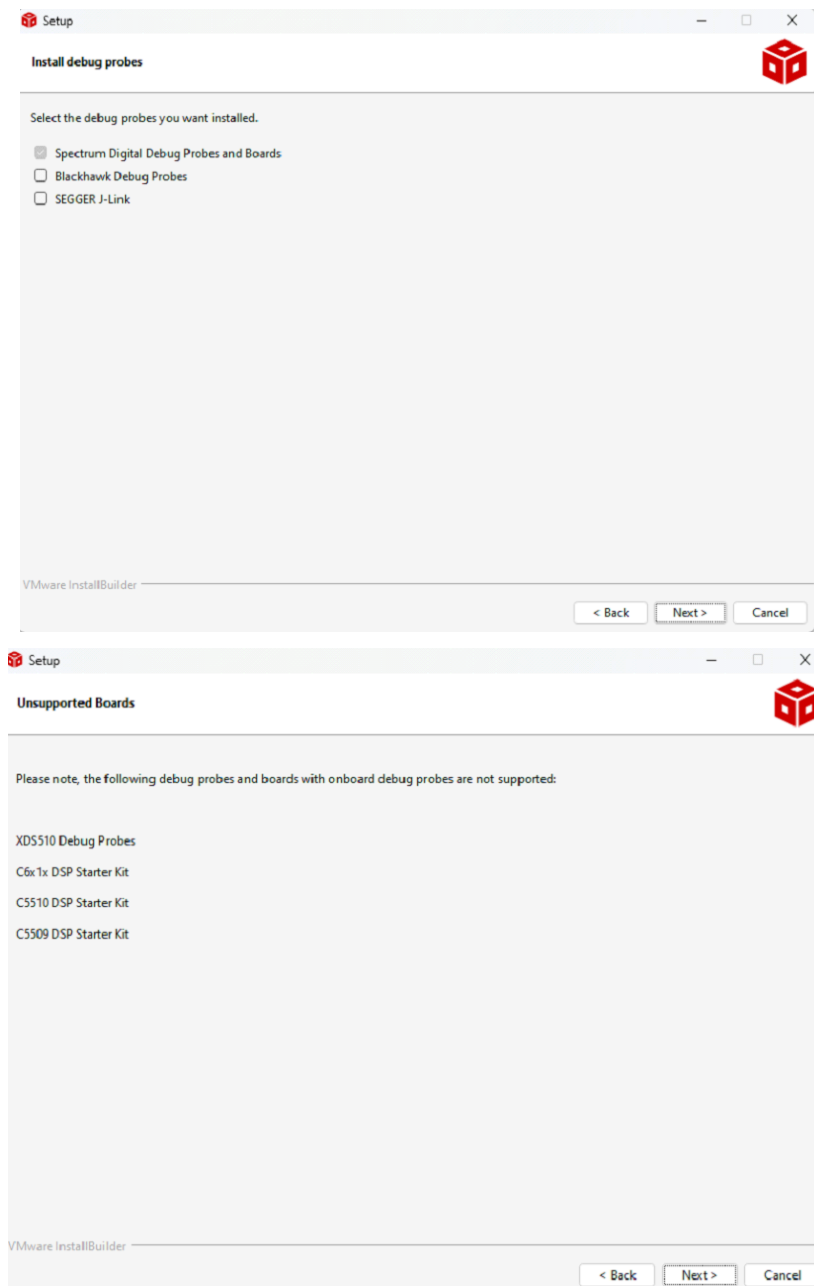


Figure 15. Debug Probes and Unsupported Boards Windows

Press 'Next' in the ready to install window to begin installation. Reboot your PC when the installer indicates to do so.

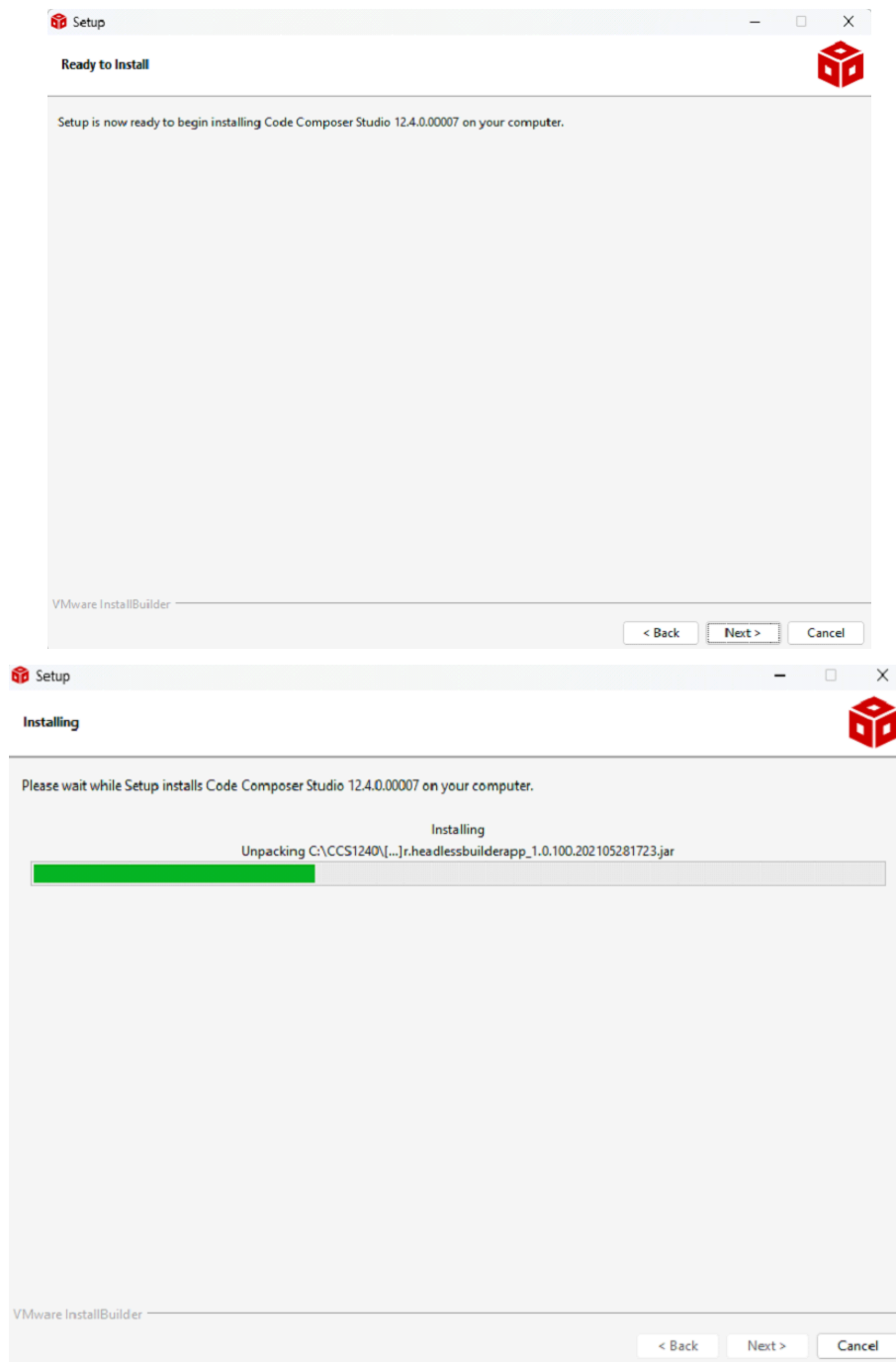


Figure 16. Installation Windows

After rebooting, we will now open Code Composer Studio (CCS). When opening CCS for the first time we are prompted with a select directory as workspace window. We can create a folder and specify it here to hold all of our projects, but this will be done later. Select 'Launch'. Also 'Allow access' if a firewall window opens.

