Git

|  |
| --- |
| Installing Git  <https://git-scm.com/downloads>  Select Windows, Mac or Linux. |
| Setting up a Local Repository  This is largely a personal preference. My preferred organization is to create a folder called ‘\_LocalRepo’ and maintain subfolders of the different project types.  The Local Repository is where the working copy is. The Version Controlled copy of the project is managed by git and your chosen client. I use GitHub.  This means when I’m done with a project, or just want to save space on my local machine – I can delete the project from my Local Repository and restore it from Git if I need to pick it up again. |
| Remapping default path in Command Prompt or PowerShell |
| New Project Workflow (gitting started cheatsheet) |
| Push or “Save” your code to the remote repository (eg GitHub)   1. Add the changed files to the commit >git add . 2. Commit the added files with a note >git commit -m “Added / Updated / Removed / Notes” 3. Push the commit to the remote repository >git push origin master |
| Avoid tracking unnecessary files.  Some files are created in the IDE and don’t need to be tracked, or pushed up. Ignore these files with a .gitignore file.  GitHub will let you add this when you create a repository:  Otherwise, add a new text document named “.gitignore”    In it, add the extensions or paths to be ignored by git. The “#” are comments:    Add, commit and push to the repo. |
| Create a repository and add a project to it |
| Add an already existing project to a repository |
| Accidentally pushing to the wrong branch:  Eg: Checkout branch ‘addi2c’, intended to commit to branch ‘addi2c’ but accidentally pushed commit to ‘master’   1. Checkout the branch committed to ‘master’   >git checkout master   1. Undo commit   >git reset ~HEAD1   1. Stash changes   >git stash   1. Checkout the intended branch   >git checkout hotfix   1. Bring the changes back into the new branch   > git stash pop   1. Re-add, commit and push   >git add .  >git commit -m “added i2c feature”  >git push origin addi2c |
| Using branches to try something in the code.  Before making edits to the branch currently working on:  >git checkout -b newbranchname  Refresh the code, and continue working.  When pushing up to the repository, make sure to push to **newbranchname**  >git push origin **newbranchname** |
| Made changes to the code, and now I want to commit them to a new branch  Eg: Intended to only change one thing, ended up refactoring a significant part of the code and would rather not commit it to the main or current branch.   1. Set the changes aside   >git stash   1. Checkout a new branch   >git checkout -b newbranchname   1. Bring the changes back into the new branch   >git stash pop   1. Re-add, commit and push to the **NEW** branch   >git add .  >git commit -m “Refactored communication protocol”  >git push origin **newbranchname** |
| Discard a branch that did not pan out  We have branch tryfeature that we no longer want. Simply delete the branch.   1. Checkout some other branch (eg the master). If the changes in the current branch haven’t been committed, there may be an error/warning.    1. Put the changes aside   >Git stash .   * 1. Checkout a branch other than the one to be deleted.   >git checkout master   1. Delete the branch to be removed. > git branch -d tryfeature   \*\*This will permanently delete this branch. All changes made in that branch that weren’t merged or otherwise incorporated into the branch will be gone. You may want to run a code compare, or just leave the branch and ignore it. \*\* |
| Incorporate a branch that did pan out (“Merging”)   1. Checkout the branch to merge *into.* Eg “master” >git checkout master 2. Merge the branch with the fix *into* master >git merge hotfix master 3. Git will display the changes, or any errors that need to be corrected. 4. Optional: Delete the branch you merged *from* (if desired, not necessary). >git branch -d hotfix |
| See the commits and branches on a project:   1. To see the brief information about a project’s commits: >git log –oneline 2. To see more detailed information: >git log 3. Press ‘Enter’ to scroll through all the commits until the text (END) appears at the bottom. 4. To leave the log, press ‘q’ |
| Contributing to someone else’s project |
| What changed between these commits / branches? |
| Adding a second repository  If you have multiple git repositories to commit your projects to – maybe a corporate and a personal, or a public and a private… add a second repo to your project and automatically push to both.  <https://docs.aws.amazon.com/codecommit/latest/userguide/how-to-mirror-repo-pushes.html>   1. Look at your current destinations for fetch and push: >**git remote -v** 2. Add a second destination to push with the following command where *git-repository-name* is the URL of the second destination repository to add.   **>git remote set-url --add --push origin***git-repository-name*   1. Look at the new destinations for fetch and push to validate that it added >>**git remote -v** 2. >git push --set-upstream origin *branchname* |