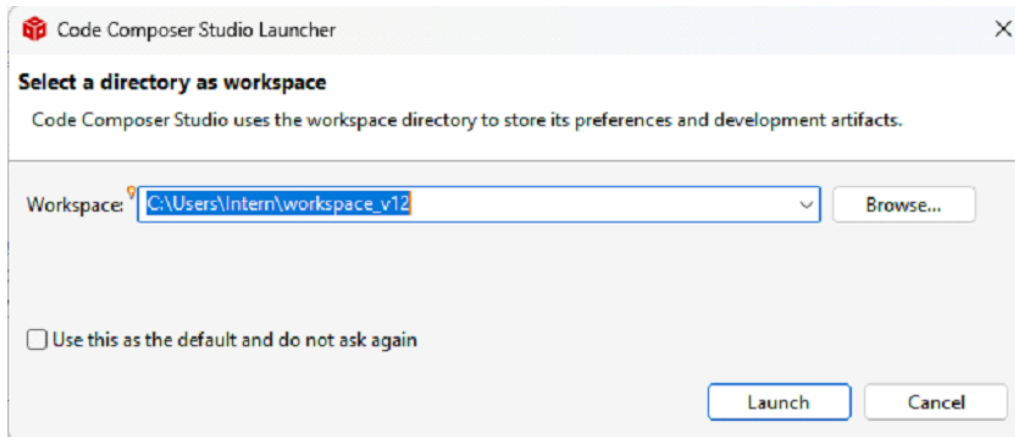


Figure 17. Select Workspace Window

Downloading Example Projects and Setting up the IDE Workspace

The example projects that will be used for this guide can be run as either embedded execution on the MSP342 microcontroller (using Code Composer Studio IDE), or as a simulation using the terminal. First we need to download the sample repository from the following link: https://github.com/SimulationEverywhere-Models/cadmium_v2_rt_msp432. Select the green '<> Code' button and select 'Download ZIP' from the bottom of the menu that appears.

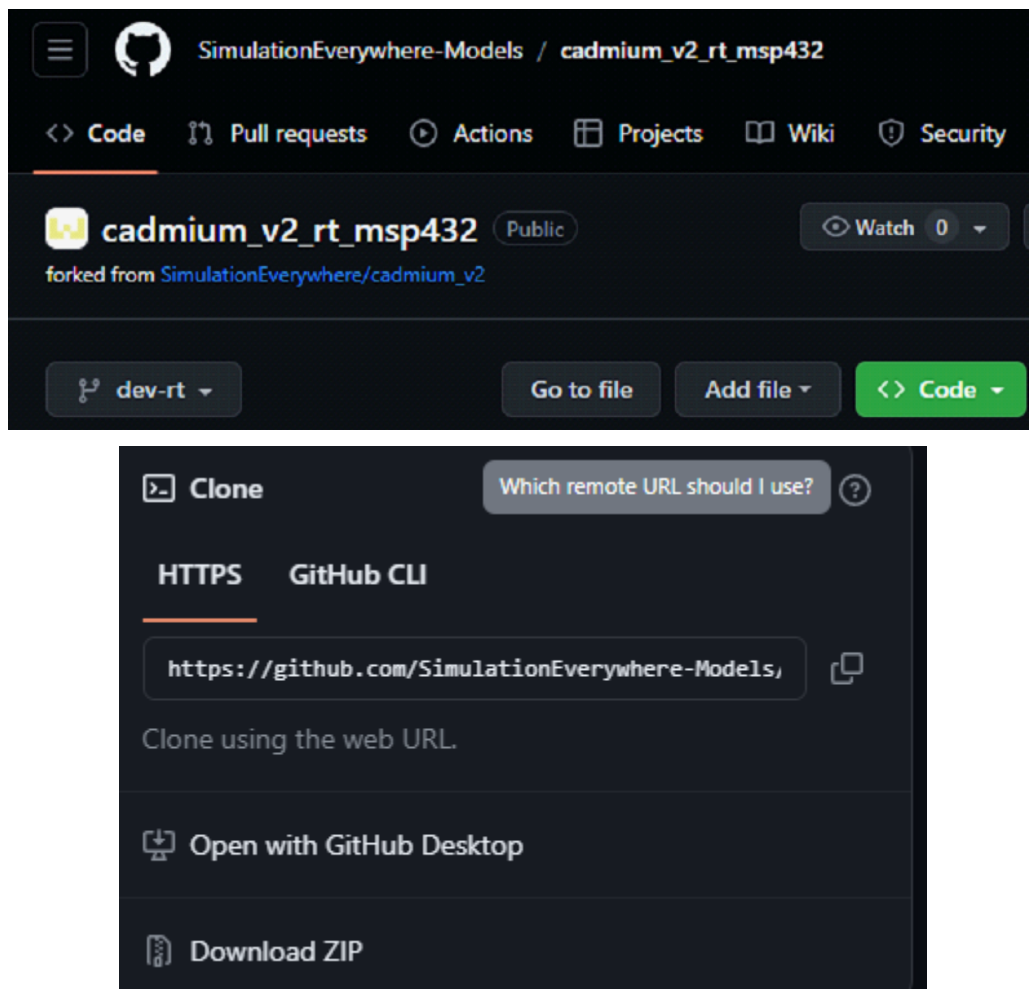


Figure 18. Example Projects GitHub

Extract the zip folder to a known location (such as the Desktop).

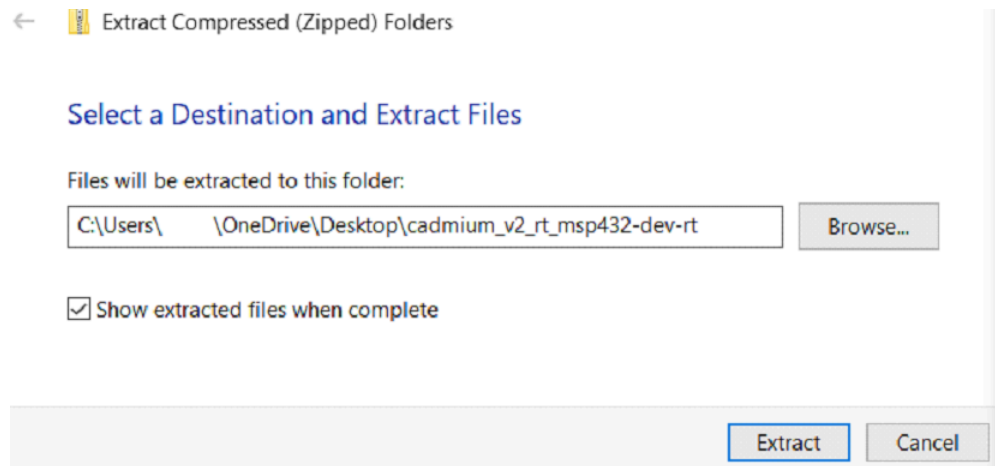


Figure 19. Extracting Project Zip Folder

Open Code Composer Studio IDE, if it is not already open. We will first set our workspace in Code Composer Studio. When we set the workspace for the IDE, we are choosing the location of where all of our project files are stored.

It is important to select the correct folder (rt_msp432) from the extracted zip folder downloaded previously so that we can share the same set of files (code) for both simulation and embedded execution.

Set the workspace via File → Switch Workspace... → Other... Select Browse... then navigate to the extracted folder, and select the folder: 'cadmium_v2_rt_msp432-dev-rt/example/rt_msp432'

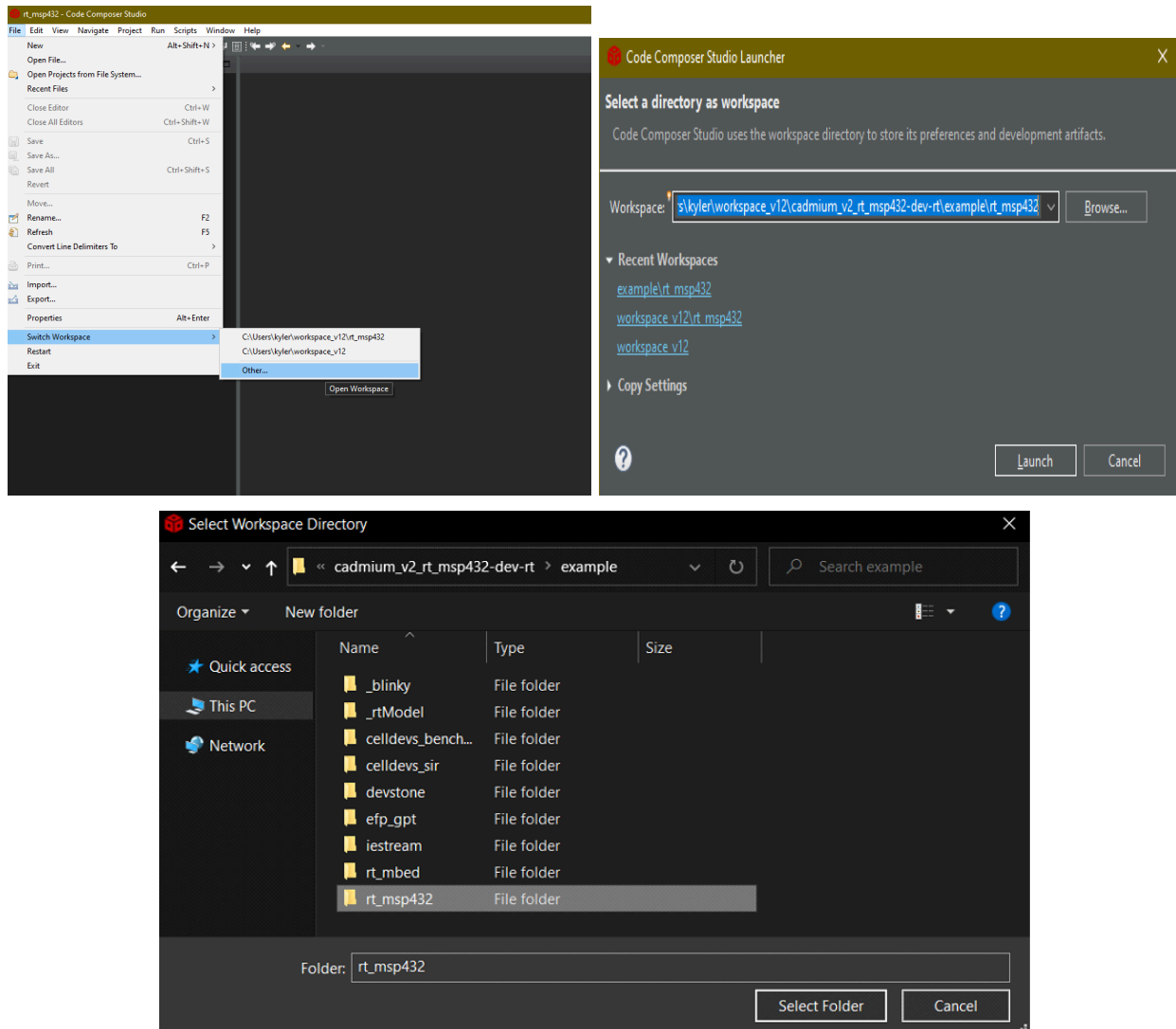


Figure 20. Selecting rt_msp432 as Workspace

Select 'Launch' once rt_msp432 is chosen as the workspace. The IDE will restart using the new workspace.

When the IDE is on again, we can open a sample project via Project → Import CCS Projects...

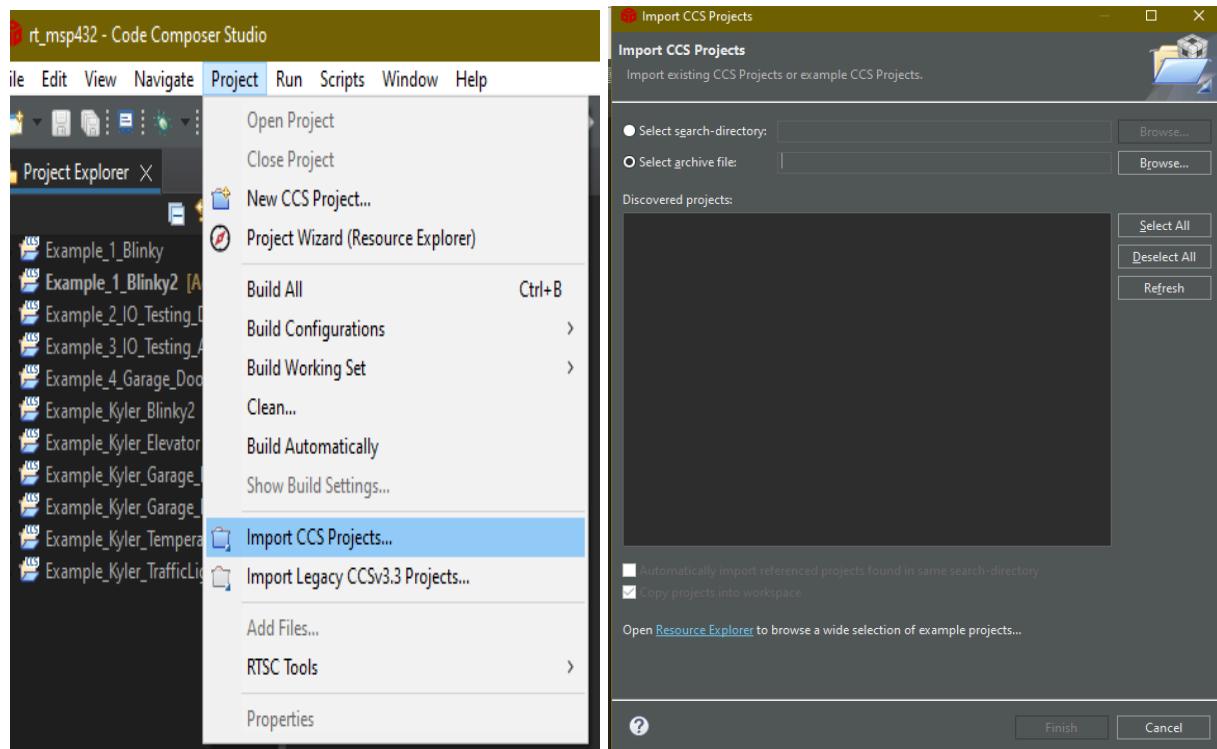


Figure 21. Import CCS Projects Window

Select the 'Browse...' button that corresponds to 'Select archive file'. What should be expected is a window opening up with the chosen workspace folder, we will see a series of zip folders containing example projects, and some additional library and Code Composer folders.

Navigate to and select the zipped folder:

'cadmium_v2_rt_msp432-dev-rt/example/rt_msp432/Example_1_Blinky', select 'Open'. Select 'Finish', this will import the Blinky zip folder, creating a new unzipped folder with the same name in the process.