# Firmware

|  |
| --- |
| A Useful ReadMe Template in Markdown (and how to view it) |
| A useful function header template:  /\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*  \* @fn - nameOfFunction  \* @brief - Handles the Control user button presses  \* @param[in] - GPIO\_Pin : GPIO Pin number  \* @return - void  \*  \* @note - Identify which button was pressed and Update data packet  \* GPIO pin defines in main.h  \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/  **void** nameOfFunction (uint16\_t GPIO\_Pin)  {  // Function Actions  } |
| STM32 Software to Install   * STM32CubeIDE (Development Environment) * STM32CubeMX (IO Configuration tool. Most of these function can be done in CubeIDE, but some require CubeMX) * ST-LINK Utility (To program boards with \*.bin files – eg production/semi-production) * STM32CubeProgrammer (To program boards with \*.bin files – eg production/semi-production) |
| STM32 Manuals to Use with the MCU |
| Setting up Dev Board |
| Decoding UART with a Logic Analyzer |
| Programming boards with ST-LINK Utility  STM32 ST-LINK Utility is required for automatic programming.  Download and install: <https://www.st.com/en/development-tools/stsw-link004.html>   1. Perform these steps after completing Fixture Setup. An ST-LINK programmer/debugger needs to be connected:  <https://www.st.com/en/development-tools/st-link-v2.html> 2. Start the ST-LINK utility. 3. Drag and drop the desired target program into the gray empty space of the ST-LINK utility, under the Device Memory field. The window should populate with the hex of the target program binary. 4. On the menu bar, navigate to Target > Settings. On the window that appears, confirm an ST-LINK serial number is populated to indicate the programmer is detected. 5. Change the setting under Mode to “Connect under Reset” and check “Enable debug in Low power mode” if not already set. Click OK to close settings window. All other settings should remain as default. An error may appear if a board is not connected and/or powered. It can be disregarded. 6. On the menu bar, navigate to Target > Automatic Mode. On the window that appears, check that the File field matches the desired target application. Check the following options:     1. Flash programming    2. Verify (verify while programming)    3. Run application 7. Click Start. The fixture is now ready to detect a board being inserted into the fixture. Upon insertion, the programmer will flash and run the program. |
| Generate a binary file from project to program MCUs with in STM32CubeIDE  In CubeIDE go to Project Settings -> "C/C++ Build" group -> Settings -> "Tool Settings" tab -> MCU Post build outputs -> "Convert to Intel Hex file" check box  [Hex file option in CubeIDE](https://i.stack.imgur.com/wUowJ.png)  This will make the IDE convert the output into HEX-file, which is easily parsable. You can find [the format description in Wikipedia](https://en.wikipedia.org/wiki/Intel_HEX). You can parse it before sending to the bootloader.  Or, you can set the checkbox "Convert to binary file", which will make a raw binary file. But it may give some troubles if your code starts not from zero address. |

# PCB Software

|  |
| --- |
| Useful Library Organization in Altium/Circuit Studio |
| Snippets |
| Simulation Models |
| PCB Fab Notes |
| Gerber Notes |
| ANSI Standards |
| ITAR or Not  International Traffic in Arms Regulations (**ITAR**) govern the export of weapons and weapons-related technology from the United States. In terms of PCBs, it controls Quality and Data Security.  ITAR compliant PCBs have stricter regulatory restrictions particularly in terms of Quality and Data Security, and it requires a registered and approved PCB manufacturer. Due to the stricter regulations and increased testing, it often means increased cost.  <https://www.pcbdirectory.com/manufacturers?country=United+States&certifications=ITAR>  In general, any controlled tech data or PCBs used in aerospace, defense and medical applications are required to meet ITAR compliance.  Your company or client may require ITAR compliance to not allow uncontrolled access to sensitive technical data.  <https://www.pmddtc.state.gov/regulations_laws/itar.html> for more info |
| Gerber Checklist |
| Schematic, PCB and Library Templates |
| Assembly Notes and Instructions |
| Circuit Studio Simulation  How to setup and run circuit simulations in Circuit Studio  Prerequisite: Have an Altium Live account.  If you have purchased Circuit Studio, register for the free account. Otherwise, your administrator will need to register you. If that fails, contact Altium representatives who can help.   1. Start a new schematic project 2. Add the simulation libraries to the project 3. Libraries -> Project -> Add Library -> Simulation Libraries 4. Select Simulation libraries \*.IntLib (component view) and place appropriate sources, or other simulation components. 5. Double click on any component to open the parameters of the part and edit the model. E.g., the source, a resistor, transistor… etc.      1. Tools -> Setup XSPICE Simulator 2. Select the type of simulation and edit as desired. Add signals to watch to the Active Signals list by clicking on them in ‘Available Signals’ and using the right arrow to move them to the ‘Active Signals’ list. 3. Ok. 4. Tools -> Run XPSICE Simulator Will open a new tab with graphs of the Active Signals and other data.  This can be saved as part of the project. |