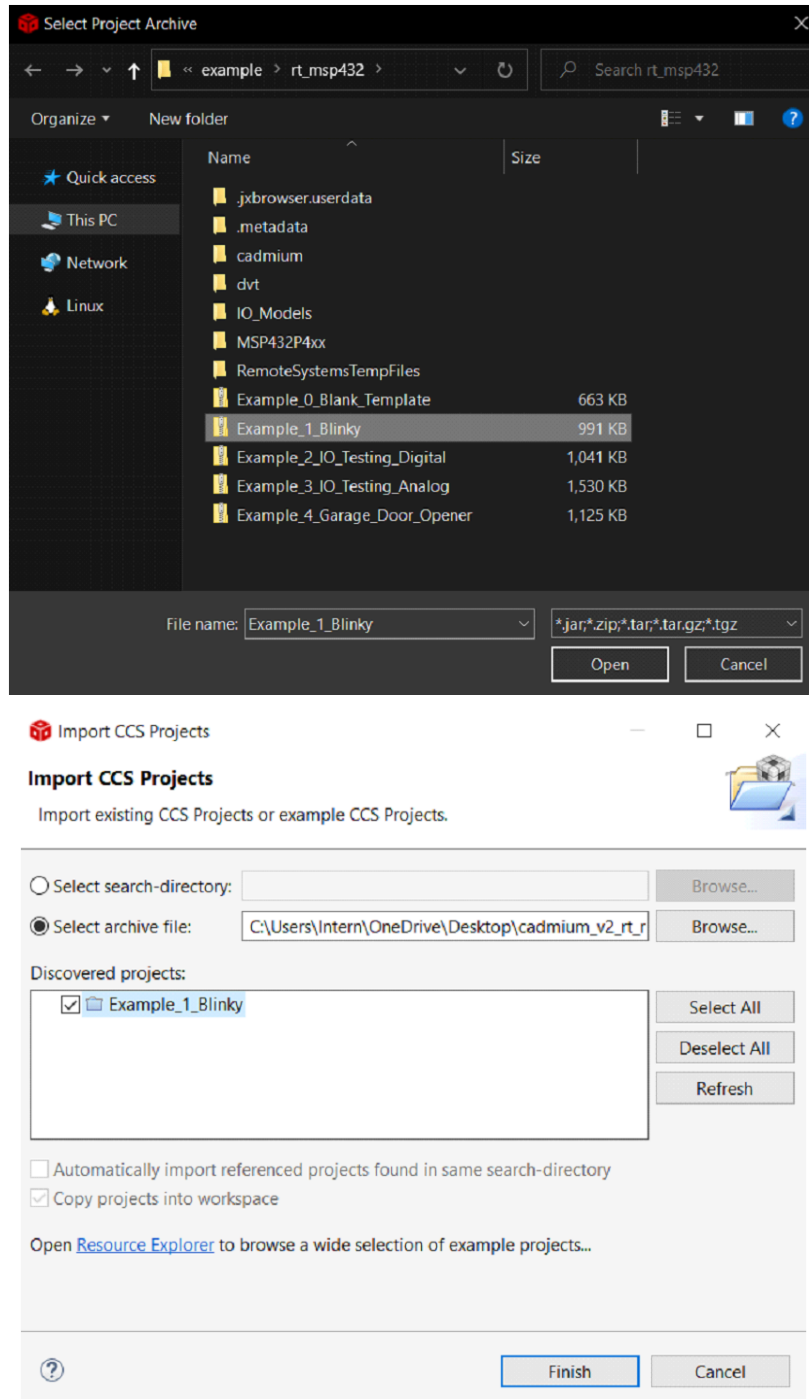


Figure 22. Import Blinky Example Zip Folder

On the left side of the IDE, the 'Project Explorer', should contain the 'Example_1_Blinky' project folder. Opening it should contain the contents displayed in Figure 23.

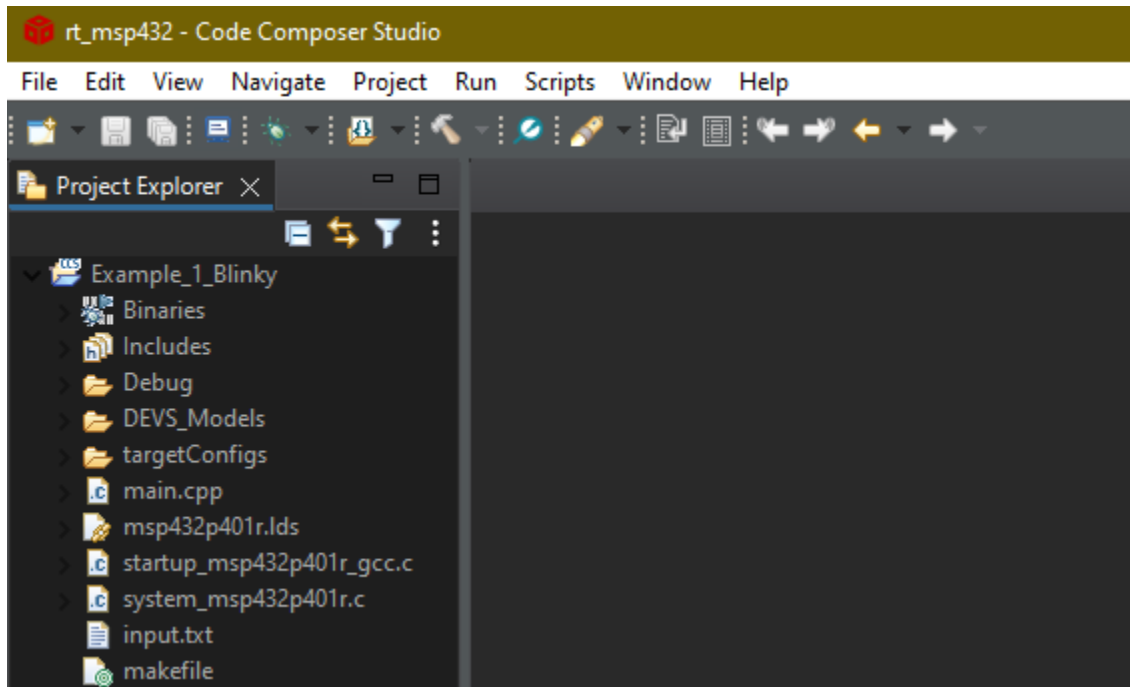


Figure 23. Project Folder Contents

Right click on the project name, and select 'Show Build Settings...'. On the left side of the dialog box that appears, expand 'Build', followed by 'GNU Compiler'. Then select the 'Directories' menu.

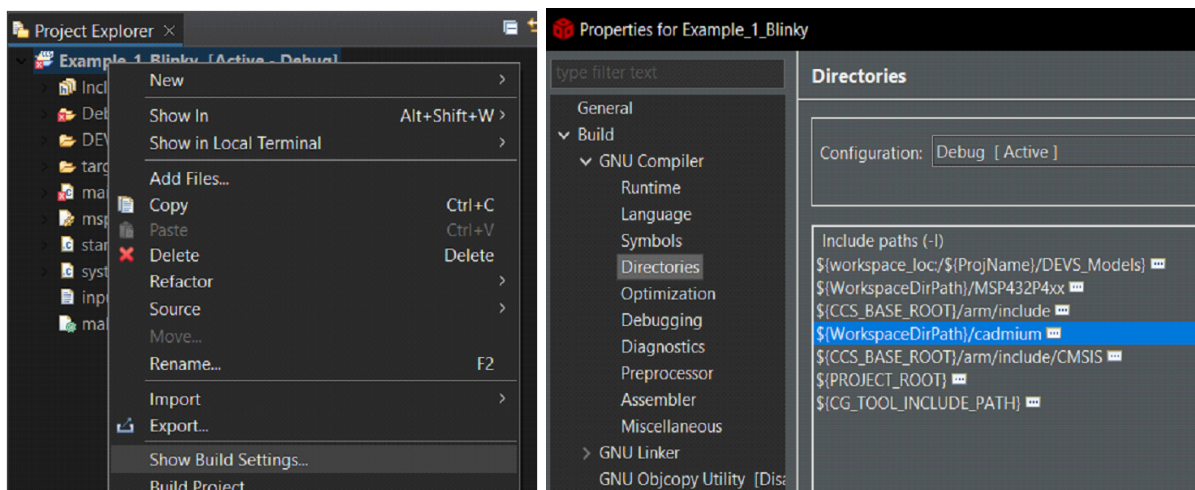


Figure 24. Project Build Settings

Double click on ‘\${WorkspaceDirPath}/cadmium’ to edit it. Select ‘Browse...’, and navigate to the downloaded repository. Double click on ‘include’ then single-click on the ‘cadmium’ folder and select ‘Select Folder’. This folder contains the actual simulation software which will be used to run our program.

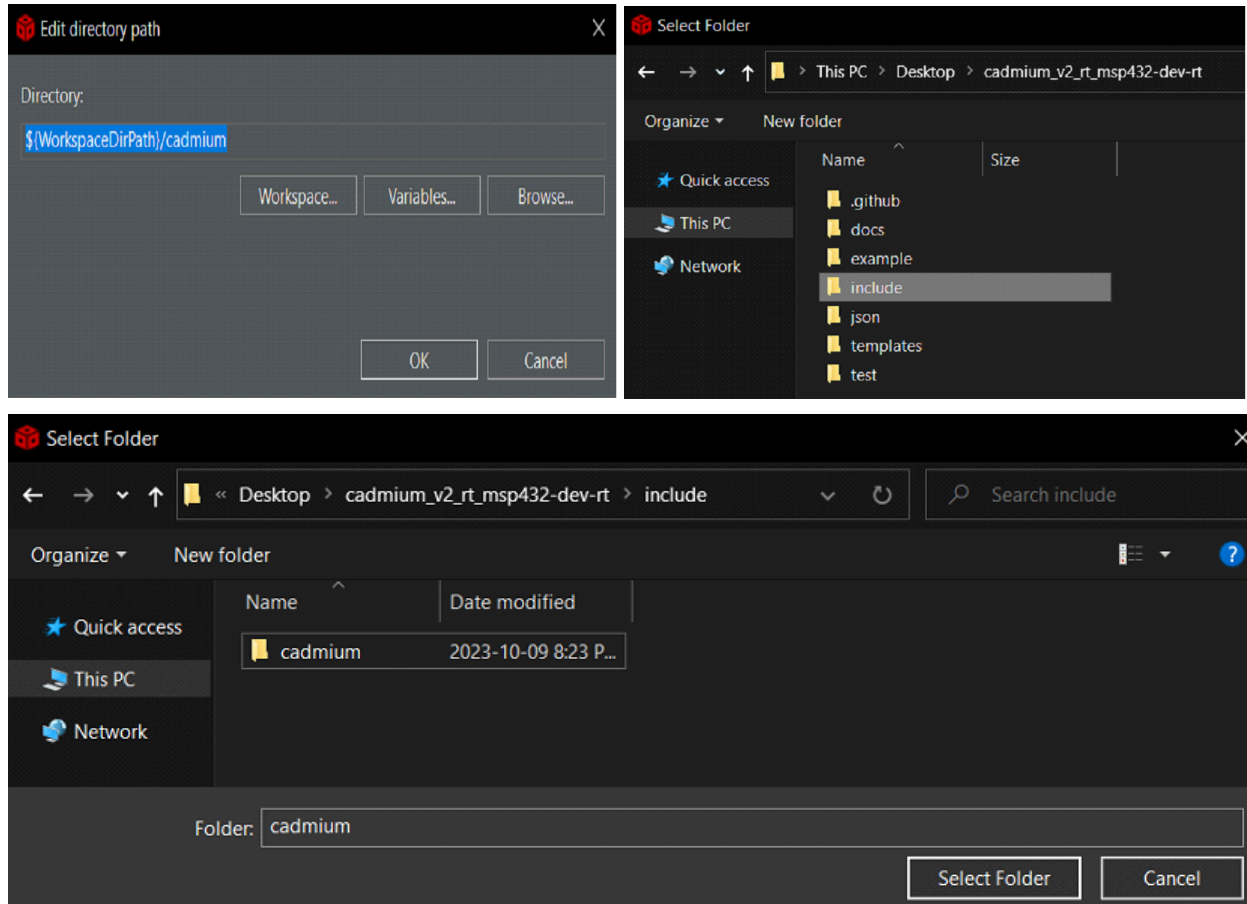


Figure 25. Edit Directory Path Windows

The updated directory should be displayed in the directory list. Select ‘Apply and Close’.

Creating New Projects From Scratch

The folder and files that are of focus when creating a new embedded system is the DEVS_Models folder and the hpp files within the folder. These files contain all the logic and interactions that the embedded system will have. The main.cpp file is also important, as it contains values that are essential for running simulations of the system. The input.txt file(s) are used for simulating inputs that would be done with the microcontroller. View Figure 23.

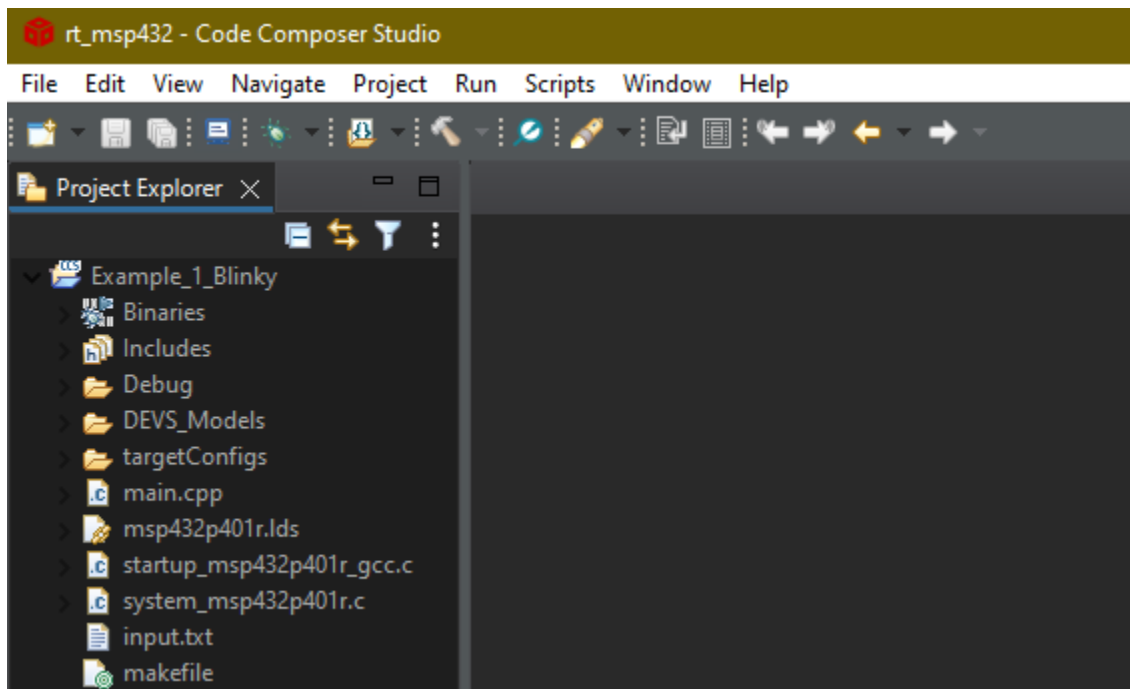


Figure 23. Project Folder Contents

When creating a new project folder, it's recommended to copy a working project folder that uses Cadmium and paste it into the workspace (the project explorer). This is to ensure that your new project folder will have all the necessary files used for running simulations with Cadmium.

With the copied project folder, rename all files and lines of code that contain names of the contents found in the original project folder to names desired for this new project folder. **For this guide, the blinky example will be used as an introduction to the process of creating a new system.**

In the blinky projects DEVS_Models folder are two hpp files, blinky.hpp and blinkySystem.hpp. These two files are models, with blinky.hpp being an atomic model and blinkySystem.hpp being a coupled model. Atomic models take in input and give out output. Coupled models handle connections between inputs and outputs amongst atomic models and the microcontroller.