|  |
| --- |
| A compiler is a program that converts instructions into a machine-code or lower-level form so that they can be read and executed by a computer.  When a program is compiled, the compiler converts the source file into object byte code. This byte code (sometimes called object code) is mnemonic instructions that only your computer architecture understands.  When the compiler compiled your program into an OBJ file, it simply puts a reference to the functions like “printf” or any other functions in other files. The linker resolves this reference. The linker links your compiled (e.g. OBJ) file with this standard library. The linker can also link your OBJ file with other OBJ files.  The linker works almost like copy and paste. It "copies" out all the necessary functions that your program references and “pastes” them to create a single executable.  Sometimes other libraries that are copied out are dependent on yet other OBJ or library files and the linker has to get pretty recursive to do its job. |
| Inline Function:  Put the (short) function in the header file instead of just the prototype. Specify with “Inline” above the function.  Functions.hpp  Inline  Void writeme(char keypress)  {  std::cout << keypress << "\n";  } |
| Default parameters can be set in the function prototype and will be overwritten if otherwise defined in the call.  Function.hpp  void coffee(bool milk = false, bool sugar = false);  main.cpp  coffee(); //will by default have milk and sugar as false  coffee(true); //will have milk as true, and by default sugar as false. |
| **Overloaded Functions** are multiple functions with the same name that take different types, or number of parameters.  Each variant of the function gets it’s own prototype in a header file, and it’s own definition.  void print\_key(char let);  void print\_key(int num);  The appropriate function will be used depending on the call:  print\_key (‘A’); or print\_key (4);  void print\_key(char let) {   std::cout << " " << let << "   " << let << " " << "\n"; }  void print\_key(int num) {  for (int I = 0; I < num; i++)  {  std::cout << " A “ << "\n"; }  **Templates** allow data types to be added as parameters.  Eg in numbers.hpp  template <typename T>  T get\_smallest(T num1, T num2)  {      return num2 < num1? num2 : num1;    } |
| Classes: An object gets it’s characteristics and behaviors from a class.  Eg int age;  “int” is a class.  “age” is an object.  “age” has certain properties – eg age++; There are no decimal places, the length is guaranteed to be at least 16 bits… and behaviors (eg what happens when you divide int 3 / int 2?)  Eg  Std::String myData = “some data”;  “Std::string” is a class  “myData” is an object  Components of a class are class members. There are attributes (member data to associate with an object – eg a length) and methods (member functions). Methods use a . in front of the name to distinguish them from regular functions (eg .length() );  Eg  myData.length();  “.length” is a class member of type method. It likely returns the attribute “length” (see how methods use attributes in the example below)  Define methods outside a class by using ClassName:: before the method name to indicate its class like this:  int City::get\_population() {   return population; }  We need to #include the header file in the **.cpp** file where we define the methods.  Instantiate (create) the object before using it.   |  |  |  | | --- | --- | --- | | Main.cpp | Func.hpp | Func.cpp | | #include <iostream>  #include "song.hpp"  int main() {    Song folk;    folk.add\_title("ZenHen");    std::cout << "Now playing: " << folk.get\_title() << "\n";  } | #include <string>  class Song  {  std::string title;  public:      void add\_title(std::string new\_title);      std::string get\_title();  }; | #include "song.hpp"  void Song::add\_title(std::string new\_title){    title = new\_title;  }  std::string Song::get\_title(){    return title;  } | |
| Public and Private Methods in a class  By default, everything in a class is private.  The main function cannot access or accidentally change these private variables or methods.  Usually, the attributes are kept private, and the methods to set/get them are public to allow specific access in specific ways to the variable attributes. |
| Constructors give *structure* to how an object of a class gets created.  Prototype this like a method in the header file, without a return type.  Define it like a method in the C file, without a return type  Song::Song(std::string new\_title, std::string new\_artist)  {  }  And then create it  #include <iostream>  #include "song.hpp"  int main() {    Song back\_to\_black("Back to Black", "Amy Winehouse");    std::cout << back\_to\_black.get\_title() << " by " << back\_to\_black.get\_artist() << "\n";    } |

|  |
| --- |
| **Creating and Debugging Arduino Programs in Visual Studio Code - Part 1** <https://www.codeproject.com/Articles/5150391/Creating-and-Debugging-Arduino-Programs-in-Visual>   1. Download the [Arduino IDE]( <https://www.arduino.cc/en/software>). \*\*Not\*\* the Windows Store App – it is not the full application. 2. Arduino for Visual Studio Code extension by Microsoft 3. Open the **Settings** – click the gear wheel in the Action Bar (bottom left) and select **Settings** from the menu. 4. In the **settings** window, type **Arduino**. 5. Locate the item Arduino: Path and enter the path to the folder where you placed the Arduino IDE. For example, *c:\Program Files (x86)\Arduino*. 6. There is no save. You’re all set. |
| **VSCode is Different.**  **Note 1**: Working with VS Code is somehow different from standard IDE. You perform most tasks by typing and selecting commands from so called Command Palette. This is a small text box which opens when you press **Ctrl+Shift+P** key combination or just **F1** key.  Also many options are set by editing text files (*.json* files) rather than by clicking buttons in the graphical user interface (GUI).  And there is no “project” to work with, rather you work with folders. But you can just think of a folder as a project.  **Note 2**: The Arduino examples are available in VS Code, just open the Command Palette (**Ctrl+Shift+P** or **F1** key) and type/select Arduino: Examples. |