This guide will start with inspecting the **blinky.hpp** atomic model. Figure 26 displays the initial lines of code found within the blinky.hpp atomic model.

```

1 #ifndef _BLINKY_HPP_
2 #define _BLINKY_HPP_
3
4 // This is an atomic model, meaning it has its' own internal logic/computation
5 // So, it is necessary to include atomic.hpp
6 #include <modeling/devs/atomic.hpp>
7
8 #if !defined NO_LOGGING || !defined EMBED
9 #include <iostream>
10 #endif
11
12 namespace cadmium::blinkySystem {
13     // A class to represent the state of this specific model
14     // All atomic models will have their own state
15     struct BlinkyState {
16
17         // sigma is a mandatory variable, used to advance the time of the simulation
18         double sigma;
19
20         // Declare model-specific variables
21         bool lightOn;
22         bool fastToggle;
23
24         // Set the default values for the state constructor for this specific model
25         BlinkyState(): sigma(0), lightOn(false), fastToggle(false) {}
26     };
27
28 #if !defined NO_LOGGING || !defined EMBED
29 /**
30  * This function is used to output the current state of the model to the .csv output
31  * file after each internal transition. This is used to verify the accuracy of the
32  * simulation.
33  *
34  * In this model, each time the internal transition function is invoked the current
35  * output from the out port is listed, followed by the model's current state for
36  * state.lightOn, and state.sigma.
37  */
38  * Note that state.sigma is NOT mandatory to include here, but it is listed due to
39  * the desired program logic.
40  *
41  * @param out output stream.
42  * @param s state to be represented in the output stream.
43  * @return output stream with sigma and lightOn already inserted.
44  */
45  std::ostream& operator<<(std::ostream& out, const BlinkyState& state) {
46      out << ", Status: " << state.lightOn << ", sigma: " << state.sigma;
47      return out;
48  }
49 #endif
50
51 // Atomic model of Blinky
52 class Blinky : public Atomic<BlinkyState> {
53 private:
54
55 public:
56
57     // Declare ports for the model
58
59     // Input ports
60     Port<bool> in;
61
62     // Output ports
63     Port<bool> out;
64
65     // Declare variables for the model's behaviour
66     double slowToggleTime;
67     double fastToggleTime;
68
69     /**

```

Figure 26. Blinky Atomic Model Initial Code

The atomic model begins with `#ifndef` and `#define` of the files name, followed up with essential lines such as `include atomic.hpp` and lines 8 to 10. These lines should not be touched, exception being the renaming of the first two lines to the new files name. Lines 12 to 26's format should not be changed, as this format is necessary for the coupled model to use the atomic model.

What should be changed within lines 12 to 26 are the names of the system and state to names that reflect the atomic model. Within the struct `BlinkyState`, consists of model specific variables and the setting of these variables default values. The sigma variable is mandatory for atomic models, as this value determines the amount of time the model will take to trigger its internal and external transition functions. Other variables can be added, removed, or changed, all being necessary when making a new system.

Lines 28 and 45 to 47 are necessary for generating a log file when running a simulation of the system. Lines 45 and 46 should be changed to reflect the new names and variables made. To retrieve the current value of a variable, `'state.variableName'` is used. **Figure 27 displays the next chunk of code, containing the class code, constructor function, and internal transition.**

```

51 // Atomic model of Blinky
52 class Blinky : public Atomic<BlinkyState> {
53 private:
54
55 public:
56
57     // Declare ports for the model
58
59     // Input ports
60     Port<bool> in;
61
62     // Output ports
63     Port<bool> out;
64
65     // Declare variables for the model's behaviour
66     double slowToggleTime;
67     double fastToggleTime;
68
69 /**
70  * Constructor function for this atomic model, and its respective state object.
71  *
72  * For this model, both a Blinky object and a BlinkyState object
73  * are created, using the same id.
74  *
75  * @param id ID of the new Blinky model object, will be used to identify results on the output file
76  */
77 Blinky(const std::string& id): Atomic<BlinkyState>(id, BlinkyState()) {
78
79     // Initialize ports for the model
80
81     // Input Ports
82     in = addInPort<bool>("in");
83
84     // Output Ports
85     out = addOutPort<bool>("out");
86
87     // Initialize variables for the model's behaviour
88     slowToggleTime = 3.0;
89     fastToggleTime = 0.75;
90
91     // Set a value for sigma (so it is not 0), this determines how often the
92     // internal transition occurs
93     state.sigma = slowToggleTime;
94
95 }
96
97 /**
98  * The transition function is invoked each time the value of
99  * state.sigma reaches 0.
100  *
101  * In this model, the value of state.lightOn is toggled.
102  *
103  * @param state reference to the current state of the model.
104  */
105 void internalTransition(BlinkyState& state) const override {
106
107     state.lightOn = !state.lightOn;
108
109 }
110

```

Figure 27. Blinky Atomic Model Class and Internal Transition

The format of the class code should not be changed. Within the class code contains a constructor function, internal and external transition functions, output function, and time advance function. All functions within the class code are what determines the behavior of the atomic model. The start of the class code is the declarations of the input and output ports along with variables for the model's behavior. In this example, blinky only has one input and output port, both being of type bool. Two variables are also declared, which will be used for the model's behavior.

Following the declarations is the constructor function. Within the constructor function is initialization of the ports and variables just declared. It is essential that the sigma value is also set to a value greater than zero within the constructor. In this example the sigma value is set to 'slowToggleTime' which happens to be 0.75, meaning every 0.75 seconds that passes, this model's internal and external functions will be run. This model's internal transition function sets the state of the model's bool variable 'lightOn' to the opposite of what it currently is, so every 0.75 seconds this variable will change from true to false and vice versa. Figure 28 displays the remaining functions in the class code.

```

112  * The external transition function is invoked each time external data
113  * is sent to an input port for this model.
114  *
115  * In this model, the value of state.fastToggle is toggled each time the
116  * button connected to the "in" port is pressed.
117  *
118  * The value of state.sigma is then updated depending on the value of
119  * state.fastToggle. Sigma is not required to be updated in this function,
120  * but we are changing it based on our desired logic for the program.
121  *
122  * @param state reference to the current model state.
123  * @param e time elapsed since the last state transition function was triggered.
124  */
125  void externalTransition(BlinkyState& state, double e) const override {
126
127      // First check if there are un-handled inputs for the "in" port
128      if(!in->empty()){
129
130          // The variable x is created to handle the external input values in sequence.
131          // The getBag() function is used to get the next input value.
132          for(const auto x : in->getBag()){
133              if (x==0)
134                  state.fastToggle = !state.fastToggle; // if the button was pressed, we change the speed of toggling
135              }
136
137              if(state.fastToggle){
138                  state.sigma = fastToggleTime;
139              }
140              else{
141                  state.sigma = slowToggleTime;
142              }
143          }
144      }
145  }
146
147  /**
148  * This function outputs any desired state values to their associated ports.
149  *
150  * In this model, the value of state.lightOn is sent via the out port. Once
151  * the value of state.ligthOn reaches the I/O model, that model will update
152  * the status of the LED.
153  *
154  * @param state reference to the current model state.
155  */
156  void output(const BlinkyState& state) const override {
157
158      out->addMessage(state.lightOn);
159
160  }
161
162  /**
163  * It returns the value of state.sigma for this model.
164  *
165  * This function is the same for all models, and does not need to be changed.
166  *
167  * @param state reference to the current model state.
168  * @return the sigma value.
169  */
170  [[nodiscard]] double timeAdvance(const BlinkyState& state) const override {
171
172      return state.sigma;
173
174  }
175  };
176 } // namespace cadmium::blinkySystem
177
178 #endif // __BLINKY_HPP__
179

```

Figure 28. External Transition and Output Functions

The external transition function handles expected behavior when the model receives an input. The initial if statement is necessary for detecting if an input was sent into the specified input port. Within the if statement is a for loop, this loop dictates what the model will do when input is received. In this example, when input is detected, the state of the 'fastToggle' bool variable will change from true to false and vice versa. After, an if else statement is used to change the sigma value to reflect the current value of 'fastToggle'. To summarize, when input is received, the sigma value will be toggled between 0.75 and 3.0 seconds, affecting the amount of time for the internal transition function to be run.

The output function contains all the output ports with a 'addMessage' function that takes in a state of a variable. In this example, the out output port will output the state of the 'lightOn' bool variable. The coupled model will handle the output value. The timeAdvance function should not be changed other than renaming the state const.

Figure 29 displays the coupled models code.

```

1 blinkySystem.hpp x blinky.hpp c main.cpp
2 #ifndef BLINKY_SYSTEM_HPP
3 #define BLINKY_SYSTEM_HPP
4
5 // This is a coupled model, meaning it has no internal computation, and is
6 // used to connect atomic models. So, it is necessary to include coupled.hpp
7 #include <modeling/devs/coupled.hpp>
8
9 #ifdef EMBED
10 #include ".../IO_Models/accelerometerInput.hpp"
11 #include ".../IO_Models/digitalInput.hpp"
12 #include ".../IO_Models/digitalOutput.hpp"
13 #include ".../IO_Models/joystickInput.hpp"
14 #include ".../IO_Models/lcdOutput.hpp"
15 #include ".../IO_Models/lightSensorInput.hpp"
16 #include ".../IO_Models/microphoneInput.hpp"
17 #include ".../IO_Models/pwmOutput.hpp"
18 #include ".../IO_Models/temperatureSensorInput.hpp"
19 #else
20 #include <lib/istream.hpp>
21 #endif
22
23 // We include any models that are directly contained within this coupled model
24 #include <blinky.hpp>
25
26 namespace cadmium::blinkySystem {
27     class blinkySystem : public Coupled {
28     public:
29         blinkySystem(const std::string& id): Coupled(id){
30
31             // Declare and initialize all controller models (non-input/output)
32             auto blinky = addComponent<Blinky>("blinky");
33
34             // Connect any non-input/output models with coupling
35             // (NOT APPLICABLE FOR THIS MODEL)
36
37             #ifdef EMBED
38                 // Declare and initialize all embedded input/output models
39                 auto digitalInput = addComponent<DigitalInput>("digitalInput",GPIO_PORT_P1,GPIO_PIN1);
40                 auto digitalOutput = addComponent<DigitalOutput>("digitalOutput",GPIO_PORT_P1,GPIO_PIN0);
41
42                 // Connect IO models with coupling to the system
43                 addCoupling(digitalInput->out,blinky->in);
44                 addCoupling(blinky->out,digitalOutput->in);
45             #else
46                 // Declare and initialize all simulated input files (these must exist in the file system before compilation)
47                 auto textInput = addComponent<cadmium::lib::IStream<bool>>("textInput","input.txt");
48
49                 // Connect the input files to the rest of the simulation with coupling
50                 addCoupling(textInput->out,blinky->in);
51             #endif
52         }
53     };
54 } // namespace cadmium::blinkySystem
55 #endif // BLINKY_SYSTEM_HPP
56

```

Figure 29. Blinky Coupled Model Code

All the include statements and general format of the coupled model should not be changed. It is important that every atomic model is included in the coupled model, as shown in line 23 of Figure 6. Within the namespace and class, it consists of all the declarations of atomic models and input output ports of the microcontroller.

Important to note the if else statement of whether the system is 'EMBED'. When 'EMBED', this means the system is being run on the microcontroller. In this scenario we declare the ports of the microcontroller. The port and pin numbers for the microcontroller can be found on the pins of the device. If the system isn't being run on the microcontroller then the else will be called, meaning the system is being simulated to generate a csv log file.

The main purpose of the coupled model is to add coupling to all the input and output ports of the atomic model(s) and microcontroller. When 'EMBED', coupling is added to the blinky.hpp atomic models in and out ports to the digital input and output pins of the microcontroller. When the user presses down on the side button of the microcontroller, this input will be sent to the 'in' input port of blinky.hpp. The out output port of blinky.hpp is coupled to the digital output pin of the microcontroller, resulting in the microcontroller's red led pin to turn on and off corresponding to the output of blinky.hpp's out output port value.

To run the system on the microcontroller, first build debug the project by clicking the hammer icon as shown in Figure 30. When clicked the system will build and check for any errors. If no errors are shown after building then it is possible to run the system onto the microcontroller. To do so, first click the green bug button shown in Figure 30.

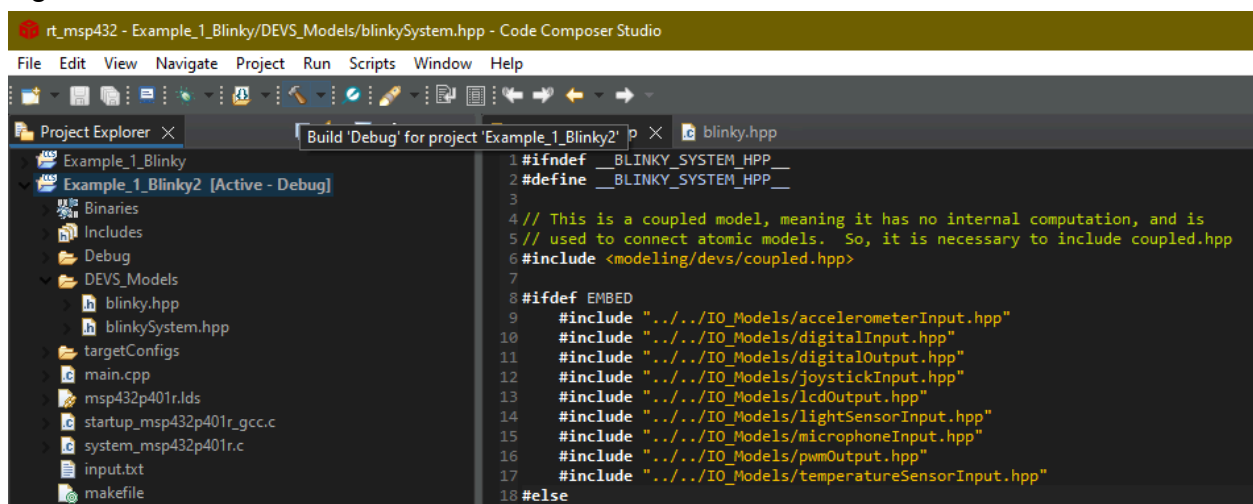


Figure 30. Build and Debug Buttons

When clicked the window will change to what is displayed in Figure 31. The last step is to click the green play button. If successful, the microcontrollers 'LED01' pin will turn on and off a red LED every 0.75 seconds. To toggle the speed of this behavior, press the button located on the left side of the microcontroller. To close the debug window and return to the default view, click the red square button to stop the debug session.

Important to note that the system behavior will continue on the microcontroller even after terminating the debug window. To change this behavior to a different build simply debug the system again with the microcontroller plugged in, this will overwrite the memory stored on the microcontroller to the new build.

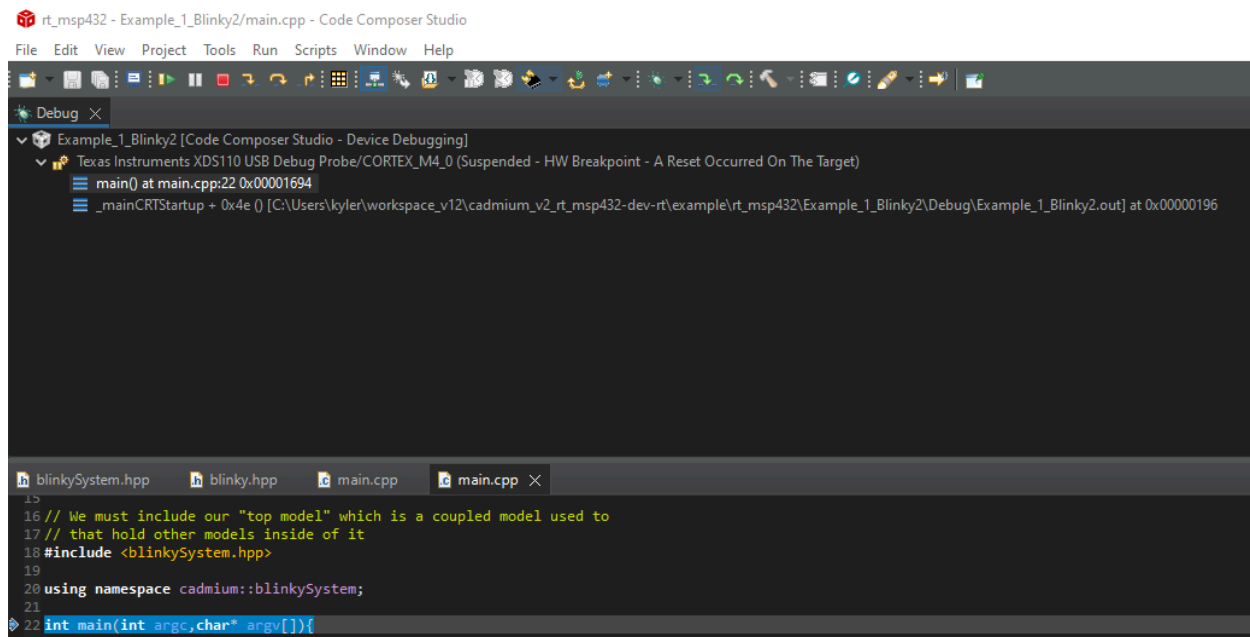


Figure 31. Debug Window

To run a simulation that will generate a csv log file, first add coupling with an input text file to the atomic model input ports that would take input from the microcontroller. This can be seen in Figure 6 on lines 46 to 54. Next, right click on the project folder in the project explorer and then click 'Show In System Explorer'. A window will open with where you have the project folders stored, open the project folder and within should contain input text file(s) as shown in Figure 32.

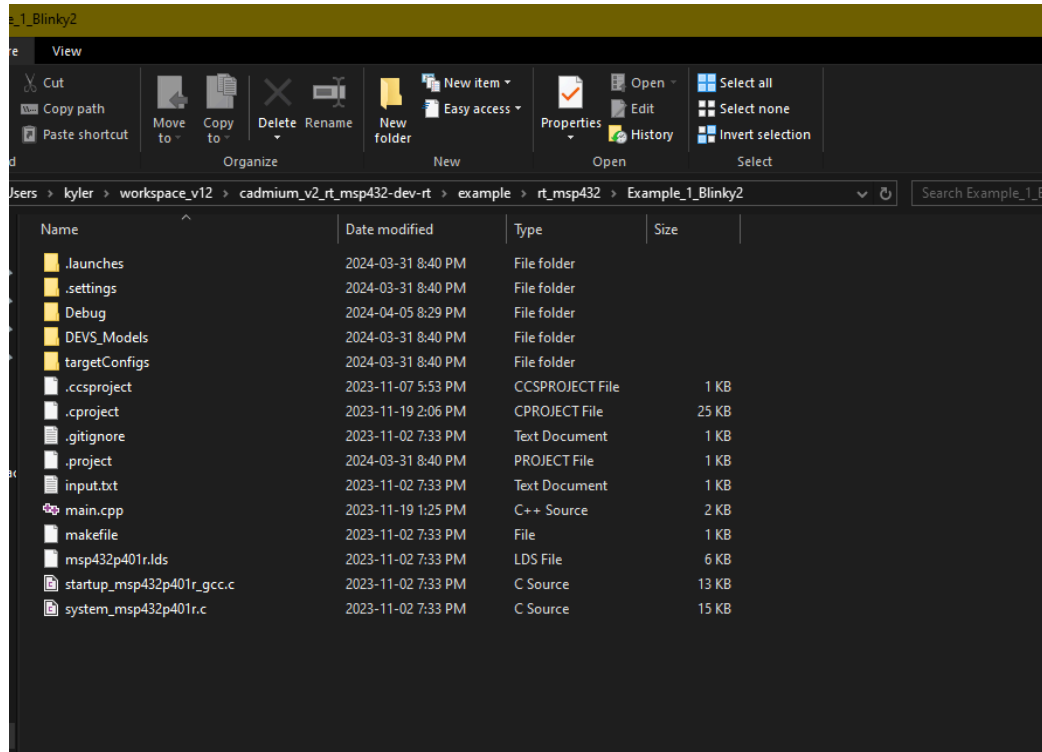


Figure 32. Project Folder Contents

To make a csv log file, right click within the project folder and click 'Bash Prompt Here'. A cygdrive window will open, within this window input 'make'. This will generate an exe file. In this example, titled 'Blinky.exe'. Next, input './Blinky.exe' to create the csv log file. The log file will display the 'std::ostream' info found in the atomic model (Figure 3, lines 45 to 47), along with info on input and output port status'. This is all shown in Figure 33.

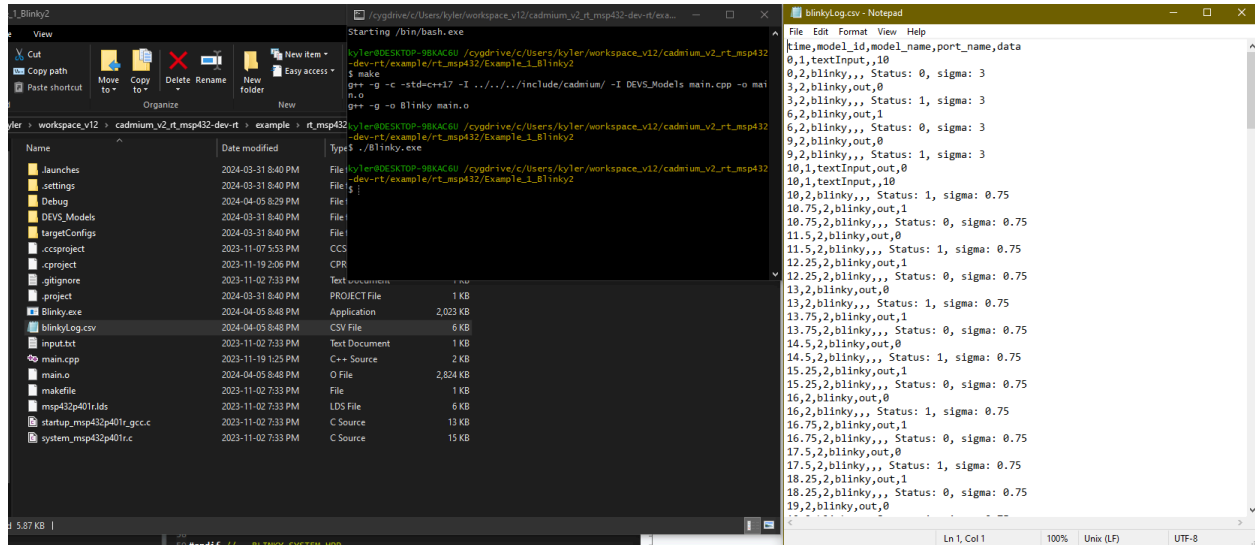


Figure 33. Make Log File

It's important to note that the command 'make clean' must be inputted after generating the log file. This will delete the log file and the exe file in the projects folder. to keep the log file intact, copy or move it to a different folder. If 'make clean' is not inputted, future simulations and debugs will result in an error.

To get a stronger grasp on the software and begin developing entire systems from scratch it's recommended to look over simple examples such as the one used for this guide and then add or change inputs and outputs to the system. After developing a strong base foundation on the software, move up in complexity by looking into systems that use two or more atomic models and various other input output pins found on the microcontroller.